HAW HAMBURG

MASTERTHESIS
Fabien Lapok

Synthetisierung von Audiosignalen mithilfe Neuronaler Netze am Beispiel von Vogelgesang

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Fabien Lapok

Synthetisierung von Audiosignalen mithilfe Neuronaler Netze am Beispiel von Vogelgesang

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Andreas Meisel
Zweitgutachter: Prof. Dr. Tim Tiedemann

Eingereicht am: 23. April 2021

Fabien Lapok

Thema der Arbeit

Synthetisierung von Audiosignalen mithilfe Neuronaler Netze am Beispiel von Vogelgesang

Stichworte

Deep Learning, Neuronale Netze, GANs, Wasserstein-GANs, Vocoder, Audiosynthese, Synthese von Vogelgesang, Mel-Spektrogramme, Python, Pytorch, FID-Score

Kurzzusammenfassung

Das Ziel der vorliegenden Arbeit ist die Konzeption, Implementierung und Evaluation einer Architektur auf Basis Neuronaler Netze für die Synthese von Vogelgesang. Die hier entwickelte Architektur besteht aus zwei Neuronalen Netzen: einem *Wasserstein-GAN*, der *Mel-Spektrogramme* mit Vogelgesang erzeugt und einem für die Sprachsynthese vortrainierten *Vocoder*, der diese *Mel-Spektrogramme* in Audiosignale in *Wellenform* umwandelt. In diesem Zusammenhang wird die Funktionsweise von *Generative Adversarial Networks (GANs)* und die darauf aufbauenden *Wasserstein-GANs* beschrieben. Anschließend werden Evaluationsmetriken für die Beurteilung der erzielten Ergebnisse ausgearbeitet. Im Kern geht die Arbeit der Frage nach, ob die Architektur bestehend aus den Komponenten *WGAN* und *Vocoder* sich für die Synthese von Vogelgesang eignet. Dazu werden verschiedene Konfigurationen dieser Architektur trainiert und die erzielten Ergebnisse quantitativ und qualitativ evaluiert. Im Rahmen der quantitativen Evaluation wird der Frage nachgegangen, ob der *FID-Score* ein plausibler Indikator für die Bewertung von generiertem Vogelgesang ist. Die qualitative Evaluation erfolgt durch den Autor der Arbeit über den Vergleich von generierten Daten mit Daten aus dem Trainingsdatensatz. Als Grundlage dieses Vergleichs dienen ermittelte *Nearest Neighbours* von generierten Daten zu typischen Mustern aus dem Trainingsdatensatz. Die durchgeführte Evaluation zeigt, dass die vorgestellte Architektur die Synthese von einfachen Signalen des Vogelgesangs ohne Qualitätsverlust ermöglicht. Bei komplexen Signalen sind erkennbare Qualitätsunterschiede festzustellen. Zusätzlich wird gezeigt, dass der *FID-Score* einen validen Indikator für das Messen des Trainingsverlaufs und der Bewertung der synthetisierten *Mel-Spektrogramme* darstellt. Die Ergebnisse der Arbeit können als Ausgangspunkt für weiterführende Forschungsarbeiten dienen. Die erzeugten Ergebnisse und Implementationen sind für die Öffentlichkeit bereitgestellt.

Fabien Lapok

Title of Thesis

Audio synthesis with neural networks using the example of bird song

Keywords

Deep Learning, Neuronale Netze, GANs, Wasserstein-GANs, Vocoder, Audio synthesis, synthesis of bird songs, mel spectrograms, Python, Pytorch, FID-Score

Abstract

The goal of this paper is to design, implement, and evaluate a neural network-based architecture for bird song synthesis. The developed architecture consists of two neural networks: A *Wasserstein GAN* that generates *mel spectrograms* of bird song and a vocoder, pre-trained for speech synthesis, that converts these *spectrograms* into audio signals. In this context, *Generative Adversarial Networks (GANs)* and *Wasserstein GANs* are described and based on this evaluation metrics for the assessment of the obtained results are elaborated. In essence, this thesis addresses the question of whether the architecture consisting of the components *WGAN* and *Vocoder* is suitable for the synthesis of bird song. For this purpose, different configurations of this architecture are trained and the obtained results are quantitatively and qualitatively evaluated. In the quantitative evaluation, the question of whether the *FID score* is a plausible metric for the evaluation of generated bird song is investigated. The qualitative evaluation is done by the author of the paper by comparing nearest neighbors of generated data to typical patterns from the training dataset. The evaluation shows that the presented architecture allows the synthesis of simple bird song without quality loss. More complex signals show a noticeable difference in quality. In addition, it is shown that the *FID score* is a valid indicator for measuring the training progress and the evaluation of the synthesized *mel spectrograms*. The results of the work can serve as a starting point for further research. The generated results and implementations have been made available to the public.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

Tabellenverzeichnis

Abkürzungen

FID Fréchet Inception Distance.

GANs Generative Adversarial Networks.

MOS Mean Opinion Score.

STFT short-time-(fast)-fourier-transformation.

VAE Variational Autoencoder.

WGANs Wasserstein - Generative Adversarial Networks.

1 Einleitung

Neuronale Netze haben sich in unterschiedlichen Domänen als eine exzellente Methode erwiesen, neue Daten zu erschaffen und kreative Aufgaben zu bewältigen. Liu et al. (2018) stellen ein Neuronales Netz vor, welches zu Bildern passende Gedichte entwirft, Hewahi et al. (2019) zeigen, wie *LSTM-Netze* Musik in MIDI-Form generieren und Gatys et al. (2016) ermöglichen, neue Bilder in der Zeichenart bestimmter Künstler zu erzeugen.

Die jüngsten Entwicklungen im Bereich der Sprachsynthese haben mit den Veröffentlichungen von van den Oord et al. (2016) und Shen et al. (2017) neue Maßstäbe gesetzt und menschenähnliche Ergebnisse in Bezug auf die Qualität der generierten Daten erzielt. Die Erzeugung von Sprache macht im Bereich der Audiosynthese den größten Anteil aus, wohingegen die Generierung von abstrakteren Geräuschen, wie dem Klang von Instrumenten, Umgebungsgeräuschen oder Tierlauten vergleichsweise wenig erforscht ist.

An dieser Stelle soll die vorliegende Arbeit anknüpfen und am Beispiel von Vogelgesang die Erzeugung von komplexen Geräuschen, die in der Natur vorkommen, mithilfe von Neuronalen Netzen untersuchen.

1.1 Zielsetzung

Das primäre Ziel der vorliegenden Arbeit ist die Konzeption und Implementierung einer Architektur auf Basis von Neuronalen Netzen, mit der Vogelgesang synthetisiert werden kann. Dabei sollen die Erkenntnisse aus dem Bereich der Sprachsynthese berücksichtigt werden und in die Architektur einfließen.

Die in der Arbeit entwickelte Architektur soll anschließend trainiert und die erzielten Ergebnisse quantitativ und qualitativ evaluiert werden. Im Rahmen der quantitativen Evaluation soll geprüft werden, inwieweit sich der *FID-Score* für die Bewertung von synthetisiertem Vogelgesang eignet. Aus der qualitativen Analyse, die durch den Autor

der Arbeit erfolgt, sollen exemplarische Audiobeispiele entstehen, die von der Leserin bzw. dem Leser angehört werden können.

Die Implementierung der Architektur und des Trainingsprozesses sollen für eine mögliche weiterführende Forschung öffentlich in einem *Repository* und in der beigefügten CD zugänglich gemacht werden.

1.2 Aufbau der Arbeit

Für das Erreichen der Zielsetzung ist die Arbeit in sechs weitere Abschnitte unterteilt. Im folgenden Kapitel 2 werden die Grundlagen gängiger Methoden zur Generierung von Daten im Bereich der Neuronalen Netze erörtert. Darüberhinaus werden sowohl Repräsentationsmöglichkeiten von Audiosignalen als auch Evaluationsmetriken für die vorliegende Problemstellung beschrieben und erarbeitet. Kapitel 3 widmet sich den wichtigsten verwandten Arbeiten aus dem Bereich der Audiosynthese.

Aus den Erkenntnissen der Grundlagen und den verwandten Arbeiten werden in Kapitel 4 Forschungsfragen konkretisiert und daraus Anforderungen abgeleitet, die sich auf die zu entwickelnde Architektur, ihre Implementierung und das Training beziehen. Anschließend wird in Kapitel 5 die konzipierte Architektur für die Synthese von Vogelgesang sowie die Implementierung ihrer wichtigen Komponenten vorgestellt.

In Kapitel 6 werden die aufgestellten Forschungsfragen anhand von Versuchen beantwortet und eine qualitative Analyse der erzielten Ergebnisse durchgeführt.

Aufbauend auf den Ergebnissen dieser Arbeit wird abschließend das Fazit gezogen und ein Ausblick über mögliche weiterführende Forschungsgebiete gegeben.

Im Laufe der Arbeit wird auf Daten im *Repository* verwiesen, die in der gleichen Struktur wie im *Repository* auf der beigefügten CD auffindbar sind.

2 Generierung neuer Daten mithilfe Neuronaler Netze

Abhängig vom Anwendungsfall gibt es mit Neuronalen Netzen unterschiedliche Herangehensweisen in Bezug auf das Generieren von neuartigen Datensätzen. Im folgenden Abschnitt werden kurz die Methoden vorgestellt, die sich in der Audiosynthese etabliert haben. Darauf aufbauend wird das *latent space* beschrieben, was in der vorliegenden Arbeit das Fundament zum Generieren neuer Daten darstellt. Anschließend werden *Generative Adversarial Networks (GANs)* und die darauf aufbauenden *Wasserstein - Generative Adversarial Networks (WGANs)* erläutert, die in der Versuchsdurchführung eingesetzt werden. Darauf folgt die Beschreibung der Repräsentationsmöglichkeiten von Audiosignalen, die für den weiteren Verlauf der Arbeit relevant sind. Abschließend werden Evaluationsverfahren zur Bewertung der Ergebnisse beschrieben.

2.1 Ansätze zum Generieren von Audiosignalen

Im Kontext der Neuronalen Netze haben sich verschiedene Ansätze zum Generieren neuer Daten etabliert, wovon besonders die folgenden zwei im Bereich der Audiosynthese Verwendung finden:

- *Autoregressive Modelle*, in denen die Daten in Zeitreihen eingeordnet und Vorgänger genutzt werden, um Nachfolger zu generieren.
- *Modelle* wie *Generative Adversarial Network (GANs)* oder *Variational Autoencoder (VAE)*, die in einem niedrig dimensionierten Raum Abbildungen (einem *latent space*) auf reelle Datensätze darstellen.

Autoregressive Modelle generieren neue Daten in Abhängigkeit von zuvor bekannten Daten. Die neu generierten Daten werden im darauf folgenden Schritt verwendet, um

die nächste Datenreihe zu erzeugen. Dieser Vorgang wird so lange wiederholt, bis die gewünschte Datenmenge vorhanden ist. Eine Übersicht gängiger *Autoregressiver Modelle* wird von Lim und Zohren (2020) aufgestellt.

Für den Einsatz *autoregressiver Modelle* zur Erzeugung von Audiosignalen hat die Veröffentlichung von van den Oord et al. (2016) einen wichtigen Meilenstein gelegt und gleichzeitig bewiesen, dass anhand roher Audiosignale erfolgreich gesprochene Sprache in einer hohen Qualität synthetisiert werden kann. An der Veröffentlichung von van den Oord et al. (2016) wird indessen ein wichtiges Manko *autoregressiver Modelle* deutlich: die Generierung neuer Datensätze ist sehr rechenintensiv. Das liegt unter anderem daran, dass die ausgegebenen Audiosignale einzeln wieder in das *Modell* zurückgeführt werden müssen, um neue Daten zu generieren (Donahue et al., 2018, S. 1).

Modelle auf Basis des *latent space* finden in einem niedrig dimensionierten Raum eine Repräsentation der zu generierenden Daten. Das bedeutet, dass jeder Punkt in diesem *latent space* zu einem realistischen Datensatz zuordenbar ist. Prominente Konzepte in diesem Bereich sind *VAE* (vgl. Kingma und Welling (2014) und Rezende et al. (2014)) und *GANs* (Goodfellow et al., 2014).

Diese Ansätze haben den Vorteil, dass die zu generierenden Daten nicht aufeinander aufbauen und ihre Berechnung ressourcenschonender ist. Das ist der ausschlaggebende Grund, weshalb in der vorliegenden Arbeit auf eine Methode zurückgegriffen wird, die diesem Konzept entspricht. In einer Voruntersuchung, die im Abschnitt A.1 vorgestellt wird, haben *Variational Autoencoder* schlechtere Ergebnisse erzielt als *GANs*, weshalb im weiteren Verlauf *GANs* näher beschrieben werden.

Der folgende Abschnitt erklärt die Rolle des *latent spaces* und beschreibt wie mit ihrer Hilfe neue Daten generiert werden können. Anschließend wird auf die zugrundelegenden Konzepte von *GANs* eingegangen, auf ihre Probleme hingewiesen und eine verbesserte Version vorgestellt, die in dieser Arbeit verwendet wird - die *Wasserstein-GANs*.

2.2 Der latent space

Der *latent space* dient dazu, einer Maschine Zusammenhänge zu einem Themengebiet in Form eines räumlichen *Modells* zugänglich zu machen. Durch das Training lernt das

Modell Punkte im *latent space* zu konkreten Datensätzen[1] zuzuordnen. Der *latent space* selbst hat ohne das zugehörige *Modell* keine Bedeutung.

Die Abbildung 2.1 zeigt vereinfacht einen zweidimensionalen *latent space* und die Zuordnung eines konkreten Punktes zu einem Bild.

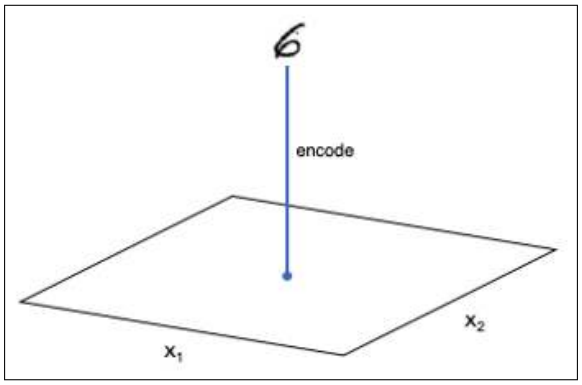


Abbildung 2.1: Beispielhafte Darstellung des *latent spaces* (Foster, 2019, S.78).

Hier wird der konkrete Punkt (x_1, x_2) zu dem Bild einer sechs zugeordnet.

Der *latent space* eines gut trainierten *Modells* hat die Eigenschaft, dass sich Daten umliegender Punkte stark ähneln. Bei einem durch *VAEs* trainierten *latent space* ist diese Eigenschaft so stark ausgeprägt, dass sich Gruppen der einzelnen Klassen bilden, wie es in Abbildung 2.2 dargestellt ist. Der *latent space* von *GANs* weist keine Gruppenbildung in dieser Form auf und muss bei Bedarf entsprechend trainiert werden (Mukherjee et al., 2018).

Die Generierung neuer Daten mithilfe eines *Modells*, welches die Trainingsdaten auf ein *latent space* abbildet, kann über verschiedene Wege erfolgen. Unter anderem gibt es folgende Möglichkeiten:

- Die einfachste Methode ist es, einen beliebigen Punkt $\vec{x_1}$ in der Nähe eines bekannten Punktes $\vec{x}$ im *latent space* auszuwählen. Aufgrund der Eigenschaft, dass ähnliche Daten im *latent space* nah beieinander liegen, ist die Wahrscheinlichkeit hoch, dass das Datenelement, das $\vec{x_1}$ zugeordnet ist, ähnlich aber ungleich dem von $\vec{x}$ ist.

- Eine weitere Möglichkeit ist das sogenannte *Morphing*. Hier werden zwei Punkte auf dem *latent space* gewählt, deren Eigenschaften miteinander kombiniert werden. Soll

[1] Als Datensatz können Bilder, Wörter oder andere Daten verstanden werden.

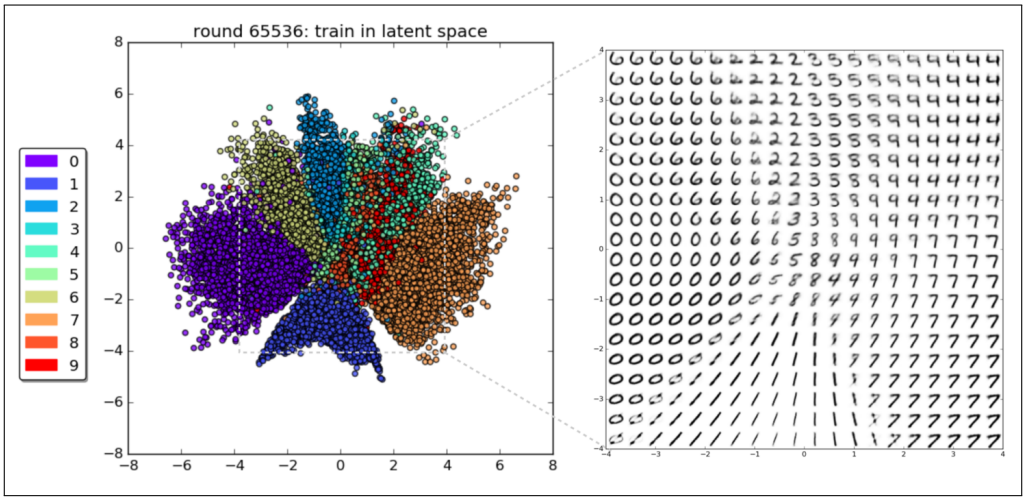


Abbildung 2.2: Beispielhafte Verteilung von Datensätzen im durch ein *VAE* trainierten *latent space* (Miriam, 2016).

Links: Die Darstellung einer Verteilung von Punkten, die den Bildern der Zahlen null (violette Punkte) bis neun (rote Punkte) zugeordnet sind. Rechts: Die entsprechenden Bilder der Zahlen. Anhand der Abbildung werden klare Gruppierungen der Zahlen innerhalb des *latent spaces* sichtbar. Am Beispiel der Zahlen vier (hell türkis) und neun (rot) wird deutlich, dass Abbildungen, die ähnliche Eigenschaften aufweisen, in der Nähe zueinander liegen. Selbst für einen Menschen ist es schwer zu erkennen, ob es sich um eine vier oder eine neun handelt.

bspw. eine neun ($\vec{x}_9$) mit einer vier ($\vec{x}_4$) kombinieren werden, kann $\vec{x}_9$ in Richtung $\vec{x}_4$ mit der folgenden Gleichung verschoben werden: $\vec{x}_{9neu} = \vec{x}_9 \cdot (1-\alpha) + \vec{x}_4 \cdot \alpha$ mit $\alpha \in [0,1]$ vgl. (Foster, 2019, S. 94f).

- Für eine gezielte Generierung neuer Datensätze ist es möglich, *arithmetische Operationen* auf den Vektoren des *latent spaces* durchzuführen. Auf diese Art können neue Daten generiert werden, die konkrete Eigenschaften aufweisen. Exemplarisch wird ein auf Gesichter trainiertes *Modell* wie von Shen et al. (2020) aufgeführt. Soll ein generiertes Gesicht $\vec{x}$ bspw. um eine Brille erweitert werden, kann der Vektor $\vec{x}$ mit dem Eigenschafts-Vektor einer Brille $\vec{f}_{brille}$[2] addiert werden. Die Abbildung 2.3 zeigt Ergebnisse *arithmetischer Operationen.*

  Für einen detaillierten Überblick über *arithmetische Operationen* im *latent space* von *GANs* wird auf die Ergebnisse von Shen et al. (2020) verwiesen.

[2]Der Eigenschafts-Vektor wird ermittelt, indem der Durchschnitt aller Punkte im *latent space* berechnet wird, die die gewünschte Eigenschaft aufweisen und dieser Vektor mit dem Durchschnittsvektor der Punkte ohne der Eigenschaft subtrahiert wird (Foster, 2019, S.93).

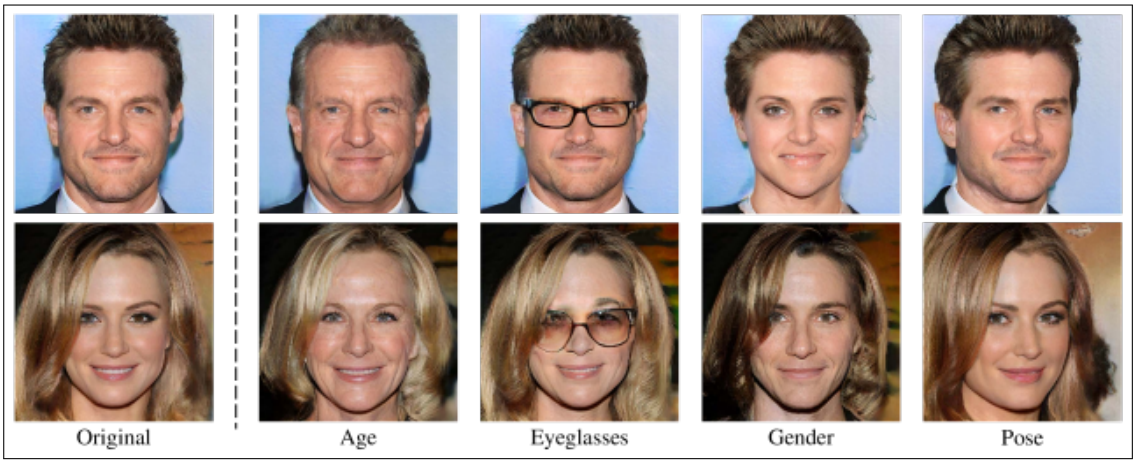


Abbildung 2.3: Ergebnisse *arithmetischer Operationen* im *latent space* (Shen et al., 2020).

Manipulation von Eigenschaften durch das Variieren der Vektoren aus dem *latent space*. Die erste Spalte zeigt die ursprüngliche Synthese, während jede der anderen Spalten die Ergebnisse der Manipulation eines bestimmten Attributs zeigt.

Der nachfolgende Abschnitt beschreibt die Funktionsweise von *Generative Adversarial Networks*, mit deren Hilfe ein *latent space* erzeugt werden soll, um Vogelgesang zu synthetisieren. Dazu wird zunächst auf den grundlegenden Aufbau von *GANs* und anschließend auf ihre Probleme eingegangen, die beim Training entstehen können. Darauf aufbauend wird eine Weiterentwicklung, die *WGANs*, erläutert, die einen Großteil dieser Probleme behebt. Die *WGANs* bilden das Fundament der in der vorliegenden Arbeit entwickelten Architektur.

2.3 Grundlagen von Generative Adversarial Networks (GANs)

Generative Adversarial Networks (*GANs*) sind eine Art der Modellierung im Kontext von *Deep Learning*, die es ermöglicht, Zuordnungen von Punkten im *latent space* zu Daten wie Bildern zu erlernen. Nach dem Training soll es möglich sein, neue Daten zu generieren, wie es im Abschnitt 2.2 beschrieben wird.

GANs wurden 2014 von Goodfellow et al. (2014) entwickelt, die seitdem stark an Popularität gewonnen haben und sukzessiv optimiert werden (vgl. Arjovsky et al. (2017),

Brock et al. (2019)). In diesem Abschnitt wird der grundlegende Aufbau und die Funktionsweise der *GANs* beschrieben. Anschließend wird ihr mathematisches Fundament, die *Loss-Funktion*, erörtert, was für das Verständnis ihrer Weiterentwicklung - den *WGANs* grundlegend ist. Dieser Abschnitt schließt mit den Trainingsproblemen ab, die mit den *GANs* einhergehen.

2.3.1 Aufbau von GANs

Der Aufbau eines *GANs* besteht aus zwei Komponenten, die im Wettbewerb zueinander stehen:

- Der *Generator* verfolgt das Ziel, Datensätze einer bestimmten Domäne zu generieren.
- Der *Diskriminator* versucht, die vom *Generator* generierten Daten von echten Daten aus einem originalen Datensatz zu unterscheiden.

Im Laufe des Trainings werden beide Komponenten optimiert, sodass der *Diskriminator* zunehmend schwerer zu täuschen ist und der *Generator* dadurch immer präzisere Imitate erzeugt. Die Abbildung 2.4 skizziert das Zusammenspiel beider Komponenten.

Grundsätzlich ist es möglich, alle Arten von Daten zu verwenden. Im Weiteren werden ausschließlich matrixförmige Datensätze (wie Bilder oder entsprechend formatierte Audiosignale) berücksichtigt, die in der vorliegenden Arbeit relevant sind.

Im Folgenden wird die Funktionsweise beider Komponenten *Generator* und *Diskriminator* näher beschrieben.

Generator Netzwerk

Der *Generator* hat die Aufgabe neue Daten zu generieren. Es erfolgt über die Zuordnung von Punkten aus dem *latent space* zu konkreten Datensätzen. Somit ist die Eingangsgröße des *Generators* ein Vektor mit üblicherweise 100- bis 200 Dimensionen. Jede Variable wird aus einer normalisierten gaußschen Verteilung mit einem Mittelwert von null und einer Standardabweichung von eins erzeugt (Brownlee, 2020b). Die Ausgangsgröße des *Generators* ist im vorliegenden Anwendungsfall matrixförmig und kann als Bild interpretiert werden, wie es in Abbildung 2.4 dargestellt ist.

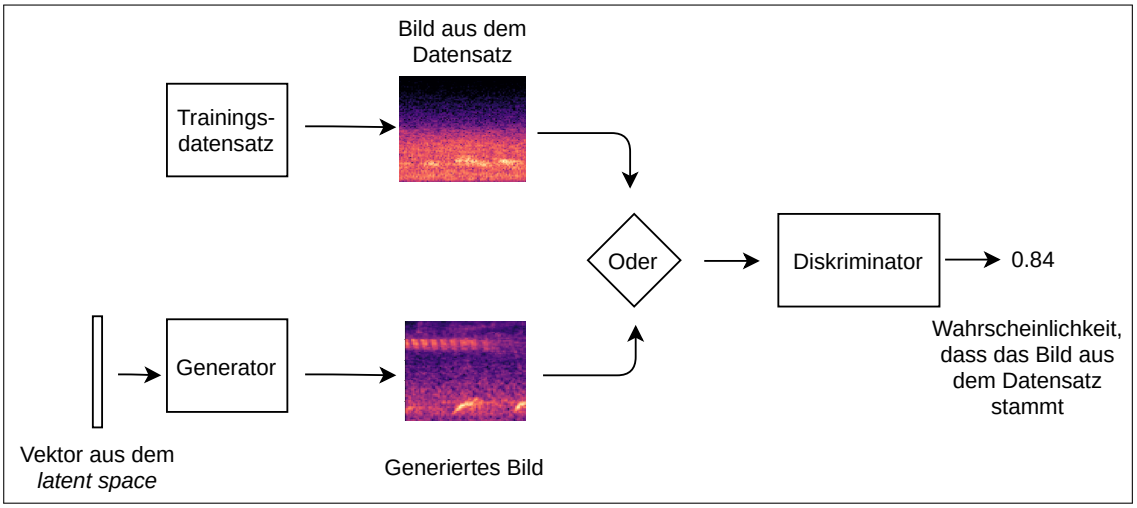


Abbildung 2.4: Zusammenspiel von *Generator* und *Diskriminator*.

Der *Generator* erzeugt über Vektoren aus dem *latent space* Daten. Der *Diskriminator* bewertet seine Eingaben dahingehend, ob es sich dabei um generierte oder originale Daten handelt und gibt einen entsprechenden Rückgabewert aus. Dieser Rückgabewert liegt in einem Intervall von $[0; 1]$ und kann als Wahrscheinlichkeit, dass das Eingangssignal vom originalen Datensatz stammt, interpretiert werden.

Dieser Abschnitt beschreibt die Architekturelemente des *Generators*, die es ermöglichen, einen Vektor in eine matrixförmige Ausgangsgröße umzuwandeln. Wie der *Generator* die Abbildung von Punkten auf Datensätze erlernt, wird in Unterabschnitt 2.3.2 beschrieben.

Das Upsampling erfolgt mit einer *Upsampling*-Schicht, die ihre Eingabewerte nach einem bestimmten Prinzip anreichert und vergrößert. Die *Upsampling*-Schicht vollführt kein *Lernen* im Sinne von Neuronalen Netzen, weshalb darauf eine *Convolutional*-Schicht[3] folgt, welche die angereicherten Daten zum Lernen nutzt.

Es haben sich folgende zwei Arten des *Upsamplings* im Kontext von *GANs* etabliert, die anhand der Erweiterung der Matrix I in Gleichung 2.1 veranschaulicht werden (Foster, 2019, S. 104f.):

- Die Wiederholung der Werte in allen Reihen und Spalten (O_1)
- Die Erweiterung der Zwischenräume jeder Zeile und Spalte mit null-Werten (O_2)

[3]Die *Convolutional*-Schicht wird im Rahmen der vorliegenden Arbeit als bekannt betrachtet und nicht näher erläutert. Für eine detaillierte Funktionsweise von *Convolutional*-Schichten wird auf die Brownlee (2020a) verwiesen.

$$I = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad O_1 = \begin{pmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{pmatrix} \quad O_2 = \begin{pmatrix} 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.1}$$

Je nach Anwendungsfall ist der Einsatz beider Methoden denkbar. Studien weisen allerdings auf, dass die zweite Methode (O_2) zu schachbrettartigen Artefakten führen kann (Odena et al., 2016).

Die hier vorgestellten Methoden des *Upsamplings* setzen eine matrixförmige Eingangsgröße voraus. Der nachfolgende Abschnitt beschreibt, wie Vektoren des *latent spaces* für das *Upsampling* genutzt werden können.

Die Umformung des Vektors aus dem *latent space* in eine Matrix kann mithilfe einer *Dense-Schicht* mit der Eingangsgröße des Vektors und einer Ausgangsgröße, die sich in eine Matrix umformen lässt, erfolgen (vgl. Implementierung (Foster, 2019, S. 104)). Diese Ausgangsgröße wird anschließend für die Weiterverwendung in eine Matrix umgeformt. Die Gleichung 2.2 skizziert diesen Vorgang.

$$\vec{i} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \vec{o} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad R = \begin{pmatrix} x_0 & x_1 \\ x_2 & x_3 \end{pmatrix} \tag{2.2}$$

Hier bildet $\vec{i}$ den Vektor aus dem *latent space*, $\vec{o}$ den Ausgangsvektor der *Dense-Schicht* mit einer quadratischen Länge und R die daraus resultierende 2×2 Matrix. Diese Matrix kann im nächsten Schritt weiter verarbeitet werden. Es folgt eine Übersicht über das Zusammenspiel aller Komponenten.

Das Zusammenspiel der vorgestellten Komponenten ermöglicht es, dem *Generator* aus einem Vektor eine Ausgangsgröße zu erzeugen, die den Maßen des gewünschten Bildes entspricht. Die Abbildung 2.5 veranschaulicht eine beispielhafte Aneinanderreihung der Komponenten. Hier implementieren das *project and reshape* u.a. die Umformung des Vektors und die dargestellten Schichten *CONV* u.a. das hier beschriebene *Upsampling*.

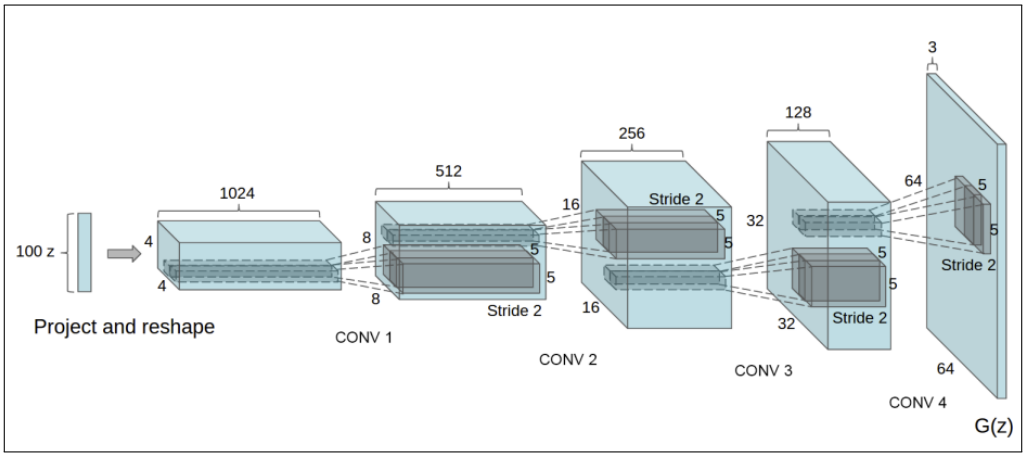


Abbildung 2.5: Skizzenhafter Aufbau eines *Generators* mit zweidimensionalen *Transposed Convolutional-Schichten* von (Radford et al., 2015)

Die Eingangsgröße ist ein Vektor mit 100 Parametern, der wie in Gleichung 2.2 beschrieben, in eine Matrix umgeformt wird. Das *Modell* besteht aus vier *Transposed Convolutional-Schichten*, welche das Signal nach dem Prinzip des *Upsamplings* sukzessiv vergrößern, bis die Ausgangsgröße von $64 \times 64 \times 3$ erreicht ist.

Die Deep Learning Frameworks wie *Keras*[4] oder *PyTorch*[5] bieten für das *Upsampling* vorgefertigte Schichten an. Im Abschnitt 5.4 wird eine konkrete Implementierung eines *Generators* gezeigt. Es folgt die Vorstellung der zweiten Komponente eines *GANs*, des *Diskriminators*.

Diskriminator Netzwerk

Der *Diskriminator* hat die Aufgabe, seine Eingangsgrößen in originale- oder generierte Daten zu klassifizieren. Die Ausgangsgröße des *Diskriminators* ist im einfachsten Fall eine Zahl zwischen null und eins - null für generierte und eins für originale Datensätze.

Bei dem *Diskriminator* handelt es sich um ein viel erforschtes Klassifizierungsmodell (vgl. Khan et al. (2019)) und wird, um den Rahmen der vorliegenden Arbeit nicht zu überschreiten, nicht weiter erläutert.

Obwohl der *Diskriminator* nach dem Training in der Regel nicht weiter verwendet wird, spielt er für den *Generator* eine entscheidende Rolle: Der *Diskriminator* ist aus Sicht

[4] https://keras.io/api/layers/convolution_layers/convolution2d_transpose/
[5] https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html

des *Generators* ein Weg, das Trainingsproblem zu einem *supervised learning*-Problem umzuformulieren. Dies wird im folgenden Abschnitt konkretisiert.

2.3.2 Der Trainingsprozess von GANs

Dieser Abschnitt beschreibt das Trainingsverfahren des *Diskriminators*, des *Generators* und ihr Zusammenspiel. Es wird an dieser Stelle noch kein konkreter Algorithmus vorgestellt, da sich das Training von *GANs* an einigen Stellen leicht vom Training der in dieser Arbeit verwendeten *WGANs* unterscheidet. Ein entsprechender Algorithmus folgt in Unterabschnitt 2.4.3. Dieser Abschnitt bildet konzeptionell die Basis für das Training von *WGANs* und wird daher kurz wiedergegeben.

Der Diskriminator wird analog zu einem *supervised learning*-Problem trainiert. Darunter wird ein Problem verstanden, in welchem zu den Eingangsgrößen bekannte Ausgangsgrößen vorhanden sind (Chollet, 2017, S. 94). Im vorliegenden Fall hat der *Diskriminator* die Aufgabe Eingangsgrößen, die entweder aus dem Trainingsdatensatz oder von dem *Generator* stammen, entsprechend zu klassifizieren.

Zu diesem Zweck wird beim Training des *Diskriminators* ein Trainingssatz erstellt, der sich aus zufällig ausgewählten originalen Daten und aus zufälligen generierten Daten vom *Generator* zusammensetzt. Der *Diskriminator* wird so trainiert, dass er eine eins für originale Daten aus dem Trainingsdatensatz und eine null für generierte Daten vom *Generator* ausgibt (Foster, 2019, S. 107f).

Für das Training des Generators existiert kein Datensatz, der die Information für eine Zuordnung von Punkten im *latent space* zu konkreten Bildern enthält. An dieser Stelle kommt der *Diskriminator* zum Einsatz, der im Trainingsprozess des *Generators* einbezogen wird. Der *Generator* wird so trainiert, dass er versucht, den *Diskriminator* zu täuschen, sodass er bei Eingabe von Daten des *Generators* eine eins statt einer null ausgibt (Foster, 2019, S. 107f).

Beim Training wird der Ausgang des *Generators* an den Eingang des *Diskriminators* geschlossen und das kombinierte *Modell* mit den erwarteten Ausgangsgrößen *eins* trainiert. Die Gewichte des *Diskriminators* werden nicht aktualisiert.

Der folgende Abschnitt beschreibt die grundlegenden Konzepte der *Loss*-Funktion, die für das Training von *GANs* entscheidend sind.

2.3.3 Die Loss-Funktion von GANs

Dieser Abschnitt widmet sich dem mathematischen Fundament der *GANs* - ihrer *Loss-Funktion*. Dabei wird das oben beschriebene Zusammenspiel von *Generator* und *Diskriminator* und ihre gegensätzliche Zielstellung aus einer anderen Perspektive beleuchtet.

Es wird zunächst die *binary cross-entropy* beschrieben, worauf aufbauend die Ziele der zwei Komponenten mathematisch ausgedrückt werden. Abschließend wird daraus die *Loss-Funktion* abgeleitet, die von Goodfellow et al. (2014) vorgestellt wird.

Dieser Abschnitt bildet die Grundlage für das Verständnis der *WGANs*, die an den Schwächen der hier vorgestellten *Loss-Funktion* anknüpft.

Die Basis der Loss-Funktion ist die sogenannte *binary cross-entropy*, welche in Gleichung 2.3 dargestellt ist (Foster, 2019, S. 116). Sie wird verwendet, um die Diskrepanz zwischen einem erwarteten und einem tatsächlichen Ausgabewert eines Neuronalen Netzes zu ermitteln, um daraus ihre Gewichte zu aktualisieren. $\hat{y}$ steht für das ausgegebene *Label* und y für das erwartete *Label*[6].

$$L(\hat{y}, y) = (y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \tag{2.3}$$

Für die Beschreibung der *Loss-Funktionen* vom *Diskriminator* und *Generator* werden folgende Annahmen getroffen.

Grundlegende Annahmen

- p_{data} ist als Datenverteilung der originalen Daten x_1 definiert.
- p_z ist als Datenverteilung des Eingangsrauschen z definiert und kann als *latent space* verstanden werden.
- p_g ist die Datenverteilung über die generierten Daten x_2.
- $D(x) \in [0, 1]$ ist eine differenzierbare Funktion, die die Wahrscheinlichkeit repräsentiert, dass x von p_{data} stammt. Die Funktion ist mit dem *Diskriminator* gleichzusetzen.

[6] Gleichzusetzen mit dem Ausgabewert des *Generators*

- $G(z)$ ist eine differenzierbare Funktion und ist eine Abbildung von z auf die generierten Daten. Die Funktion ist mit dem *Generator* gleichzusetzen.
- $\mathbb{E}_{x \sim p_{data}}$ ist der Erwartungswert über alle originalen Daten.
- $\mathbb{E}_{x \sim p_g}$ ist der Erwartungswert über alle generierten Daten.
- $\mathbb{E}_{z \sim p_z}$ ist der Erwartungswert über das Eingangsrauschen.

Die in 2.3 dargestellte Gleichung stellt den *Loss* für eine Beobachtung dar. Es soll der *Loss* für alle Beobachtungen optimiert werden, aus diesem Grund wird im weiteren Verlauf der Erwartungswert über alle Beobachtungen der Verteilungen also $\mathbb{E}_{x \sim p_{data}}$, $\mathbb{E}_{z \sim p_z}$ bzw. $\mathbb{E}_{x \sim p_g}$ eingeführt und in die Gleichungen eingesetzt.

Mit den Annahmen folgt die Erörterung der *Loss-Funktionen* des *Diskrmininators* und des *Generators*.

Der Diskriminator $D(x)$ versucht die Eingangsgrößen, die entweder aus dem originalen Datensatz p_{data} oder vom *Generator* $G(z)$ stammen, korrekt voneinander zu unterscheiden.

Das *Label* der aus p_{data} stammenden Daten x ist $y = 1$. Eingesetzt in die Gleichung 2.3 (*binary-cross-entropy*) und über alle Beobachtungen gemittelt, ergibt sich daraus Gleichung 2.4.

$$L(D(x), 1) = \mathbb{E}_{x \sim p_{data}}[\log(D(x)] \tag{2.4}$$

Für aus dem *Generator* stammenden Daten $G(z)$, ist das *Label* $y = 0$. Eingesetzt ergibt sich daraus über alle Beobachtungen gemittelt Gleichung 2.5.

$$L(D(G(z)), 0) = \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))] \tag{2.5}$$

Damit der *Diskriminator* die gewünschte Zuordnung trifft, müssen beide Gleichungen $L(D(x), 1)$ und $L(D(G(z)), 0)$ maximiert werden (Goodfellow et al., 2014, S. 2). Anhand der Abbildungen 2.6 und 2.7 wird verdeutlicht, wieso eine Maximierung der Gleichungen zu den gewünschten Ergebnissen führt.

Das Ziel des *Diskriminators* wird anhand der folgenden Funktion $V(D, G)$ zusammengefasst:

$$\max_D V(D, G) = \mathbb{E}_{x \sim p_{data}}[\log(D(x))] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))] \tag{2.6}$$

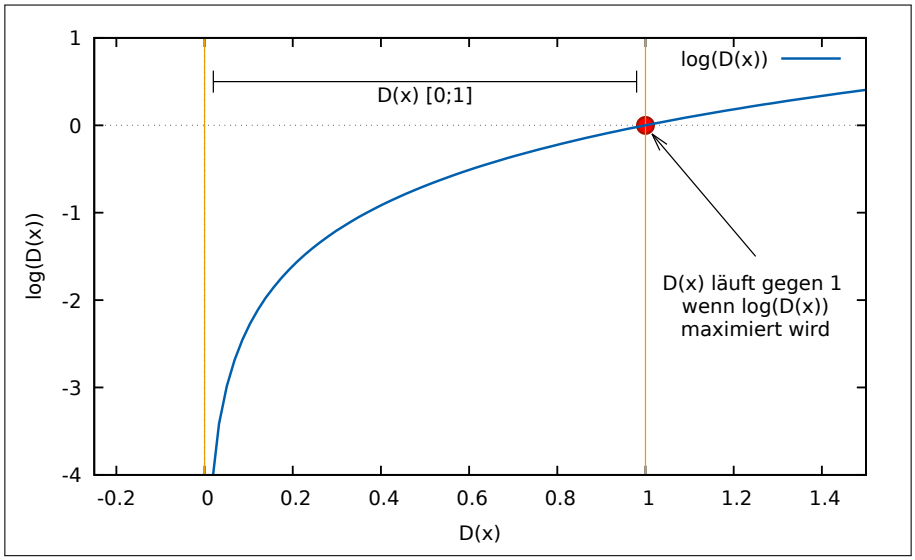


Abbildung 2.6: Darstellung der Funktion $\log(D(x))$ (y-Achse) in Abhängigkeit von $D(x)$ (x-Achse).

$\log(D(x))$ repräsentiert den *Loss* im Fall von originalen Eingangsdaten (vgl. Gleichung 2.4). Wird diese Funktion maximiert, läuft $D(x)$ gegen 1, was dem erwarteten Ausgabewert für originale Daten entspricht.

Der Generator hat das Ziel, den *Diskriminator* zu täuschen, sodass die Daten vom *Generator* als originale Daten klassifiziert werden. Das Ziel des *Generators* ist es, die Gleichung 2.5 zu minimieren. Abbildung 2.7 beschreibt, wieso das Minimieren der Funktion zu dem gewünschten Ausgabewert von eins führt.

Da der erste Summand ($\mathbb{E}_{x\sim p_{data}}[\log(D(x))]$) unabhängig vom *Generator* ist, kann das Ziel des *Generators* auch wie folgt beschrieben werden. Diese Schreibweise wird im Zusammenspiel von *Diskriminator* und *Generator* zum Einsatz kommen.

$$\min_G V(D,G) = \mathbb{E}_{x\sim p_{data}}[\log(D(x))] + \mathbb{E}_{z\sim p_z}[\log(1-D(G(z)))] \tag{2.7}$$

Das Zusammenspiel von *Diskriminator* und *Generator* wird als *minimax* Spiel mit zwei Spielern zusammengefasst (vgl. Goodfellow et al. (2014) S. 3]). Sie bildet die finale *Loss-Funktion* der *GANs*.

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}}[\log(D(x))] + \mathbb{E}_{z\sim p_z}[\log(1-D(G(z)))] \tag{2.8}$$

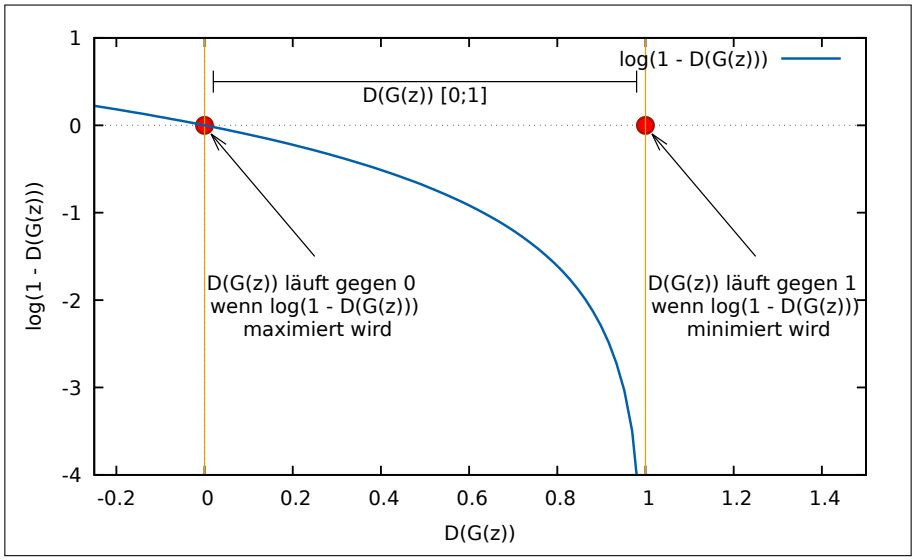


Abbildung 2.7: Darstellung der Funktion $\log(1 - D(G(z)))$ (y-Achse) in Abhängigkeit von $D(G(z)))$ (x-Achse).

$\log(1 - D(G(z)))$ repräsentiert den *Loss* im Fall von generierten Eingangsdaten (vgl. Gleichung 2.5). Wird diese Funktion maximiert, läuft $D(G(z))$ gegen 0, was aus Sicht des *Diskriminators* der erwartete Ausgabewert für generierte Daten ist.

Aus Sicht des *Generators* muss $\log(1 - D(G(z)))$ minimiert werden, damit $D(G(z))$ gegen 1 läuft und der *Diskriminator* eine falsche Klassifizierung für generierte Eingabedaten macht.

Der folgende Abschnitt beschäftigt sich mit den Problemen, die beim Training von *GANs* auftreten.

2.3.4 Herausforderungen beim Training von GANs

Dieser Abschnitt untersucht die häufigsten Herausforderungen, die beim Training von *GANs* entstehen können.

- Das Training von *GANs* ist aufgrund des *Vanishing Gradients* fragil. Wenn der *Diskriminator* perfekt ist, gilt $D(x) = 1, \forall x \in p_{data}$ und $D(x) = 0, \forall x \in p_g$. Daraus folgt, dass die *Loss-Funtion* $L(D, G)$ bei den Trainingsiterationen auf null fällt und es findet kein Lernen statt (Weng, 2019, S. 5f). Auf diese Weise entsteht beim Trainieren von *GANs* ein Dilemma: Einerseits darf der *Diskriminator* nicht zu schlecht sein, sonst hat der *Generator* ein leichtes Spiel und hat keine Motivation

gute Datensätze zu erlernen. Andererseits darf der *Diskriminator* nicht zu gut sein, sonst ist es dem *Generator* nicht möglich etwas zu lernen.

- Im Laufe des Trainings kann der *Generator* einen Zustand erreichen, in dem er den *Diskriminator* mit nur einer kleinen Menge von Ausgabewerten täuscht. Dadurch misslingt es dem *Generator* eine gute Repräsentation der Datenverteilung p_{data} zu finden, sodass die generierten Daten keine Ähnlichkeit aufweisen. Dieser Zustand wird auch als *mode collapse* bezeichnet (Foster, 2019, S. 113).

- Der während des Trainings berechnete *Loss* stellt nur das momentane Verhältnis der Qualität des *Generators* im Bezug zum *Diskriminator* dar. Dadurch kann es kommen, dass der *Loss* im Verlauf größer wird, obwohl der *Generator* immer bessere Datensätze generiert. Das passiert, wenn der *Diskriminator* schneller lernt als der *Generator*. Daraus folgt, dass der Verlauf des *Losses* kein guter Indikator für den Trainingserfolg ist (Foster, 2019, S. 114).

Weitere Trainingsschwierigkeiten können aus der Veröffentlichung von Weng (2019) entnommen werden. Im folgenden Abschnitt wird eine überarbeitete *GAN*-Version vorgestellt, die die hier genannten Probleme behandelt. Diese *WGANs* sind ein essenzieller Bestandteil der vorliegenden Arbeit.

2.4 Wasserstein-GANs

Dieser Abschnitt widmet sich einer Weiterentwicklung der oben beschriebenen *GANs*, die von Arjovsky et al. (2017) vorgestellt und als *Wasserstein-GANs (WGANs)* bezeichnet werden. Die wichtigsten Optimierungen, die die *WGANs* aufweisen, sind die Folgenden (Arjovsky et al., 2017, S. 9 und S. 16):

- Das Training wird stabilisiert.

- Es wird eine verbesserte *Loss-Metrik* eingeführt, die aussagekräftigere Ergebnisse liefert.

- Die Gefahr vom Auftreten des *Mode Collapse* wird reduziert.

Die vollständige mathematische Analyse aller Einzelheiten liegt nicht im Rahmen der vorliegenden Arbeit. Es werden hier stattdessen sukzessiv die essenziellen Überlegungen

erörtert, die zu den optimierten *WGANs* führen. Indessen werden die folgenden Punkte beschrieben:

1. Als erstes werden die Probleme der alten *Loss-Funktion* beschrieben, die die Autoren Arjovsky et al. (2017) zu der Weiterentwicklung der *GANs* motiviert haben.
2. Darauf aufbauend wird die neue *Loss-Funktion* mit ihren einhergehenden Vorteilen vorgestellt.
3. Abschließend werden die weiteren Anpassungen, die das Netz und den Trainingsablauf betreffen, beschrieben.

2.4.1 Probleme der alten Loss-Funktion

Den notwendigen Optimierungsbedarf sehen Arjovsky et al. (2017) in der oben beschrieben *Loss-Funktion* der *GANs*. Die Minimierung der Gleichung 2.7 unter Einsatz des optimalen *Diskriminators* D^* ist äquivalent zur Minimierung der *Jensen-Shanon-Divergenz* (im Folgenden $D_{JS}(p||q)$). Die in Gleichung 2.8 aufgeführte *Loss-Funktion* $L(D, G)$ kann unter Einsatz eines optimalen *Diskriminators* D^* wie folgt beschrieben werden (Goodfellow et al., 2014, S. 5). Für eine detailliertere mathematische Ausführung wird auf die Veröffentlichung (Weng, 2019, S. 4f) verwiesen.

$$L(D^*, G) = 2D_{JS}(p_{data}||p_g) - 2\log 2 \tag{2.9}$$

Die *Jensen-Shanon-Divergenz* ist eine Methode zur Messung der Ähnlichkeit zwischen zwei Wahrscheinlichkeitsverteilungen. Ihre mathematische Beschreibung bleibt an dieser Stelle aus. Es wird bei Bedarf auf Arjovsky et al. (2017) verwiesen. Stattdessen wird anhand des nachfolgenden Beispiels die entscheidende Eigenschaft der *Jensen-Shanon-Divergenz* beschrieben, die das Optimieren der Gleichung 2.9 laut Arjovsky et al. (2017) erschwert.

Es seien zwei Wahrscheinlichkeitsverteilungen p und q gegeben, deren Ähnlichkeit anhand der *Jensen-Shanon-Divergenz* gemessen werden soll. Die *Jensen-Shanon-Divergenz* ist konvergent und ergibt $\log(2)$, wenn p und q keine signifikanten Überschneidungen haben. Das daraus resultierende Problem soll anhand des folgenden Beispiels verdeutlicht werden.

Angenommen, es sollen gaußverteilte Wahrscheinlichkeitsverteilungen verglichen werden, die sich nur im Mittelwert voneinander unterscheiden (vgl. Abbildung 2.8). Bei einem Vergleich einer beliebigen Verteilung q_1, q_2, q_3 mit p, ergibt die *Jensen-Shanon-Divergenz* immer log(2). Dieses Ergebnis lässt keine Interpretation zu, welche der Verteilungen am nächsten zu p ist. Bezogen auf das Training des *Generators*, liefert die *Jensen-Shanon-Divergenz* keinen guten Gradienten für disjunkte Datenverteilungen und das Training ist sehr langsam (Arjovsky et al., 2017, S. 8f).

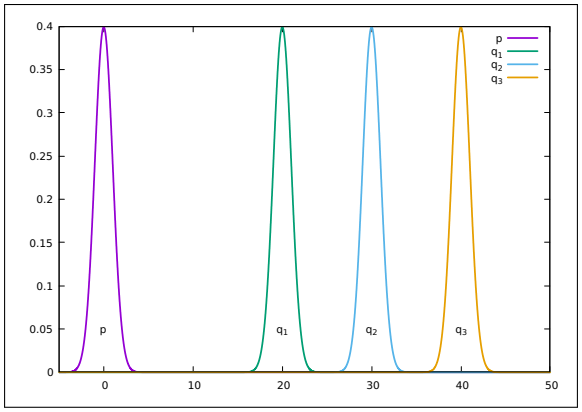


Abbildung 2.8: Gaußverteilte Wahrscheinlichkeitsverteilungen, die sich im Mittelwert voneinander unterscheiden.

Die *Jensen-Shanon-Divergenz* zwei beliebiger Wahrscheinlichkeitsverteilungen ergibt immer log(2).

Das Ziel der Autoren Arjovsky et al. (2017) ist es, eine Metrik einzusetzen, die an jeder Stelle einen für das Training nutzbaren Gradienten liefert und die auch für Datenverteilungen, die keine signifikante Überschneidungen haben, miteinander vergleichbare Werte erzeugt.

Um dieses Ziel zu erreichen, schlagen die Autoren die *Wasserstein-Distanz* vor (auch als *Earth-Mover's-Distanz* bekannt). Der folgende Abschnitt beschreibt die wesentlichen Eigenschaften und formuliert die daraus entstehende *Loss-Funktion* des *WGANs*.

2.4.2 Die Wasserstein-Distanz als Loss-Funktion

Die neue *Loss-Funktion*, die das Auftreten der *Jensen-Shanon-Divergenz* umgehen soll, ergibt sich aus der sogenannten *Wasserstein-Distanz*. Sie gibt die minimalen Kosten an, um eine Verteilung q in eine andere Verteilung p zu transformieren (Arjovsky et al.,

2017, S. 4). Ein detailreiches Beispiel zur Berechnung dieser Bewegungskosten kann aus der Veröffentlichung (Weng, 2019, S. 8f.) entnommen werden.

Die Autoren Arjovsky et al. (2017) stellen zur Berechnung der *Wasserstein-Distanz* die Gleichung 2.10 nach der *Kantorovich-Rubinstein Dualität* auf, in der das *Supremum* über alle $1-Lipschitz$[7] Funktionen $f : X \to \mathbb{R}$ die *Wasserstein-Divergenz* ergibt.

$$W(p_{data}, p_g) = \sup_{||f||_{L\leq 1}} \mathbb{E}_{x\sim p_{data}}[f(x)] - \mathbb{E}_{x\sim p_g}[f(x)] \tag{2.10}$$

Die genaue Erörterung dieser Funktion würde den Rahmen dieser Arbeit überschreiten. Wichtig bleibt festzuhalten, dass die Funktion f, die das Supremums- bzw. Maximierungsproblem löst, mithilfe des *Diskriminators* approximiert wird. Der *Diskriminator* berechnet somit die *Wasserstein-Distanz* zwischen p_{data} und p_g.

Um die *1-Lipschitz* Bedingung zu erfüllen, gibt es mehrere Möglichkeiten. Die ursprüngliche Veröffentlichung schlägt als ersten Ansatz den Einsatz von *weight clipping* vor. Mittlerweile hat sich das sogenannte *gradient penalty* von Gulrajani et al. (2017) als effektiver herausgestellt, was auch in der vorliegenden Arbeit zur Verwendung kommt.

Unter dem *gradient penalty* wird ein Zusatz in der *Loss-Funktion* verstanden, der das *Modell* bestraft, sofern der Gradient den Wert 1 überschreitet. Als *gradient penalty* schlagen die Autoren Gulrajani et al. (2017) Gleichung 2.11 vor:

$$\lambda \mathbb{E}_{\hat{x}\sim p_{\hat{x}}}[(||\nabla_{\hat{x}} D(\hat{x})||_2 - 1)^2] \tag{2.11}$$

Dabei gilt $\hat{x} = \epsilon\tilde{x} + (1-\epsilon)x$ mit $0 \leq \epsilon \leq 1$. $\tilde{x}$ steht für ein originales Bild und x für ein generiertes Bild.

Für den Diskriminator ergibt sich zusammengesetzt aus Gleichung 2.10 und Gleichung 2.11 die neue *Loss-Funktion* (Gulrajani et al., 2017, S. 3)[8]:

$$\max_D L(D,G) = \mathbb{E}_{x\sim p_{data}}[D(x)] - \mathbb{E}_{z\sim p_z}[D(G(z)))] - \lambda \mathbb{E}_{\hat{x}\sim p_{\hat{x}}}[(||\nabla_{\hat{x}} D(\hat{x})||_2 - 1)^2] \tag{2.12}$$

[7] Die *Lipschitzstetigkeit* einer Funktion gibt an, dass diese Funktion zu keinem Punkt eine bestimmte Steigung beträgt. Eine 1-Lipschitz Funktion hat zu keinem Zeitpunkt eine Steigung, die größer als eins ist.

[8] Die Formel aus der Veröffentlichung ist mit einem anderen Vorzeichen, weil die Autoren vom Minimieren der *Loss-Funktion* ausgehen. Der Grund dafür wird im folgenden Abschnitt näher erläutert.

Mit der neuen *Loss-Funktion* lernt der *Diskriminator* die *Wasserstein-Distanz* zu approximieren. Dadurch ändert sich seine Rolle vom *Diskriminator* im weiteren Verlauf zu einem *Kritiker*, der die *Wasserstein-Distanz* zwischen den beiden Wahrscheinlichkeitsverteilungen p_{data} und p_g berechnet.

Anders als beim oben beschriebenen *GAN* ist für das Training ein optimaler *Kritiker* früh erwünscht, weil dadurch eine zuverlässige *Wasserstein-Distanz* gewährleistet wird. Mit dieser neuen Rolle des *Kritikers* gewinnt der *Loss* des *Generators* an Aussagekraft, da seine Ausgabewerte stets an der *Wasserstein-Distanz* gemessen werden.

Die Änderung an der Architektur, die mit den *WGANs* einhergeht, betrifft lediglich die letzte Schicht des *Kritikers*. Hier wird die *Sigmoid* Funktion entfernt, damit die *Wasserstein-Distanz* kontinuierlich verläuft und nicht zwischen 0 und 1 begrenzt bleibt (Foster, 2019, S. 120).

Der Generator versucht die *Wasserstein-Distanz* zwischen den beiden Wahrscheinlichkeitsverteilungen p_{data} und p_g zu minimieren. Dabei spielt der *gradient penalty*-Term keine Rolle. Daraus ergibt sich die *Loss-Funktion* 2.13 für den *Generator* (Arjovsky et al., 2017, S. 6ff):

$$\min_G L(D, G) = \mathbb{E}_{x \sim p_{data}}[D(x)] - \mathbb{E}_{z \sim p_z}[D(G(z))] \tag{2.13}$$

Da der erste Summand unabhängig vom Generator ist, kann er weg gelassen werden. Daraus ergibt sich die finale Gleichung 2.14:

$$\min_G L(D, G) = -\mathbb{E}_{z \sim p_z}[D(G(z))] \tag{2.14}$$

Der nachfolgende Abschnitt beschreibt den neuen Trainingsablauf, in dem das Zusammenspiel der hier vorgestellten *Loss-Funktionen* verdeutlicht wird.

2.4.3 Der neue Trainingsablauf

Das Training der *WGANs* ähnelt dem oben beschriebenen Training der *GANs*. Der *Kritiker* und der *Generator* werden separat trainiert. Dadurch, dass die *Wasserstein-Distanz* kontinuierlich und (fast) immer differenzierbar ist, wird empfohlen, ihn im Vorfeld bis

zum Optimum zu trainieren. Das ist möglich, weil der oben beschriebene *vanishing gradient* bei der hier verwendeten Metrik nicht vorhanden ist (Arjovsky et al., 2017, S. 8). In der Praxis wird der *Kritiker* für jede Iteration des *Generators* n-Mal trainiert.

Der Trainingsprozess beider Komponenten wird im folgenden Pseudocode dargestellt (Gulrajani et al., 2017, S.4). Hier ist das Augenmerk auf die *Loss-Funktion* des *Kritikers* in Zeile 7 zu legen. Die verwendete Funktion unterscheidet sich von der oben vorgestellten Funktion des *Kritikers*. Das liegt daran, dass die *Optimierungsfunktion Adam* intern eine *Minimierung* durchführt. Als Abhilfe wird hier die zu maximierende Funktion negiert. Die *Optimierungsfunktion Adam* wird im Rahmen dieser Arbeit nicht weiter beschrieben, bei Bedarf wird auf Kingma und Ba (2017) verwiesen.

Algorithm 1: WGAN Training mit *gradient penalty*. Folgende Werte sind empfohlen $\lambda = 10, n_{critic} = 5, \alpha = 0.0001, \beta_1 = 0, \beta_2 = 0.9$.

Data: λ: *Gradient penalty* Koeffizient, n_{critic}: *Diskriminator*-Iterationen pro *Generator*-Iteration, m:*batch*-Größe, α, β_1, β_2: *Adam*-Parameter (sie werden zur Übersichtlichkeit im Pseudocode ausgelassen)

Data: w_0: initiale *Diskriminator*-Parameter, θ_0: initiale *Generator*-Parameter

1 **while** θ *sich nicht an 0 genähert hat* **do**
2 **for** $t = 1, ..., n_{critic}$ **do**
3 **for** $i = 1, ..., m$ **do**
4 Sample echte Daten $x \sim p_{data}$, die latent variable $z \sim p_z$, eine zufällige Zahl $\epsilon \sim U[0, 1]$
5 $\tilde{x} \leftarrow G_\theta(z)$
6 $\hat{x} \leftarrow \epsilon x + (1 - \epsilon)\tilde{x}$
7 $L^{(i)} \leftarrow D_w(\tilde{x}) - D_w(x) + \lambda(||\nabla_{\hat{x}} D_w(\hat{x})||_2 - 1)^2$
8 **end for**
9 $w \leftarrow Adam(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, w)$
10 **end for**
11 sample ein Batch von latent Variablen $\{z^{(i)}\}_{i=1}^{m} \sim p_z$
12 $\theta \leftarrow Adam(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(z), \theta)$
13 **end while**

Eine konkrete Implementierung der *WGANs* ist in Abschnitt 5.4 vorgestellt.

Die hier beschriebene Architektur ist dafür ausgelegt, zweidimensionale Daten zu generieren. Der folgende Abschnitt beschäftigt sich mit zweidimensionalen Darstellungsformen von Audiosignalen.

2.5 Modellierung von Audiosignalen

In dem vorangehenden Abschnitt wird beschrieben, wie mithilfe Neuronaler Netze neue Daten generiert werden. Für die Implementierung werden *WGANs* beschrieben und exemplarisch dargestellt, wie mit ihrer Hilfe Bilder erzeugt werden können.

Das Ziel der vorliegenden Arbeit ist es, Audiosignale zu synthetisieren. Dieser Abschnitt stellt den Bezug von Audiosignalen zu einer zweidimensionalen Repräsentation wie Bildern her. Es werden die zwei folgenden Methoden vorgestellt: *Spektrogramme* und ihre verwandten *Mel-Spektrogramme*.

2.5.1 Spektrogramme zur Darstellung von Audiosignalen

Spektrogramme sind eine Möglichkeit Signale in ihrer Frequenzdomäne darzustellen. Die folgende Abbildung 2.9 zeigt die Darstellung eines Audiosignals in *Wellenform* und daneben als *Spektrogramm*.

Die linke Darstellung zeigt den zeitlichen Verlauf der Amplitude. Die mittlere Darstellung zeigt das selbe Signal als *Spektrogramm*. Hier bildet die y-Achse den Frequenzbereich des Signals ab. Es kommt zusätzlich eine weitere Ebene hinzu, die die Lautstärke der jeweiligen Frequenz angibt und in der Färbung dargestellt ist. Je heller der Bereich, desto höher ist die Amplitude der Frequenz.

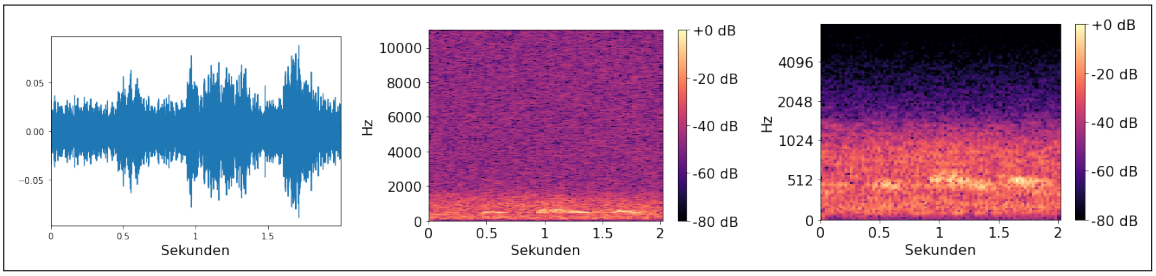


Abbildung 2.9: Darstellung des Gesangs einer Ringeltaube in *Wellenform* (links), als *Spektrogramm* (mitte) und als *Mel-Spektrogramm* (rechts)

Anhand der Abbildung wird visualisiert, dass *Spektrogramme* das Potenzial haben, Muster deutlicher hervorzuheben als die *Wellenform*.

Eine wichtige Eigenschaft der *Spektrogramme* ist, dass sie sich wieder in Audiosignale umwandeln lassen. Die Umformung von Signalen in ein *Spektrogramm* erfolgt über die *short-time-(fast)-fourier-transformation (STFT)* (Kehtarnavaz, 2008) und kann über die Inverse-STFT mithilfe des *Griffin-Lim-Algorithmus* wieder zurückgewandelt werden (Griffin und Lim, 1983).

Um eine konkrete Vorstellung zu haben, wie ein *Spektrogramm klingt* und wie sie mithilfe einer *Python-Bibliotheken* erzeugt wird, ist diesem Abschnitt eine Seite des *Jupyter-Notebook* gewidmet[9]. Die mathematische Herleitung von *Spektrogrammen* bleibt an dieser Stelle aus, da es sich um ein viel erforschtes Thema handelt. Es gibt eine große Auswahl von Bibliotheken, die die Arbeit mit *Spektrogrammen* vereinfachen[10].

Der nachfolgende Abschnitt beschreibt eine weitere Art der *Spektrogramme*, die auf das subjektive Hörempfinden der Menschen angepasst ist.

2.5.2 Mel-Spektrogramme zur Darstellung von Audiosignalen

Empirische Studien haben ergeben, dass die von dem Menschen subjektiv wahrgenommene Tonhöhe nicht proportional zum Verlauf der Frequenz ist. Das bedeutet, dass zwei durch ein Delta getrennte Frequenzpaare vom Menschen nicht immer als äquidistant wahrgenommen werden. Eine Skala, die die Proportionalität an jeder Stelle gewährleistet, ist die sogenannte *Mel-Skala*.

Der Höhenunterschied zweier beliebiger Frequenzpaare, die in der *Mel-Skala* denselben Abstand aufweisen, werden vom Menschen als gleich weit entfernt wahrgenommen (NVIDIA, 2018, Truax, 1999). Die folgende Abbildung 2.10 stellt den Zusammenhang von *Mel* und Frequenz dar.

Der Verlauf der *Mel-Skala* verdeutlicht das menschliche Wahrnehmungsverhalten von Frequenzen. Es kann zwischen niedrigen Frequenzbereichen stärker differenziert werden als zwischen hohen Frequenzbereichen.

[9] https://github.com/batonfabi/master_thesis/blob/submission/kapitel_2/spektrogramme.ipynb
[10] Vgl. https://librosa.org/ oder https://realpython.com/python-scipy-fft/

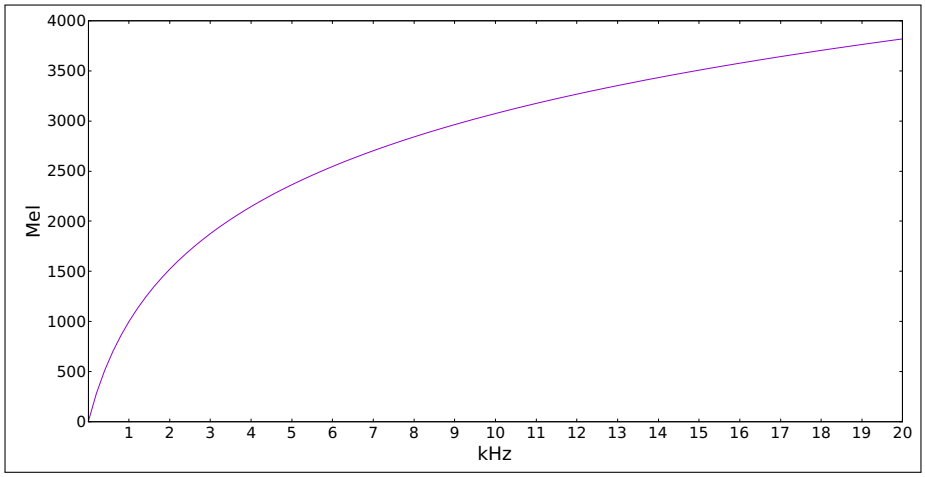


Abbildung 2.10: Darstellung der *Mel-Skala* nach der Formel $mel(f) = 2595 \log_{10}(1 + \frac{f}{700})$ approximiert (O'Shaughnessy, 2000).

Diese Eigenschaft wird in der Darstellung des Signals als *Mel-Spektrogram* in Abbildung 2.9 (rechts) deutlich. Die niedrigen Frequenzbereiche, die differenzierter wahrgenommen werden, sind in dem *Mel-Spektrogram* prominenter vertreten als im herkömmlichen *Spektrogram*.

Mel-Spektrogramme erfahren im Bereich von *Deep Learning* in unterschiedlichen Domänen wie Audioklassifizierung oder Sprachverarbeitung eine häufige Verwendung. Auch in der vorliegenden Arbeit werden *Mel-Spektogramme* aufgrund ihres Fokuses auf das menschliche Hörempfinden verwendet. Anders als die *Spektogramme* lassen sich die *Mel-Spektogramme* nur über Approximationen wieder zurück in Audiosignale umformen (Kumar et al., 2019, S. 2). In der vorliegenden Arbeit erfolgt die Umformung der *Mel-Spektogramme* über ein *Neuronales Netz*. Dieser Vorgang wird in Kapitel 5 genauer erläutert.

Der nachfolgende Abschnitt beschreibt Evaluationsmethoden für die Ergebnisse der *GANs*.

2.6 Evaluation der Ergebnisse von GANs

Die Evaluation eines *GAN-Modells* ist ein breites Forschungsgebiet, in dem verschiedene Metriken entwickelt wurden. Aufgrund der Komplexität der Fragestellung nach der Bewertung eines *GAN-Modells* besteht bisher kein Konsens darüber, welche dieser Metriken am besten die Stärken und Schwächen des zu evaluierenden *Modells* erfasst. Ein Grund

hierfür ist, dass der Fokus bei den generierten Daten auf unterschiedlichen Bereichen liegen kann.

Bei einigen *Modellen* liegt das Augenmerk beispielsweise in der Diversität der generierten Daten, bei dem nächsten in der Qualität und bei anderen in der Struktur des *latent spaces*, um gezielt Daten generieren zu können. Die gängigsten Metriken zur Evaluation von *GANs* werden von Borji (2018) analysiert. Dabei wird zwischen qualitativen und quantitativen Metriken unterschieden.

Qualitative Metriken sind solche Maße, die nicht numerisch sind. Diese evaluieren entweder über menschliche, subjektive Bewertungen oder über die Bewertung durch den direkten Vergleich, wie bei dem Heranziehen des *Nearest Neighbours* von generierten Daten mit dem Trainingsdatensatz. Quantitative Metriken vergeben spezifische nummerische Noten oder Punkte, mit denen die Qualität und Diversität der generierten Bilder bewertet wird (Borji, 2018).

Das in der vorliegenden Arbeit entwickelte *Modell* wird sowohl an der Qualität als auch in der Diversität der generierten Daten gemessen. Dafür wird jeweils eine qualitative und eine quantitative Metrik eingesetzt. Diese Auswahl orientiert sich an den Ergebnissen von Borji (2018).

Als qualitative Metrik erfolgt eine subjektive Bewertung der Ergebnisse durch den Autor. Dabei werden zufällig generierte Daten sowohl visuell als auch akustisch untersucht. Zusätzlich werden typische Signale aus dem Trainingsdatensatz mit ihren *Nearest Neighbours* aus einer Menge von generierten Daten verglichen. Die manuelle Bewertung der Ergebnisse ist ein zeitaufwändiger Prozess und kann nur begrenzt eine Aussage bzgl. der Diversität der Daten treffen. Zu diesem Zweck wird zusätzlich der durchschnittliche geringste *euklidischer Abstand* aller Elemente innerhalb einer generierten Datenmenge berechnet. Das bedeutet, dass von jedem Element aus der Menge der geringste *euklidische Abstand* zu allen anderen Elementen der selben Menge identifiziert wird. Der daraus resultierende Durchschnitt wird anschließend als Indikator für die Datendiversität genommen. Je größer der Wert, desto höher die Diversität.

Als quantitative Metrik wird die *FID* von Heusel et al. (2018) gewählt, die von Borji (2018) als plausible Metrik für verschiedene Domänen empfohlen wird. Die *FID* wird mithilfe eines auf 1000 Klassen vortrainierten *Modells*, dem *Inception V3 Net*, ermittelt.

Die Abbildung 2.11 zeigt exemplarisch den Verlauf der *FID* bei steigender Verzerrung von Bildern. Die *FID* liegt im Bereich $[0; +\infty]$ und läuft gegen 0, sofern die generier-

ten Bilder den statistischen Eigenschaften der realen Bilder entsprechen. Die konkrete Funktionsweise der *FID* kann aus Heusel et al. (2018) entnommen werden.

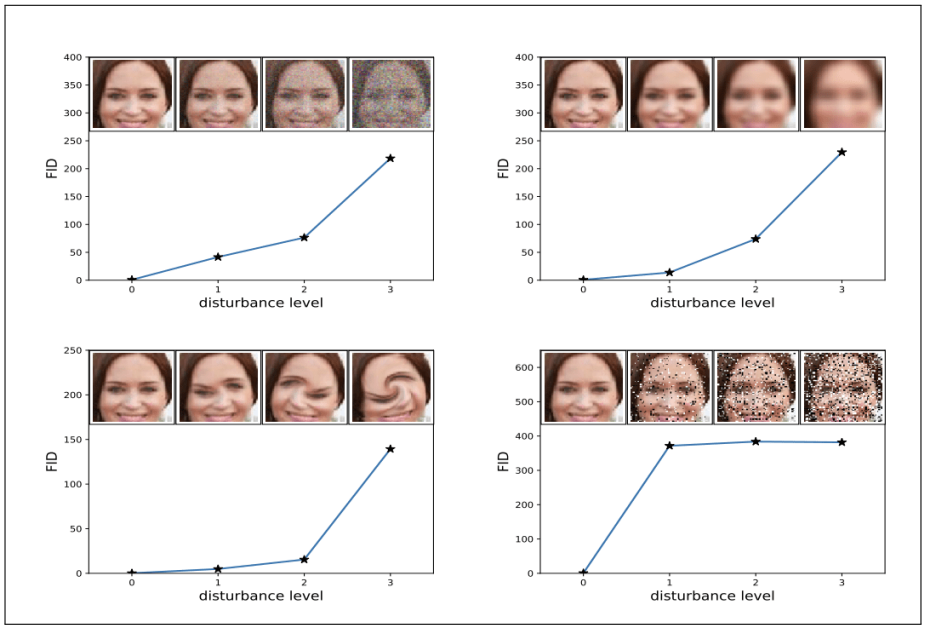


Abbildung 2.11: Beispielhafter Verlauf der *FID* bei der Verzerrung von Bildern (Heusel et al., 2018, S. 6).

.

Mit steigender Verzerrung steigt die *FID*. Im Umkehrschluss läuft die *FID* gegen 0, sofern die verglichenen Bilder übereinstimmen.

Der nachfolgende Abschnitt stellt verwandte Veröffentlichungen vor, die sich mit dem Generieren von Audiosignalen im Bereich der Neuronalen Netze beschäftigen und auf die die vorliegende Arbeit aufbaut.

3 Verwandte Arbeiten

In diesem Abschnitt werden drei Arbeiten vorgestellt, die Audiosignale mithilfe *Neuronaler Netze* synthetisieren. In der ersten Veröffentlichung Donahue et al. (2018) werden zwei unterschiedliche *GAN-Architekturen* untersucht. In van den Oord et al. (2016) wird ein *autoregressives Modell* entwickelt, welches menschenähnliche Ergebnisse bei der Sprachsynthese erzielt. Die letzte Arbeit Shen et al. (2017) nutzt Ideen des *autoregressiven Modells*, um *Mel-Spektogramme* in *Wellenform* umzuwandeln.

Die vorliegende Arbeit nutzt verschiedene Ideen der hier vorgestellten Architekturen, um mithilfe von *GANs* Vogelgesang zu synthetisieren.

3.1 Adversarial Audio Synthesis

Dieser Abschnitt beschreibt die wesentlichen Erkenntnisse von Donahue et al. (2018), in dem zwei *GAN-Architekturen* zur Synthetisierung von Audiosignalen untersucht werden. Dabei handelt es sich bei der ersten Architektur um das sogenannte *SpecGAN*, welches *Spektogramme* erzeugt, die anschließend in Audiosignale umgewandelt werden. Die zweite Architektur (*WaveGAN*) erzeugt direkt Audiosignale in *Wellenform*.

Um die Vergleichbarkeit der Ergebnisse zu gewährleisten, ist die Signallänge und Abtastrate der generierten Signale beider Architekturen gleich. Die Signallänge beträgt knapp eine Sekunde mit der Abtastrate von 16 kHz. Beide *Modelle* implementieren *Wasserstein-GANs* mit *gradient panelty*.

3.1.1 SpecGAN

Das *SpecGAN-Modell* (Donahue et al., 2018, S.4f) erzeugt *Spektogramme*, die über den *Griffin-Lim-Algorithmus* Griffin und Lim (1983) in Audiosignale mit 16384[1] Sampeln

[1] Entspricht bei eine Abtastrate von 16kHz etwas mehr als eine Sekunde Audiosignal.

umgewandelt werden. Die Architektur basiert auf dem *DCGAN* von Radford et al. (2015), welches zweidimensionale *Convolutional-Schichten* verwendet - ähnlich wie es in Abschnitt 2.3.1 beschrieben ist. Das ursprüngliche Model mit der Ausgangsgröße von $64 \times 64 \times 3$ wird um eine $128 \times 128 \times 1$ Schicht erweitert.

Für das Training werden *Spektogramme* mithilfe der *STFT* mit den folgenden Parametern erzeugt: die Fenstergröße beträgt 16ms und die Schrittgröße 8ms. Die daraus resultierenden 129×128 werden auf 128×128 gekürzt und normalisiert. Für die Umwandlung der generierten *Spektogramme* in Audiosignale in *Wellenform* wird die *Inverse STFT* genutzt.

Der konkrete Aufbau des *SpecGANs* kann aus (Donahue et al., 2018, S.15f) entnommen werden.

3.1.2 WaveGAN

Die Architektur des *WaveGAN-Modells* (Donahue et al., 2018, S.3f) basiert ebenfalls auf dem *DCGAN* Radford et al. (2015), generiert aber anders als das oben beschriebene *SpecGAN* direkt *rohe* Audiosignale in *Wellenform*. Das *DCGAN* ist für matrixförmige Daten ausgerichtet und muss für die Ausgabe von Audiosignalen entsprechend angepasst werden. Die wichtigsten Änderungen und Eigenschaften des *WaveGAN-Modells* sind die Folgenden (Donahue et al., 2018, S.4):

- Die zweidimensionalen *Convolutional-Schichten* mit der Kernelgröße 5×5 werden durch eindimensionale *Convolutional-Schichten* mit der Kernelgröße 25 ersetzt.
- Die Schrittgröße (*Stride*) der *Convolutional-Schichten* beträgt 4.
- Alle *Batch-Normalisierungsschichten* werden entfernt.
- Die *Eingangsgsschicht* des *Diskriminators* und die *Ausgangsschicht* des *Generators* betragen 16384 Parameter.

Der konkrete Aufbau des *WaveGANs* kann aus (Donahue et al., 2018, S.15) entnommen werden.

3.1.3 Ergebnisse

Die Autoren bewerten die Ergebnisse sowohl quantitativ über analytische Methoden, als auch qualitativ über menschliche Beurteilungen. Die folgenden Daten werden für die Bewertung der Ergebnisse erhoben:

- *Inception score* (Salimans et al., 2016), welches mithilfe eines trainierten *Inception-Klassifizierer* ermittelt wird. Er misst die Vielfalt und die semantische Unterscheidbarkeit der erzeugten Daten. Aufgrund der in (Donahue et al., 2018, S.6) aufgeführten Fehlerquellen ziehen die Autoren die zwei nachfolgenden Metriken hinzu.
- $|D|_{self}$ misst den durchschnittlichen *euklidischen Abstand* von einer Menge zufälliger Stichproben der Daten zu deren nächstem Nachbarn. Ein höherer Wert zeugt von einer höheren Diversität der Daten (Donahue et al., 2018, S.6).
- $|D|_{train}$ gibt den durchschnittlichen *euklidischen Abstand* von generierten Beispielen zu ihrem nächsten Nachbarn im Trainingsdatensatz. Wenn das generative *Modell* nur Beispiele aus Originaldaten produziert, wird der Wert 0 erwartet. (Donahue et al., 2018, S.7)
- Für die menschliche Beurteilung werden Werte zur Genauigkeit, Qualität, Deutlichkeit der Aussprache und Diversität der Sprecher erhoben.

Die folgende Tabelle 3.1 zeigt das beste Ergebnis des *WaveGANs*, *SpecGANs* und zum Vergleich des Trainingsdatensatzes.

Tabelle 3.1: Chapter (Donahue et al., 2018, S.7)

Datenquelle	Inception	$\|D\|_{Self}$	$\|D\|_{Train}$	Gen.	Qualität	Deutl.	Diversität
Testdatensatz	$8,01 \pm 0,24$	$1,0$	$1,0$	$0,95$	$3,9 \pm 0,8$	$3,9 \pm 1,1$	$3,5 \pm 1,0$
WaveGan	$4,67 \pm 0,01$	$0,8$	$2,3$	$0,58$	$2,3 \pm 0,9$	$2,8 \pm 0,9$	$3,2 \pm 0,9$
SpecGAN	$6,03 \pm 0,4$	$1,1$	$1,4$	$0,66$	$1,9 \pm 0,8$	$2,8 \pm 0,9$	$2,6 \pm 1,0$

Die Autoren fassen zusammen, dass beide *Modelle* neue (hohe $|D|_{self}$ Werte) und vielfältige (hohe $|D|_{Train}$ Werte) Daten produzieren. Die Qualität der generierten Audiosignale beider *Modelle* kommt nicht an die der originalen Datensätzen ran, auch die Genauigkeit der generierten Daten liegt bei nur knapp 60%. Es lässt sich an der Stelle noch nicht sagen, ob das Konzept des *SpecGANs* oder *WaveGANs* besser ist und die Autoren lassen diese Fragestellung für weitere Untersuchungen offen (Donahue et al., 2018, S. 7f).

Die vorliegende Arbeit verfolgt einen ähnlichen Ansatz zum *SpecGANs*, doch statt des *Griffin-Lim-Algorithmus* zur Konvertierung der *Spektogramme* kommt ein weiteres Neuronales Netz zum Einsatz.

3.2 WaveNet - ein generatives Model für rohe Audiosignale

Dieser Abschnitt beschreibt ein *autoregressives Modell*, welches von van den Oord et al. (2016) vorgestellt und als *WaveNet* bezeichnet wird. Das *WaveNet* synthetisiert Audiosignale in *roher Wellenform*, indem es ein *Abtastwert* pro Zeit auf Basis der Vorgänger generiert.

Die Autoren drücken die Wahrscheinlichkeit einer *Wellenform* $x = x_1, ..., x_T$ als Produkt von bedingten Wahrscheinlichkeiten aus (van den Oord et al., 2016, S.2):

$$p(x) = \prod_{t=1}^{T} p(x_t | x_1, ..., x_{t-1}) \tag{3.1}$$

Jedes *Abtastwert* x_t ist von allen Vorgängern abhängig. Für die Berechnung des nächsten *Abtastwertes* ist das *WaveNet* auf maximal 1024 Vorgänger beschränkt (van den Oord et al., 2016, S.3).

Die Kernkomponenten des *WaveNets* sind sogenannte *Dialed Casual Convolutions*. Sie ermöglichen die folgenden Eigenschaften:

1. Sie stellen sicher, dass das *Modell* die Reihenfolge, in der die Daten modelliert sind, nicht verletzt wird: Die Vorhersage $p(x_{t+1}|x_1, ..., x_t)$, die vom *Modell* zum Zeitpunkt t berechnet wird, kann von keinem zukünftigen Zeitschritt $x_{t+1}, ..., x_T$ abhängen (van den Oord et al., 2016, S. 2).

2. Sie ermöglichen den Aufnahmebereich der Vorgängerdaten mit linear zunehmender Anzahl von Parametern exponentiell zu vergrößern. Das wird erreicht, indem die Eingangswerte in einem bestimmten Schritt (die als *Dilation* bezeichnet wird) übersprungen werden (van den Oord et al., 2016, S. 3).

Die folgende Abbildung 3.1 stellt das Verhalten der *Dialed Causal Convolutions* schematisch dar.

Über eine Erweiterung der *Activation-Funktion* wird das Konditionieren des *WaveNets* ermöglicht, sodass die *Abtastwerte* von weiteren Eingangsvariablen abhängen können van den Oord et al. (2016)[S. 4]. Diese Eingangsvariablen können im Beispiel der Sprachsyntetisierung klangliche Merkmale sein, die aus Text extrahiert werden. Die oben dargestellte Wahrscheinlichkeitsverteilung der *Abtastwerte* wird um die Eingangsvariable h erweitert:

$$p(x|h) = \prod_{t=1}^{T} p(x_t|x_1, ..., x_{t-1}, h) \tag{3.2}$$

Den detailierten Aufbau des *WaveNets* darzustellen überschreitet den Rahmen der vorliegenden Arbeit. Für eine ausführliche Erörterung wird auf die Quellen[2] verwiesen.

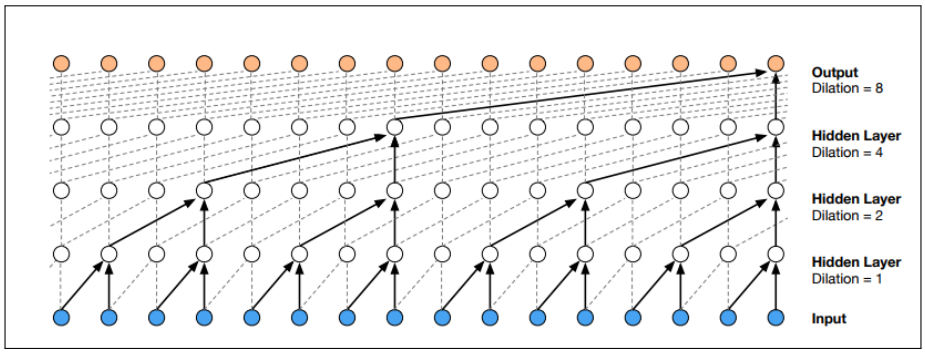


Abbildung 3.1: Schematische Darstellung der Funktionsweise von *Dilated Casual Convloutions* (van den Oord et al., 2016, S. 3).

Die Abbildung zeigt die zwei oben beschriebenen Eigenschaften: Die Eingabewerte - dargestellt als blaue Punkte - haben in der darauffolgenden Schicht keine nach links führenden Verbindungen, was die Reihenfolge der Informationen sicherstellt. Die *Dilation* gibt an, wieviele Verbindungen übersprungen werden - sie sind mit gestrichelter Linie dargestellt. Mit jeder neuen Schicht wird die *Dilation* verdoppelt, was es ermöglicht, den Aufnahmebereich der Vorgängerdaten mit linear zunehmender Anzahl von Parametern exponentiell zu vergrößern.

Der Ausgabewert des *Modells* ist eine Wahrscheinlichkeitsverteilung mit 256 Werten für den nächsten *Abtastwert*, die mithilfe einer *Softmax-Schicht* modelliert wird. Dieser

[2] https://medium.com/@kion.kim/wavenet-a-network-good-to-know-7caaae735435
https://www.analyticsvidhya.com/blog/2020/01/how-to-perform-automatic-music-generation/

Wert wird anschließend in ein 16-Bit-Ganzzahlwert umgewandelt und kann als *Abtastwert* an die *Wellenform* angehängt werden. (van den Oord et al., 2016, S. 3f)

Zum Generieren neuer Daten werden folgende Schritte unternommen (van den Oord et al., 2016, Pai, 2020):

1. Ein zufälliges Array mit Abtastwerten wird erzeugt und dem *Modell* als Eingangsgröße zusätzlich zu der oben beschriebenen Konditionierungsvariable h übergeben.
2. Das *WaveNet* gibt die Wahrscheinlichkeitsverteilung des nächsten *Abtastwerts* mit 256 möglichen Werten zurück.
3. Der wahrscheinlichste Wert wird in eine 16-Bit-Ganzzahl umgewandelt, was dem nächsten *Abtastwert* entspricht.
4. Der erste Wert des in Schritt 1. generierten Arrays wird entfernt und das *Abtastwert* von Schritt 3. wird am Ende des Arrays angehängt.
5. Mit dem neuen Array werden die Schritte wiederholt bis die gewünschte Länge des Audiosignals erreicht wird.

Die Ergebnisse der generierten Audiosignale haben eine Abtastrate von 24 kHz und erzielen bei der Sprachsynthese menschenähnliche Ergebnisse. Die Autoren evaluieren die Qualität der Audiosignale qualitativ mithilfe von Beurteilungen von Probanden:

Es wird der *Mean Opinion Score (MOS)* erhoben, in dem die Probanden nach jedem Hörbeispiel dessen Natürlichkeit auf einer fünfstufigen Skala bewerten (1: Sehr schlecht, 2: Schlecht, 3: Angemessen, 4: Gut, 5: Ausgezeichnet) (van den Oord et al., 2016, S.6). Die Ergebnisse eines *text to speech* Experiments sind sehr nah an dem *MOS* von gesprochener Sprache von Menschen: Das Ergebnis von englisch generierten Texten ist $4,21$ im Vergleich zu natürlich gesprochenem Text $4,55$.

Sowohl das Generieren neuer Daten als auch das Training des *WaveNets* ist aufwändig. Das liegt daran, dass *Abtastwert* für *Abtastwert* generiert und wieder ins *Modell* zurückgeführt werden muss. Für ein einsekündiges Signal muss dieser Vorgang 24.000 mal wiederholt werden. Auch die Optimierungen des *WaveNets* (van den Oord et al., 2017) schaffen in dieser Hinsicht keine ausreichende Abhilfe, um sie in dieser Arbeit als potenzielle Kandidaten für ein *Modell* auszuwählen. Es wird in der vorliegenden Arbeit auf eine ressourcenschonende Methode zurückgegriffen.

Es folgt die Beschreibung eines weiteren Ansatzes der als *Tacotron 2* bekannt ist.

3.3 Text-To-Speech Synthese durch die Konditionierung des WaveNets auf Mel-Spektrogramm-Vorhersagen

Dieser Abschnitt beschreibt die für die vorliegende Arbeit relevanten Konzepte von Shen et al. (2017) und fasst dazu den Anwendungsfall und die Architektur des Systems zusammen. Eine detaillierte Erörterung der Komponenten wird nicht ausgeführt, da sie in dem Detailgrad für die vorliegende Arbeit irrelevant sind und es gleichzeitig den Rahmen der Arbeit überschreiten würde.

Die Architektur, die in der Veröffentlichung vorgestellt wird, wird als *Tacotron 2* bezeichnet. Das Ziel ist es, gesprochene Sprache direkt aus Texteingaben zu synthetisieren ohne spezifisches Domänenwissen, wie sprachliche Merkmale, zu besitzen. Das System besteht aus zwei hintereinandergeschalteten *Neuronalen Netzen* (Shen et al., 2017, S. 1):

1. Das erste *Neuronale Netz* dient zur Erzeugung von *Mel-Spektrogrammen*. Der zu synthetisierende Text stellt die Eingangsgröße des *Netzes* dar. Diese erste Komponente ist ein *recurrent sequenz-to-sequenz Netz*, die im Wesentlichen aus *LSTM-* und *Conv2D-Schichten* besteht. (Shen et al., 2017, S. 2)

2. Das zweite *Netz* wandelt die *Mel-Spektrogramme* des ersten *Neuronalen Netzes* in *Wellenform* um. Es baut auf dem oben beschriebenen *WaveNet* auf und wird als *Vocoder* bezeichnet. Mit der Umwandlung der *Mel-Spektrogramme* mithilfe des modifizierten *WaveNets* statt mithilfe des *Griffin-Lim* Algorithmus erhoffen sich die Autoren einen geringeren Qualitätsverlust (Shen et al., 2017, S. 1ff).

Die Abbildung 3.2 skizziert den beschriebenen Aufbau der zwei *Netze*. Alle hier dargestellten Komponenten bis zum *Mel-Spektrogramm* gehören zu dem ersten *Netz*. Das *Mel-Spektrogramm* dient als Eingangsgröße für das zweite *Netz* dem *WaveNet Vocoder*, welches daraus das Audiosignal in *Wellenform* erzeugt.

Die Ergebnisse der von *Tacotron 2* generierten Audiosignale haben eine Abtastrate von 24 kHz und erzielen bei der Sprachsynthese bessere Ergebnisse als das oben beschriebene *WaveNet*. Das Evaluieren der Audiosignale erfolgt qualitativ durch die Ermittlung des oben beschriebenen *MOS*. *Tacotron 2* erzielt ein *MOS* von $4,526 \pm 0,066$, was im Vergleich zum *MOS* der Trainingsdaten $4,582 \pm 0,053$ sehr ähnlich ist. Beispiele der generierten Audiodateien sind hier[3] zu finden.

[3] https://google.github.io/tacotron/publications/tacotron2

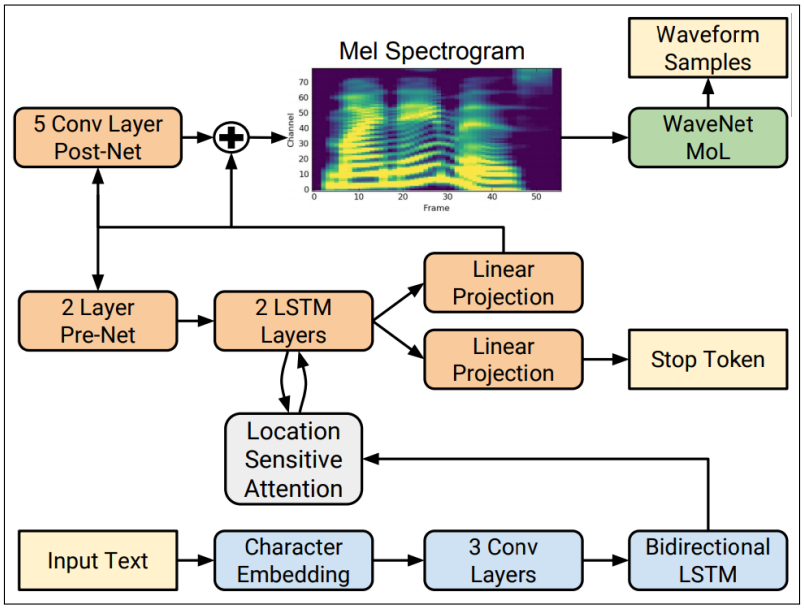


Abbildung 3.2: Aufbau der Tacotron 2 Architektur Shen et al. (2017)[S. 2].

Die vorliegende Arbeit adaptiert die Idee von *Tacotron 2* ein *Vocoder* als Schicht zur Umwandlung von *Mel-Spektrogrammen* zu nutzen, um Vogelgesang zu generieren.

Der nachfolgende Abschnitt geht auf die Forschungsfragen, die im Rahmen dieser Arbeit erörtert werden sollen. Darauf aufbauend werden Anforderungen definiert, die für die Versuchsvorbereitung und Implementierung relevant sind.

4 Forschungsfragen und Anforderungen

Dieser Abschnitt formuliert Forschungsfragen, die sich aus der in Kapitel 1 definierten Zielsetzung ergeben. Aus diesen Forschungsfragen leiten sich Anforderungen an die Architektur und die Versuchsvorbereitung ab, die abschließend ausgearbeitet werden.

Wie einleitend erarbeitet, ist das Ziel der vorliegenden Arbeit ein Konzept mit *Wasserstein-GANs* zu entwickeln, um Vogelgesang zu synthetisieren. Unter Berücksichtigung der Erkenntnisse der Voruntersuchung aus Abschnitt A.1 und der aufgeführten verwandten Arbeiten in Kapitel 3 bietet sich für die Synthese von Vogelgesang die Verwendung von *Mel-Spektrogrammen* an, die mithilfe eines vortrainierten *Vocoders* in *Wellenform* umgewandelt werden.

Aus dieser Zielsetzung lassen sich Forschungsfragen ableiten, die im Folgenden näher erläutert werden.

4.1 Forschungsfragen

Die aufgestellten Forschungsfragen gliedern sich in unterschiedliche Themengebiete ein. Sie werden durch eine eindeutige Kennung gekennzeichnet, worauf im Laufe der Arbeit Bezug genommen wird.

Die ersten beiden Forschungsfragen adressieren die Eignung des *WGANs* und des vortrainierten *Vocoders* für die vorliegende Zielstellung. Es soll überprüft werden, ob sich *WGANs* für die Synthese von Vogelgesang in Form von *Mel-Spektrogrammen* eignen (FF1) und ob ein für die Sprachsynthese vortrainierter *Vocoder Mel-Spektrogramme* mit Vogelgesang ohne Qualitätsverlust in *Wellenform* umwandeln kann (FF2).

Unter der Annahme einer positiven Beantwortung der ersten beiden Fragen soll anschließend überprüft werden, ob das Zusammenspiel beider Komponenten die Synthese von Vogelgesang in *Wellenform* ermöglicht (FF3).

Abschließend wird untersucht, ob der *FID-Score* sich für die Evaluation von der Qualität der *Mel-Spektrogramme* eignet (FF4). Da der *FID* unter Berücksichtigung bestimmter Merkmale - wie Kanten und Kurven von Bildern - berechnet wird, ist nicht klar, ob es sich im Kontext von *Mel-Spektrogrammen* eignet.

Zusammengefasst ergeben sich daraus die folgenden Forschungsfragen:

1. Sind *WGANs* in der Lage, Vogelgesang in Form von Mel-Spektrogrammen zu generieren (FF1)?
2. Ist ein vortrainierter *Vocoder*, der für die Sprachsynthese entwickelt ist, für die Umwandlung von *Mel-Spektrogrammen* mit Vogelgesang in *Wellenform* ohne signifikantem Qualitätsverlust geeignet (FF2)?
3. Ermöglicht das Zusammenspiel von *WGAN* und *Vocoder* die Generierung von Vogelgesang in *Wellenform* (FF3)?
4. Ist der *FID-Score* im Kontext der Synthese von Vogelgesang über *Mel-Spektrogramme* ein geeigneter Indikator für ihre Qualität (FF4)?

Aus den hier aufgestellten Fragestellungen ergeben sich konkrete Anforderungen, die im folgenden Abschnitt zusammengefasst werden.

4.2 Anforderungen

Aus der Zielstellung und den oben aufgestellten Forschungsfragen ergeben sich Anforderungen, die sich an die zu entwickelnde Architektur und an den Trainingsaufbau richten. Sie werden im Folgenden erörtert und durch eine eindeutige Kennung gekennzeichnet, worauf im Laufe der Arbeit Bezug genommen wird. Es wird auf eine Unterscheidung zwischen Funktionalen- und Nicht-Funktionalen Anforderungen verzichtet. Die Tabelle 4.1 fasst die Anforderungen zusammen.

Der folgende Abschnitt beschreibt den Aufbau und Implementierung der Architektur. Darüber hinaus wird die Vorbereitung des Trainings beschrieben. Dabei werden die hier aufgestellten Forschungsfragen und Anforderungen berücksichtigt.

Tabelle 4.1: Anforderungen

Kennung	**Bezeichnung**	**Beschreibung**
A1	Flexible Anpassbarkeit der *Hyperparameter*	Die flexible Anpassung der *Hyperparameter* soll eine unkomplizierte Konfiguration des *WGANs* ermöglichen. Das ist notwendig, weil im Vorfeld nicht bekannt ist, ob und welche Konfigurationen des *WGANs Mel-Spektrogramme* abbilden können. Zu den *Hyperparametern* gehören *Netztiefe*, *Filtergröße*, *Stride* und die Ein- und Ausgangsgröße der einzelnen Schichten.
A2	Nachvollziehbarkeit des Trainingsverlaufs	Damit der Trainingsverlauf der verschiedenen Konfigurationen miteinander vergleichbar ist, muss der Trainingsverlauf aufgezeichnet werden. Zu den relevanten Indikatoren gehören der *Loss* des *Generators* und des *Kritikers*. Zusätzlich soll die Diversität der generierten Daten mittels des durchschnittlichen geringsten *euklidischen Abstands* innerhalb der generierten Daten ermittelt werden.
A3	Flexible Konfigurierbarkeit des Trainingsdaten-Generators	Neben der flexiblen Konfiguration der *Hyperparameter* in A1 muss, um verschiedene Trainingsszenarien zu ermöglichen, auch der Trainingsdaten-Generator anpassbar sein. Zu den Parametern gehören das Ausgabeformat der *Mel-Spektrogramme*, die für ihre Berechnung verwendete Signallänge der Aufzeichnungen, die Quelle der Trainingsdaten und die *Batchgröße* der Trainingsdaten.
A4	Messung des *FID-Scores* während des Trainings	Der *FID-Score* muss für die FF4 während des Trainings in definierten Schritten berechnet werden. Zusätzlich soll vor Trainingsbeginn der *FID-Score* innerhalb des Trainingsdatensatzes als Referenz berechnet werden.

Kennung	Bezeichnung	Beschreibung
A5	Ausgabe von generierten Daten während des Trainings	Währen des Trainings sollen in definierten Schritten Beispieldaten mithilfe des aktuellen Zustands des *Generators* erzeugt werden. Gleichzeitig sollen als Referenz Beispieldaten vom Trainigsdatensatz ausgegeben werden.
A6	Speichern der Zustände von *Generator* und *Kritiker* während des Trainings	Es existiert kein objektives Maß, welches den optimalen *Generator* und *Kritiker* ermittelt, sodass während des Trainings zu festgelegten Zeitpunkten der aktuelle *Generator* und *Kritiker* für eine nachträgliche Analyse gespeichert werden sollen.

5 Architektur und Versuchsvorbereitung

Dieser Abschnitt stellt die Architektur für die Synthese von Vogelgesang vor. Darüber hinaus wird der Aufbau und die Vorbereitung der Versuchsdurchführung beschrieben. Dabei wird auf den Datensatz, das Trainingskonzept und auf die wichtigsten Implementierungsdetails eingegangen.

5.1 Architektur

Zunächst wird der Aufbau der Architektur dargestellt, der aus zwei *Neuronalen Netzen* besteht. Abschließend wird eine Plausibilitätsprüfung der Architektur durchgeführt, die im Vorfeld einige potenzielle Fehlerquellen überprüft.

5.1.1 Aufbau der Architektur

Dieser Abschnitt skizziert die Architektur und beschreibt ihre Komponenten. Die konkreten *Hyperparameter*, wie die Anzahl und Größe der Schichten oder die Ein- und Ausgangsgrößen, werden in der Versuchsdurchführung in Kapitel 6 konkretisiert, wo verschiedene Konfigurationen erprobt werden.

Die Architektur besteht aus den folgenden drei Komponenten, wovon der *WGAN* und der *Vocoder* ein *Neuronales Netz* sind:

1. Einem *WGAN*, der dafür zuständig ist, *Mel-Spektrogramme* der Signale zu generieren. Die Aktivierungsfunktion der letzten Schicht des *Generators* stellt der *tangens hyperbolicus* (pytorch, 2019) dar, sodass die Ausgabewerte zwischen [-1; 1] liegen.
2. Einem *Skalierer*, der die Ausgabe des *WGANs* mit Werten im Bereich von [-1; 1] in den Wertebereich eines *Mel-Spektrogramms* hoch skaliert.

3. Einem *Vocoder*, der nach dem Vorbild von Shen et al. (2017) die *Mel-Spektrogramme* in Audiosignale umwandelt.

Die Abbildung 5.1 stellt das Zusammenspiel dieser Komponenten dar, die im Folgenden genauer beschrieben werden.

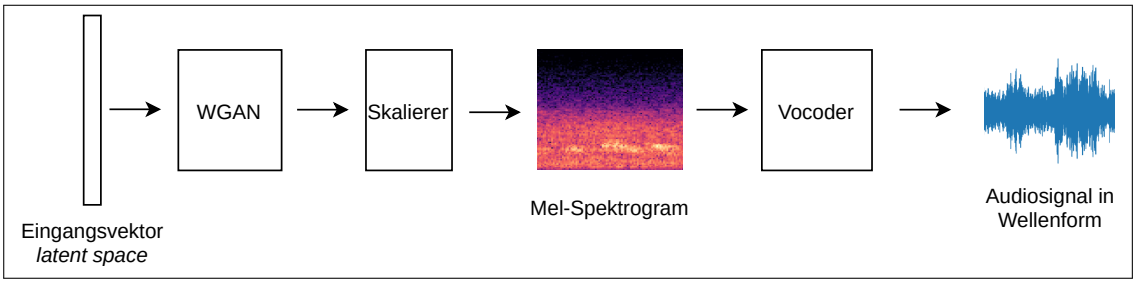


Abbildung 5.1: Aufbau der Architektur zur Synthese von Vogelgesang.

Die Architektur besteht aus drei Komponenten: Dem *WGAN*, der normalisierte *Mel-Spektrogramme* des Vogelgesangs erzeugt, einem *Skalierer*, der die normalisierten *Mel-Spektrogramme* hoch skaliert und dem *Vocoder*, der daraus Audiosignale in *Wellenform* erstellt.

Aufbau des WGANs

Der Aufbau des *WGANs*, der für das Generieren der *Mel-Spektrogramme* zuständig ist, basiert auf einer Kombination des in Abschnitt 2.4 beschriebenen *WGANs* mit *gradient panelty* und dem von Radford et al. (2015) entwickelten *DCGAN*.

Der *Generator* besteht aus zweidimensionalen *Convolutional-Transpose-Schichten* mit der *ReLU-Aktivierungsfunktion*, worauf *Batch-Normalizations* folgen. Eine Skizze des *Generators* ist in Abbildung 2.5 dargestellt. Die Eingangsgröße besteht aus einem *Batch* aus Vektoren des *latent spaces* und die Ausgangsgröße aus einem *Batch* mit *Mel-Spektrogrammen*. Da die Aktivierungsfunktion der letzten Schicht der *tangens hyperbolicus* ist, sind die Ausgangswerte auf den Bereich von [-1; 1] begrenzt, sodass die ausgegebenen Werte in den Wertebereich eines *Mel-Spekrogramms* umgeformt werden müssen. Die Umformung erfolgt durch den *Skalierer*, der weiter unten beschrieben wird.

Der *Kritiker* besteht aus zweidimensionalen *Convolutional-Schichten* mit einer *LeakyReLU-Aktivierungsfunktion*, worauf *Batch-Normalizations* folgen. Der *Kritiker* ist analog zum *Generator* aufgebaut mit dem Unterschied, dass die *Convolutional-Transpose-Schichten*

durch zweidimensionale *Convolutional-Schichten* ersetzt sind. Darüberhinaus ist das Format der Eingangsgröße des *Kritikers* gleich der Ausgangsgröße des *Generators*. Der *Kritiker* gibt einen Skalar zwischen [-1; 1] zurück.

Konkrete Konfigurationen des *Generators* können aus Tabelle 6.1 und des *Kritikers* aus Tabelle 6.2 in Kapitel 6 entnommen werden.

Für den weiteren Verlauf der Implementierung sind die folgenden Parameter relevant, deren Werte sich je nach Versuchskonfiguration voneinander unterscheiden:

- Die Netztiefe, die die Anzahl der Schichten widerspiegelt.
- Die Filtergröße der *Convolutional-* bzw. *Convolutional-Transpose-Schichten*.
- Die Schrittgröße (bzw. *Stride*) des Filters.
- Das *Padding* des Eingangssignals je Schicht.
- Das *Padding* des Ausgangssignals je Schicht (nur bei dem *Generator*).
- Die Ein- und Ausgangsgröße des *Generators* bzw. *Kritikers*.

Die Implementierung des *WGANs* ist in Abschnitt 5.4 vorgestellt. Sie ermöglicht eine flexible Anpassung dieser Parameter.

Skalierer

Der *Skalierer* nimmt die Ausgabewerte des *Generators* entgegen, die im Wertebereich zwischen [-1; 1] liegen und bringt sie in den Wertebereich eines *Mel-Spektrogramms*. Als Implementierung wird der *MaxAbsScaler*[1] der *Python* Bibliothek *sklearn* verwendet.

Dieser hat sich in einer Voruntersuchung mit dem *MaxAbsScaler*, *MinMaxScaler*[2] und einer *modifizierten tanh-Normalisierung* (Latha und Tamilnadu, 2011), als am effektivsten herausgestellt.

[1] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MaxAbsScaler.html
[2] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

Initial bedarf es, den *MaxAbsScaler* auf Basis des Trainingsdatensatzes zu trainieren. Dafür wird der vollständige Datensatz verwendet. Der Trainingsaufbau des *MaxAbsScaler* kann aus dem *Repository*[3] entnommen werden.

Vocoder

Der *Vocoder* dient zur Umwandlung der *Mel-Spektrogramme* in das Signal in *Wellenform*. Die Implementierung und das Training des *Vocoders* ist nicht Teil der vorliegenden Arbeit, da es die Kapazität dieser Arbeit überschreiten würde. Stattdessen wird sich eines bereits implementierten und vortrainierten *Modells* bedient, welches im Rahmen der Sprachsynthese entwickelt wurde (Shen et al., 2017).

Der *Vocoder* basiert auf dem in der Veröffentlichung Prenger et al. (2018) vorgestellten *Modell WaveGlow*. Das vortrainierte *Modell* stammt aus dem *PyTorch hub* NVIDIA (2020) und erzeugt Audiosignale in *Wellenform* mit einer Abstandsrate von 22050 Hz. Die *Mel-Spektrogramme*, die als Eingangssignal fungieren, werden mit den folgenden Parametern erzeugt[4]:

- Abstandsrate der Audiosignale: 22050 Hz
- FFTs: 1024
- Fenstertyp: hann
- Fenstergröße: 1024
- Hop-Länge: 256
- Anzahl erzeugter *Mels*: 80

Die Größe der *Mel-Spektrogramme*, die das *Modell* als Eingangssignal akzeptiert, sind in der Höhe von 80 *Mels* beschränkt. Die Länge des Signals ist variabel. Ein 80×80 *Mel-Spektrogramm* wird in ein Audiosignal der Länge von rund $0,92$ Sekunden umgewandelt.

[3] https://github.com/batonfabi/master_thesis/blob/submission/kapitel_5/TrainScaler.ipynb

[4] Diese Werte stammen nicht aus der Veröffentlichung, sondern aus der Konfigurationsdatei der Implementierung aus dem *Repository* https://github.com/NVIDIA/waveglow/blob/5bc2a53e20b3b533362f974cfa1ea0267ae1c2b1/config.json

Da das vortrainierte *Modell* im Rahmen der Sprachsynthese genutzt wird, bedarf es im Folgenden einer Prüfung, ob sich die hier vorgestellte Architektur auch für die Generierung von Vogelgesang eignet (vgl. FF2).

5.1.2 Plausibilitätsprüfung

Die beschriebene Architektur birgt an den Schnittstellen der Komponenten folgende potenzielle Probleme, an denen die Synthese des Vogelgesangs scheitern kann:

- Der *WGAN* kann die Signale in Form des *Mel-Spektrogramms* nicht generalisieren oder nicht in der Qualität erzeugen, wie der *Vocoder* es benötigt.
- Der *Skalierer* scheitert an der Hochskalierung der *Mel-Spektrogramme*, die vom *Generator* kommen.
- Der *Vocoder* erzeugt aus den generierten *Mel-Spektrogrammen* mit Vogelgesang keine gute *Wellenform*. Das könnte der Fall sein, weil der eingesetzte *Vocoder* für den Anwendungsfall der Sprachsynthese trainiert wurde und die Frequenzen des Vogelgesangs evtl. nicht in der benötigten Qualität umgewandelt werden können.

Die Unsicherheiten können zum gewissen Teil vorab relativiert werden. Die Arbeit von Donahue et al. (2018) zeigt, dass *GANs* grundsätzlich in der Lage sind Strukturen, wie *Spektrogramme* abzubilden. Es bleibt zu prüfen, ob die Qualität für den eingesetzten *Vocoder* ausreicht.

Ob sich der *MaxAbsScaler* für den Anwendungsfall eignet, kann gleichzeitig mit der Eignung des *Vocoders* für die Synthese von Vogelgesang überprüft werden, was der FF2 gleicht. Aus praktischen Gründen wird diese Untersuchung zum Zeitpunkt der Versuchsvorbereitung vorgezogen. Bei einer negativen Beantwortung der FF2 müsste nämlich die Architektur angepasst werden und die nachfolgende Implementierung wäre unter Umständen obsolet.

Das Ergebnis dieser Prüfung ist positiv. Der konkrete Versuchsaufbau und dessen Durchführung wird mit der Beantwortung der FF2 in Unterabschnitt 6.1.2 beschrieben.

Bevor auf den Trainingsaufbau und die Implementierungsdetails eingegangen wird, beschreibt der nächste Abschnitt den vorhandenen Datensatz.

5.2 Datensatz

Der für das Training verwendete Datensatz besteht aus Aufzeichnungen von zwei Vogelarten mit einer gesamten Aufzeichnungslänge von 61.9 Std. Die Auswahl liegt der Überlegung zu Grunde, einen relativ homogenen und gleichzeitig großen Datensatz zu verwenden, um die Komplexität des Problems im Vergleich zu einem Datensatz mit vielen Vogelarten zu reduzieren. Die Daten werden aus der Webseite *xeno-canto*[5] bezogen.

Die Tabelle 5.1 fasst die wesentlichen Informationen zu dem Datensatz zusammen.

Tabelle 5.1: Verteilung des Trainingdatensatzes nach der Bereinigung

Vogelart	Dauer	Anzahl Dateien
Fitis	1460 Min.	1405
Kohlmeise	2253 Min.	3378
Summe	61.9 Std.	4783

Die von *xeno-canto* stammenden Daten variieren im Aufzeichnungsformat und der Qualität. Der vorliegende Datensatz enthält Stör- oder Hintergrundgeräusche, Aufzeichnungen, in denen unterschiedliche Vogelarten zu hören sind oder längere Phasen, in denen der Vogel nicht zu hören ist. Der letzte Punkt stellt ein Problem dar, weil die *Mel-Spektrogramme* beim Training in einem sich über den Aufzeichnungen bewegenden Fenster erzeugt werden und dadurch Trainingsdaten ohne Vogelgesang entstehen können (näheres dazu in Unterabschnitt 5.3.1).

Der Großteil der genannten Unregelmäßigkeiten lässt sich nicht im Rahmen der vorliegenden Arbeit beheben. Es werden nur die zwei folgenden Schritte zur Bereinigung und Vereinheitlichung vorgenommen:

- Vereinheitlichung des Aufzeichnungsformats. Alle Audiosignale werden monophon in den *flac* Datentypen mit einer Abtastrate von 22050 Hz umgewandelt.
- Alle geräuschlosen Stellen, die länger sind als zwei Sekunden, werden aus den Aufzeichnungen automatisiert ausgeschnitten[6]. Die zwei Sekunden Puffer sollen sicherstellen, dass keine natürlichen Pausen zwischen den Strophen des Vogelgesangs abgeschnitten werden. Ein beispielhaftes Ergebnis einer Bereinigung dieser Art kann

[5] https://www.xeno-canto.org/

[6] Die geräuschlosen Stellen werden mithilfe der Kommandozeilenanwendung SoX - Sound eXchange entfernt. Dazu wird der folgende Aufruf sox silence -l 1 0.1 1% -1 2.0 1% für jede Audiodatei ausgeführt.

in dem zu der Projektarbeit zugehörigem *Repository*[7] gefunden werden. Hier wird eine 146-sekündinge Aufzeichnung auf 83 Sekunden gekürzt.

Der folgende Abschnitt beschreibt den Trainingsaufbau und zeigt, wie die hier vorgestellten Trainingsdaten während des Trainings verwendet werden.

5.3 Trainingsaufbau

Dieser Abschnitt beschreibt den Trainingsaufbau des *WGANs*. Dabei wird die Generierung der Trainingsdaten und die Rolle des *Skalierers* während des Trainingsprozesses genauer betrachtet. Anschließend wird die Protokollierung des Trainingsverlaufs beschrieben.

Die Abbildung 5.2 skizziert das Zusammenspiel der beim Training zum Einsatz kommenden Komponenten. Die Trainingsdaten werden von einem *Trainingsdaten-Generator* vorverarbeitet, der zur Laufzeit aus den Audiodateien randomisierte *Batches*, bestehend aus *Mel-Spektrogrammen*, erzeugt.

Wie in Abschnitt 5.1.1 beschrieben, besteht die Aktivierungsfunktion der letzten Schicht des *Generators* aus dem *tangens hyperbolicus*, sodass dessen Ausgabewerte zwischen [-1; 1] liegen. Die Werte der *Mel-Spektrogramme* können außerhalb dieses Intervalls liegen, sodass diese für das Training entsprechend skaliert werden müssen. Die *Batches* werden dazu von dem oben beschriebenen und entsprechend vortrainierten *Skalierer* in die benötigte Form umgewandelt. Die skalierten *Batches* werden abschließend für das Training verwendet.

Dadurch, dass die *Mel-Spektrogramme* vor dem Training vom *Skalierer* umgewandelt werden, müssen sie nach dem Generieren von dem *Skalierer* rückgängig umgewandelt werden, um sie für die Weiterverwendung für den *Vocoder* bereit zu machen. Die Abbildung 5.3 skizziert das Zusammenspiel des *Generators* mit dem *Skalierer* beim Generieren von *Mel-Spektrogrammen* für den *Vocoder*.

Im Weiteren wird die Komponente, die für die Generierung der Trainingsdaten zuständig ist, genauer beschrieben.

[7] https://github.com/batonfabi/master_thesis/tree/submission/kapitel_5/audio_samples/cleaning_up

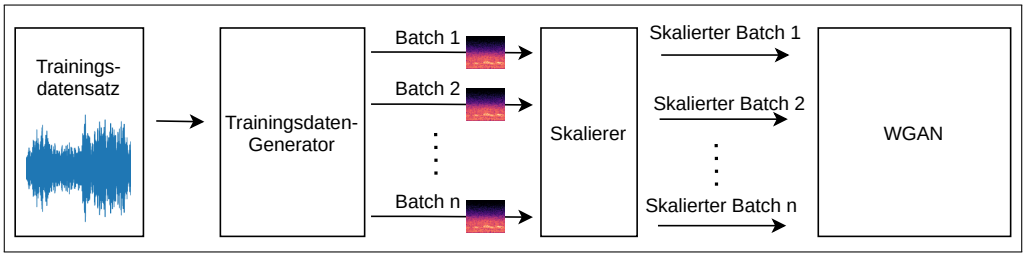


Abbildung 5.2: Zusammenspiel der Komponenten während des Trainings.

Während des Trainings sind drei Komponenten maßgeblich im Einsatz. Der *Trainingsdaten-Generator* portioniert und verarbeitet die Daten aus dem Trainingsdatensatz. Der *Skalierer* normalisiert die vom *Trainingsdaten-Generator* stammenden Daten in das Intervall [-1; 1], die abschließend vom *WGAN* für das Training verwendet werden.

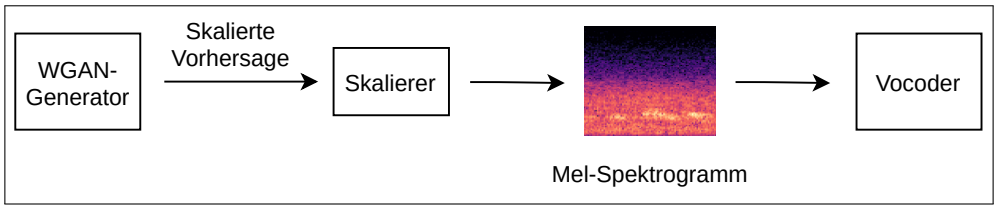


Abbildung 5.3: Zusammenspiel der Komponenten beim Generieren neuer Daten.

Beim Generieren neuer Daten werden die vom *Generator* erzeugten Datensätze, die sich im Intervall von [-1; 1] bewegen, vom *Skalierer* in die Skala eines *Mel-Spektrogramms* gebracht. Die so erzeugten *Mel-Spektrogramme* werden anschließend vom *Vocoder* in Audiosignale in *Wellenform* umgewandelt.

5.3.1 Der Trainingsdaten-Generator

Der *Trainingdaten-Generator* liest bei der Initialisierung den zu verwendenden Trainingsdatensatz aus und analysiert die vorhandene Menge der Trainingsdaten. Gleichzeitig werden die Aufzeichnungen mithilfe eines sich *bewegenden Fensters* logisch in die gewünschte Länge portioniert und die Information in einer Liste aufgezeichnet. Diese Liste hält die Information wie *Pfad zur Audiodatei*, *Startposition*, ab der das *Mel-Spektrogramm* berechnet werden soll, *Endposition* des Signals für die Berechnung des *Mel-Spektrogramms*.

Diese Liste dient zur Erzeugung der *Batches* und wird mit jeder neuen Trainingsepoche gemischt. Die *Mel-Spektrogramme* werden zur Laufzeit des Trainings berechnet, was im Vergleich zu vorberechneten *Mel-Spektrogrammen* länger dauert. Das bringt den Vorteil mit sich, verschiedene Parameter wie die Größe der *Mel-Spektrogramme* oder die

Schrittgröße des bewegten Fensters für unterschiedliche Trainingsszenarien flexibel zu konfigurieren.

Die Abbildung 5.4 skizziert den Mechanismus des hier beschriebenen *bewegenden Fensters*. Die Fenstergröße entspricht der Länge des Signals, die verwendet wird, um die *Mel-Spektrogramme* zu berechnen. Das Fenster wird schrittweise über das Signal gelegt und so werden entsprechend der Signallänge die *Mel-Spektrogramme* erzeugt.

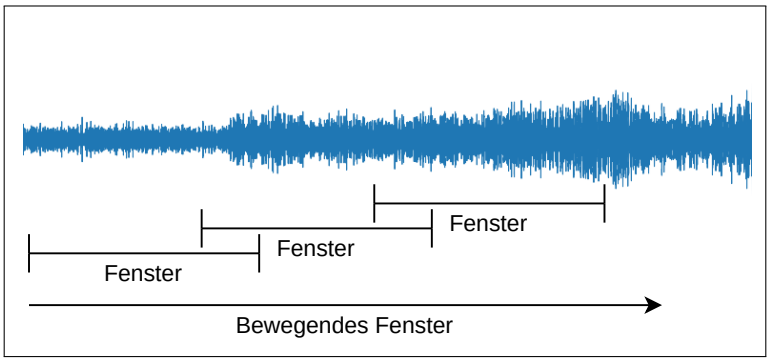


Abbildung 5.4: Funktionsweise des sich bewegenden Fensters, das für die Berechnung der *Mel-Spektrogramme* verwendet wird.

Das sich bewegende Fenster teilt die vorhandenen Audiosignale des Trainingsdatensatzes in logische Teile auf, die für die Generierung der *Mel-Spektrogramme* genutzt werden.

Der *Trainingsdaten-Generator* ermöglicht die flexible Einstellung der folgenden Parameter und erfüllt damit die in A3 definierten Anforderungen:

- Die Übergabe einer Liste mit den Ordnern der einzelnen Vogelarten, die als Datenquelle genutzt werden sollen, ermöglicht eine flexible Zusammensetzung der Trainingsdaten.
- Die Fenstergröße des sich bewegenden Fensters und somit die Signaldauer, die ein *Mel-Spektrogramm* abbildet, kann definiert werden.
- Schrittgröße des sich bewegenden Fensters ist flexibel konfigurierbar.
- Größe der erzeugten *Batches* ist flexibel konfigurierbar.
- Die Parameter zur Berechnung der *Mel-Spektrogramme* ist anpassbar, sodass auch andere *Vocoder* untersucht werden können.

Die Architektur und Implementierung des *Trainingsdaten-Generators* sind für das weitere Verständnis und für die Interpretation der Ergebnisse nicht weiter relevant, sodass keine weiteren Details beschrieben werden. Die konkrete Implementierung kann aus dem *Repository*[8] entnommen werden.

Im nachfolgenden Abschnitt wird beschrieben, welche Informationen des Trainingsverlaufs zu welchem Zeitpunkt protokolliert werden.

5.3.2 Protokollierung des Trainingsverlaufs

Die definierten Anforderungen A2, A4 und A5 beziehen sich auf die Informationen, die während des Trainings gesammelt werden sollen, um die unterschiedlichen Konfigurationen der Architektur miteinander vergleichen zu können. Zusammengefasst sollen die folgenden Informationen aufgezeichnet werden:

1. Der Verlauf des *Losses* des *Generators* und *Kritikers*.
2. Der Verlauf des durchschnittlichen geringsten *euklidischen Abstands* innerhalb der vom *Generator* erzeugten Daten. Der durchschnittliche geringste *euklidische Abstand* dient als Indikator für die Diversität der generierten Daten. Für die Berechnung werden 1000 zufällig entnommene Datenbeispiele verwendet. Um einen Vergleichswert zu haben, wird initial analog der durchschnittliche geringste *euklidische Abstand* aus dem Trainingsdatensatz gemessen.
3. Der Verlauf des durchschnittlichen *FID-Scores* der generierten Daten in Bezug zu dem originalen Datensatz. Für die Berechnung werden aufgrund der aufwändigen Berechnung jedes Mal 20 zufällige *Batches* verwendet. Um ein Vergleichswert zu haben, wird initial der durchschnittliche *FID-Score* von 20 *Batches* aus dem Trainingsdatensatz gemessen.
4. Die Entwicklung der vom *Generator* erzeugten Daten. Dazu soll initial beim Trainingsbeginn ein Satz von Vektoren aus dem *latent space* und daraus im Laufe des Trainings kontinuierlich Stichproben erzeugt werden.
5. Zustand des Generators und Kritikers für eine detaillierte Evaluation im Anschluss.

[8] https://github.com/batonfabi/master_thesis/blob/submission/shared_sources/bird_dataset.py

Die Werte aus Punkt 1. werden kontinuierlich mit jedem Trainingsschritt aufgezeichnet. Die Daten aus den Punkten 2. bis 4. werden aufgrund des Berechnungsaufwands im Abstand von 200 Trainingsschritten aufgezeichnet. Die Speicherung der Zustände des *Kritikers* und des *Generators* erfolgen, um die dadurch anfallende Datenmenge zu reduzieren, alle 600 Schritte.

Als Datensenke der aufgeführten Informationen dient die Anwendung *TensorBoard*[9], die die Analyse des Trainingsverlaufs während der Evaluation in Kapitel 6 unterstützt.

Der folgende Abschnitt gibt Einblicke in die Implementierungsdetails, die das Training des *WGANs* betreffen.

5.4 Implementierungsdetails

Dieser Abschnitt beschreibt die wesentlichen Implementierungsdetails der *WGAN* Architektur und des Trainings. Die Implementierung erfolgt mit der Programmiersprache *Python*[10]. Als Framework für die *Deep Learning* spezifischen Komponenten kommt *Pytorch*[11] zum Einsatz.

Die vorliegende Implementierung orientiert sich abschnittweise an der *WGAN*-Implementierung von *aladdinpersson*[12] und übernimmt an einigen Stellen den Quelltext. Der vollständig entwickelte Quelltext der vorliegenden Arbeit kann aus dem *Repository*[13] entnommen werden.

5.4.1 WGAN Modell

Das *WGAN Modell* implementiert die in Abschnitt 5.1 vorgestellte Architektur. Das Augenmerk bei der Implementierung liegt auf der in A1 definierten Anforderung an das *Modell*. Es soll möglich sein relevante Hypterparameter der Architektur flexibel zu definieren, damit verschiedene Trainingskonfigurationen untersucht werden können. Zu den relevanten Hyperparametern gehören:

[9] https://www.tensorflow.org/tensorboard/
[10] https://www.python.org/
[11] https://pytorch.org/
[12] https://github.com/aladdinpersson/Machine-Learning-Collection/blob/master/ML/Pytorch/GANs/4.%20WGAN-GP/
[13] https://github.com/batonfabi/master_thesis/tree/submission

- Die Anzahl der Schichten.
- Ein- und Ausgangsgrößen der Schichten.
- Die Kernel-Größe.
- Die Schrittgröße des Filters (*Stride*).
- Das *Padding*.

Für dieses Vorhaben wird ein *Daten-Objekt* namens *LayerDO* mit den entsprechenden Attributen entworfen, wie es im Python Quelltext 1 dargestellt ist.

```
1  class LayerDO():
2      def __init__(self, in_features, out_features,
3                   kernel_size, stride, padding,
4                   output_padding = 0):
5          self.in_features = in_features
6          self.out_features = out_features
7          self.kernel_size = kernel_size
8          self.stride = stride
9          self.padding = padding
10         self.output_padding = output_padding
```

Python Quelltext 1: LayerDO

Mithilfe des *LayerDO* wird die flexible Initialisierung des *Generators* bzw. *Kritikers* ermöglicht. Der Python Quelltext 2 zeigt schematisch die Definition der Schichten eines *Modells*. Hier werden drei Schichten erstellt: die erste Schicht (Zeile zwei) hat eine Eingangsgröße von einem *Feature* und gibt 32 *Features* aus. Die *Kernelgröße* ist bei einem Skalar quadratisch und in dem Fall 4×4. Die Schrittgröße (*Stride*) beträgt zwei und das Eingangssignal wird für vor der Berechnung an den Rändern um eins erweitert (*Padding*). Der Ausgang der ersten Schicht (Zeile zwei) stellt den Eingang der zweiten Schicht dar (Zeile drei). Dieses Prinzip setzt sich entsprechend bis zur letzten definierten Schicht in der Liste fort.

Die Liste mit den definierten Schichten wird bei der Initialisierung des *Generators* bzw. *Kritikers* übergeben, wo das *Modell* entsprechend der Werte und der Anzahl der Schichten aufgebaut wird. Der Python Quelltext 3 zeigt die Implementierung des *Generators*.

```
1 layers = []
2 layers.append(LayerDO(1,32,4,2,1))
3 layers.append(LayerDO(32,64,4,2,1))
4 layers.append(LayerDO(64,1,4,2,0))
```

Python Quelltext 2: Konfiguration eines *Modells* mithilfe des *LayerDO*

Bei der Initialisierung des *Generators* wird für jede bis auf der letzten Schicht (vgl. Zeile fünf) ein Block bestehend aus einer *ConvTranspose2d*, einer *Batchnorm2D* und der *ReLU* Aktivierungsfunktion erzeugt. Die *Batchnormalisierung* erfolgt über die *Batchnorm2D*-Schicht nur auf der Batchebene, wie es von Gulrajani et al. (2017) empfohlen wird. Auf die letzte Schicht (ab Zeile sieben) folgt keine Batchnormalisierung und die Aktivierunksfunktion ist *Tanh*, sodass der Wertebereich der Ausgabe zwischen [-1; 1] begrenzt ist.

```
1  class Generator(nn.Module):
2      def __init__(self, layers:LayerDO):
3          super(Generator, self).__init__()
4          nn_layers = []
5          for layer in layers[:-1]:
6              nn_layers.append(self._block(layer))
7          nn_layers.append(self._convTranspose(layers[-1]))
8          nn_layers.append(nn.Tanh())
9          self.gen = nn.Sequential(*nn_layers)
10
11     def _convTranspose(self, layer:LayerDO):
12         return nn.ConvTranspose2d(layer.in_features,
13                 layer.out_features, layer.kernel_size,
14                 layer.stride, layer.padding, layer.output_padding)
15
16     def _block(self, layer:LayerDO):
17         return nn.Sequential(
18             nn.ConvTranspose2d(layer.in_features,
19                         layer.out_features, layer.kernel_size,
20                         layer.stride, layer.padding, layer.output_padding,
21                         bias=False),
22             nn.BatchNorm2d(layer.out_features),
23             nn.ReLU(),
24         )
```

Python Quelltext 3: Implementierung des *Generators*

Die Implementierung des *Kritikers* ist analog zum *Generator* aufgebaut und wird hier nicht weiter ausgeführt. Der vollständige Quelltext beider Modelle kann aus dem *Repository*[14] entnommen werden. Es folgt die Beschreibung der Implementierung des Trainings.

5.4.2 Training

Das Training wird innerhalb der *Trainings-Klasse train.py*[15] durchgeführt. Dort ist im Wesentlichen der in Abschnitt 2.4 beschriebene Trainingsalgorythmus des *WGANs* mit *gradient panelty* und die in Unterabschnitt 5.3.2 beschriebene Protokollierung des Trainingverlaufs implementiert. Der Python Quelltext 4 beschreibt den Trainingsdurchlauf eines *Batches*. Die im Quelltext aufgeführten Kommentare beziehen sich direkt auf die Zeilen im Pseudocode aus dem Unterabschnitt 2.4.3.

Der Großteil der Beschreibung der Implementierung befindet sich in den Kommentaren des Quelltextes. An dieser Stelle wird auf zwei Abweichungen zu dem genannten Pseudocode hingewiesen:

- In der Implementierung ist die Zeile 3 des Pseudocodes nicht vorhanden, weil alle Schritte, die sich zwischen Zeile 3 und Zeile 8 des Pseudocodes befinden, für den gesamten *Batch* gleichzeitig durchgeführt werden.
- Die Berechnung des *gradient panelties* aus Zeile 7 des Pseudocodes ist zu Gunsten der Übersicht in eine separate Methode ausgelagert, die sich im Unterabschnitt A.2.1 befindet.

Der gesamte Quelltext der vorliegenden Arbeit kann aus dem Repository[16] entnommen werden. Im folgenden Abschnitt wird die Versuchsdurchführung mit den erzielten Ergebnissen dargestellt.

[14] https://github.com/batonfabi/master_thesis/blob/submission/shared_sources/models.py
[15] https://github.com/batonfabi/master_thesis/blob/submission/shared_sources/train.py
[16] https://github.com/batonfabi/master_thesis/tree/submission/

```python
# Alle Zeilenangaben in den Kommentaren beziehen sich auf den in
# Unterabschnitt 2.4.3 dargestellten Pseudocode.
def _train_batch(self, real_data):
        cur_batch_size = real_data.shape[0]
        # vgl. Zeile 2: For t = 1, ..., n_critic
        for _ in range(self.critic_iterations):
            # vgl. Zeile 4: hier nur die Initialisierung
            # der latent variable z ~ p(z); z heißt hier noise
            # x ~ p_data kommt als Parameter real_data
            # ε ~ U[0,1] wird in der Methode gradient_panelty erzeugt
            noise = torch.randn(cur_batch_size,
                        self.latent_space_dim, 1, 1).to(self.device)

            # Zeile 5: x̃ ← G_θ(z);  x̃ heißt hier fake
            fake = self.gen(noise)

            # Vorbereitungen für Zeile 7:
            # D_w(x) heißt hier critic_real
            critic_real = self.critic(real_data).reshape(-1)
            # D_w(x̃) heißt hier critic_fake
            critic_fake = self.critic(fake).reshape(-1)

            # Berechnung des gradient paneltys aus Zeile 7, inkl. Zeile 6:
            # (||∇_x̂ D_w(x̂)||_2 − 1)^2 heißt hier gp
            gp = gradient_penalty(self.critic, real_data,
                        fake, device=self.device)

            # Zeile 7: L^(i) heißt hier loss_critic
            loss_critic = torch.mean(critic_fake)
                        - torch.mean(critic_real)
                        + self.lambda_gp* gp)

            # Gradientenaktualisierung aus Zeile 9
            self.critic.zero_grad()
            # retain_graph ermöglicht eine weitere Gradientaktualisierung
            # des Generators
            loss_critic.backward(retain_graph=True)
            self.opt_critic.step()

        # Gradientenaktualisierung aus Zeile 12:
        # D_w(G_θ(z)) ist hier gen_fake
        gen_fake = self.critic(fake).reshape(-1)
        loss_gen = -torch.mean(gen_fake)
        self.gen.zero_grad()
        loss_gen.backward()
        self.opt_gen.step()
```

Python Quelltext 4: Implementierung des Trainings von *Generator* und *Kritiker* mit einem *Batch*.

6 Versuchsdurchführung und Ergebnisse

Dieser Abschnitt widmet sich der sukzessiven Erörterung der in Kapitel 4 aufgestellten Forschungsfragen. Einleitend wird zu jeder Forschungsfrage das Vorgehen für ihre Beantwortung beschrieben. Anschließend werden die entsprechenden Versuche durchgeführt und auf Basis der resultierenden Ergebnisse die Fragen beantwortet.

Darauf folgt in Abschnitt 6.2 eine qualitative Analyse des *Modells* mit dem niedrigsten *FID-Score*. Abschließend werden die hier gewonnenen Erkenntnisse zusammengefasst.

6.1 Evaluation der Forschungsfragen

6.1.1 Generierung von Mel-Spektrogrammen mit WGANs (FF1)

Frage: Sind *WGANs* in der Lage, Vogelgesang in Form von Mel-Spektrogrammen zu generieren?

Um diese Frage zu beantworten und gleichzeitig ein möglichst gutes Ergebnis zu erzielen, werden unterschiedliche Konfigurationen des *Generators* trainiert. Da zum Zeitpunkt des Trainings kein objektives Maß für die Bewertung der generierten *Mel-Spektrogramme* bekannt ist, kann der Trainingsprozess nicht auf Grundlage eines Trainingserfolges beendet werden. Um trotzdem gute Ergebnisse in Bezug auf die Qualität der generierten Daten zu erzielen, werden die *Modelle* ausgiebig trainiert und die in Unterabschnitt 5.3.2 beschriebenen Informationen aufgezeichnet. Auf Grundlage dieser Protokollierung erfolgt die Analyse der ersten Forschungsfrage.

Der nachfolgende Abschnitt beschreibt den konkreten Aufbau und die Durchführung des Trainings. Anschließend werden die Ergebnisse erörtert und hinsichtlich der Fragestellung interpretiert.

Trainingsaufbau

Das Training erfolgt unter Verwendung des in Abschnitt 5.2 beschriebenen Datensatzes. Um ein möglichst gutes Ergebnis zu erhalten, werden verschiedene Konfigurationen des *Generators* trainiert, die in der Netztiefe, *Kernelgröße*, *Stride*, Anzahl der Filter und dem *Padding* variieren. Es werden *Generatoren* mit unterschiedlicher Komplexität trainiert. Der kleinste umfasst ca. 380 Tsd. und der größte ca. 19 Mio. Parameter.

Die untersuchten Konfigurationen der *Generatoren* sind detailliert in Tabelle 6.1 beschrieben. Die Notation der *ConvolutionalTranspose2D-Schichten* ist wie folgt aufgebaut: conv - [quadratische *Kernelgröße*] - [Anzahl der ausgegebenen *Features*] - [*Stride*] - [*padding* des Eingangswertes] - [*padding* des Ausgangwertes]. Die Ausgangsgröße aller *Generatoren* ist stets $80 \times 80 \times 1$.

Für das Training der *Generatoren* wird dieselbe Konfiguration des *Kritikers* verwendet. Das soll die Ergebnisse und den Trainingsverlauf der *Generatoren* miteinander vergleichbar machen. Um sicherzustellen, dass das Training an keinem schlecht konfigurierten *Kritiker* scheitert, werden im Vorfeld in einem kleinen Rahmen verschiedene *Kritiker* untersucht. Diese Untersuchung wird nicht näher beschrieben.

Die in der Tabelle 6.2 dargestellte Konfiguration des *Kritikers* hat sich in der Voruntersuchung der verschiedenen *Kritiker* als am besten herausgestellt. Die Konfiguration der zweidimensionalen *Convolutional-Schichten* ist wie folgt notiert: conv - [quadratische *Kernelgröße*] - [Anzahl der ausgegebenen *Features*] - [*Stride*] - [*padding* des Eingangswertes].

Als *Optimierungsalgorithmus* wird für *Kritiker* und *Generator* der *Adam* (Kingma und Ba, 2017) mit einer *learning rate* von $2e^{-4}$, β_1 von 0.5 und β_2 von 0.9 gewählt. Diese Werte stammen aus zuvor durchgeführten Versuchen, die hier nicht weiter beschrieben werden und sich als geeignet herausgestellt haben.

Das Training wird mit einer *Batchgröße* von 40 Datensätzen durchgeführt und nach 25.000 Schritten pro Konfiguration beendet. Mit jedem Trainingsschritt wird der *Loss* des *Generators* und *Kritikers* aufgezeichnet. Als Indikator für die Diversität der generierten Daten wird alle 200 Schritte der durchschnittliche geringste *euklidische Abstand* innerhalb von 1.000 generierten Daten berechnet, wie es in Abschnitt 2.6 beschrieben ist. Gleichzeitig wird der *FID-Score* anhand von 20 *Batches* ermittelt und 32 generierte Beispieldaten erzeugt. Um die entstehende Datenmenge zu reduzieren, wird der Zustand

Tabelle 6.1: Konfigurationen der *Generatoren* für die Evaluierung der FF1.

Generator-Konfigurationen				
G - 1	G - 2	G - 3	G - 4	G - 5
input (100 x 1 x 1)				
conv4-1024-1-1-0	conv4-128-1-0-0	conv4-512-1-0-0	conv4-512-1-1-0	conv5-512-1-1-0
BatchNorm2d				
ReLU				
conv5-512-2-0-0	conv4-64-1-1-0	conv4-256-1-1-0	conv5-256-2-1-0	conv5-256-1-1-0
BatchNorm2d				
ReLU				
conv5-256-2-0-0	conv4-32-2-1-0	conv4-128-2-1-0	conv5-128-2-1-0	conv5-128-1-1-0
BatchNorm2d				
ReLU				
conv5-128-2-0-1	conv4-16-2-1-0	conv4-64-2-1-0	conv5-64-2-1-0	conv5-64-2-1-0
BatchNorm2d				
ReLU				
conv5-1-2-0-1	conv4-8-2-1-0	conv4-32-2-1-0	conv5-32-2-1-1	conv5-32-1-1-0
	BatchNorm2d			
	ReLU			
	conv4-1-2-1-0	conv4-1-2-1-0	conv5-1-2-1-1	conv5-16-2-1-0
				BatchNorm2d
				ReLU
				conv5-1-2-1-1
Tanh				
18.8 Mio. Parameter	379.5 Tsd. Parameter	2.8 Mio. Parameter	5.2 Mio Parameter	5.6 Mio Parameter

des *Generators* nur alle 600 Schritte gespeichert. Die *Mel-Spektrogramme* werden mit den in Abschnitt 5.1.1 beschriebenen Parametern erzeugt.

Alle hier aufgeführten Konfigurationen und Trainingsparameter erheben keinen Anspruch das Optimum darzustellen. Sie dienen zur Beantwortung der FF1, in der ein Ergebnis ausreicht, in dem die generierten Daten erkennbare Ähnlichkeiten zum Trainingsdatensatz aufweisen. Für die Suche nach einer optimalen Konfiguration sind umfangreiche Untersuchungen notwendig, die im Rahmen dieser Arbeit nicht stattfinden.

Der nachfolgende Abschnitt zeigt den Trainingsverlauf der einzelnen Konfigurationen und beantwortet anschließend anhand der generierten Daten die FF1.

Tabelle 6.2: Konfiguration des *Kritikers* für die Evaluierung der FF1.

input (80 x 80 x 1)
conv4-64-2-1
InstanceNorm2d
LeakyReLU
conv4-128-2-1
InstanceNorm2d
LeakyReLU
conv4-256-2-1
InstanceNorm2d
LeakyReLU
conv4-512-2-1
InstanceNorm2d
LeakyReLU
conv4-1-2-0
Lineare-Aktivierung
2.8 Mio. Parameter

Trainingsverlauf

Abbildung 6.1 stellt den Verlauf des *Losses* während des Trainings der einzelnen Konfigurationen dar. Der *Loss* kann als Fehlerwert interpretiert werden. Je näher der Wert an der null ist, desto kleiner ist der Fehler. An der Abbildung wird deutlich, dass die unterschiedlichen *Generatoren* keinen signifikanten Einfluss auf den Verlauf des *Losses* des *Kritikers* haben. Dieser Wert konvergiert in jedem Trainingsdurchlauf gegen -1.

Die Verläufe des *Losses* der *Generatoren* unterscheiden sich stärker voneinander. Der *Loss* von *Generator* 1 sinkt zu keinem Zeitpunkt (abgesehen von der anfänglichen Einpendlungsphase in den ersten hundert Schritten) gegen null und fängt nach einer ersten Phase des Sinkens bei ca. 7.000 Trainingsschritten an zu steigen. Ähnlich ist der Verlauf des *Losses* von *Generator* 3, der nach einer ersten Phase des Sinkens bei ca. 3.500 Trainingsschritten anfängt zu steigen.

Der *Loss* des *Generators* 2 nähert sich nach 6.000 Trainingsschritten der null und steigt anschließend kontinuierlich.

Der *Loss* von *Generator* 4 und *Generator* 5 verläuft flacher als der der zuvor beschriebenen *Generatoren*. Der *Loss* von *Generator* 5 konvergiert gegen null. Bei *Generator* 4 ist in den letzten Schritten eine Steigung zu erkennen.

Unter Berücksichtigung der beschriebenen Verläufe und ohne weitere Informationen zu betrachten, liegt die Vermutung nahe, dass *Generator* 5 aufgrund des flachen Verlaufes und des stetigen Annäherns gegen null die besten Ergebnisse erzeugt. Darauf folgt *Generator* 4 und *Generator* 2, weil dieser zu einem bestimmten Zeitpunkt einen *Loss* nahe der null hat. Die *Generatoren* 1 und 3 weisen, gemessen am *Loss*, kein gutes Lernverhalten auf.

Zieht man zusätzlich den Verlauf des durchschnittlichen geringsten *euklidischen Abstands* innerhalb der generierten Daten hinzu, welches ein Maß für die Diversität der Daten darstellt, wird ein positives Lernverhalten aller Konfigurationen deutlich (vgl. Abbildung 6.2). Ein steigender Wert signalisiert nämlich eine steigende Diversität der Daten. Der errechnete Referenzwert des Trainingsdatensatzes liegt bei 6,4, dem sich - ausgenommen des *Generator* 1 - alle deutlich annähern. Zwischen den einzelnen *Generatoren* 2, 3, 4 und 5 ist kein deutlicher Unterschied erkennbar.

Auffällig ist, dass sich der kontinuierlich steigende *Loss* (und damit der Fehlerwert) der *Generatoren* 2 und 3 nicht in einem sinkenden durchschnittlichen geringsten *euklidischen Abstand* widerspiegelt und somit die Datendiversität konstant bleibt.

Abbildung 6.3 stellt den Verlauf des *FID-Scores* der einzelnen Trainingsdurchläufe dar. Der graue Balken gibt den durchschnittlichen *FID-Score* der Daten aus dem Trainingsdatensatz an. Ähnlich wie bei dem Verlauf des *euklidischen Abstands* konvergiert der *FID-Score* der *Generatoren* gegen den durchschnittlichen *FID-Score* aus dem Trainingsdatensatz. Die starken Unterschiede hinsichtlich des Verlaufs der *Loss-Werte* spiegeln sich auch im *FID-Score* nicht wider. Der kontinuierlich steigende *Loss* von Generator 2 und 3 ist hier nicht erkennbar. *Generator 1* schneidet deutlich schlechter ab als die anderen *Generatoren*.

Unter der Betrachtung der oben aufgeführten Verläufe des *Losses*, *euklidischen Abstands* und *FID-Scores* lässt sich erkennen, dass die *Generatoren* ein Lernverhalten aufweisen. Der folgende Abschnitt präsentiert die Qualität der generierten Daten anhand von Auszügen und beantwortet anhand eines optischen Vergleichs die FF1.

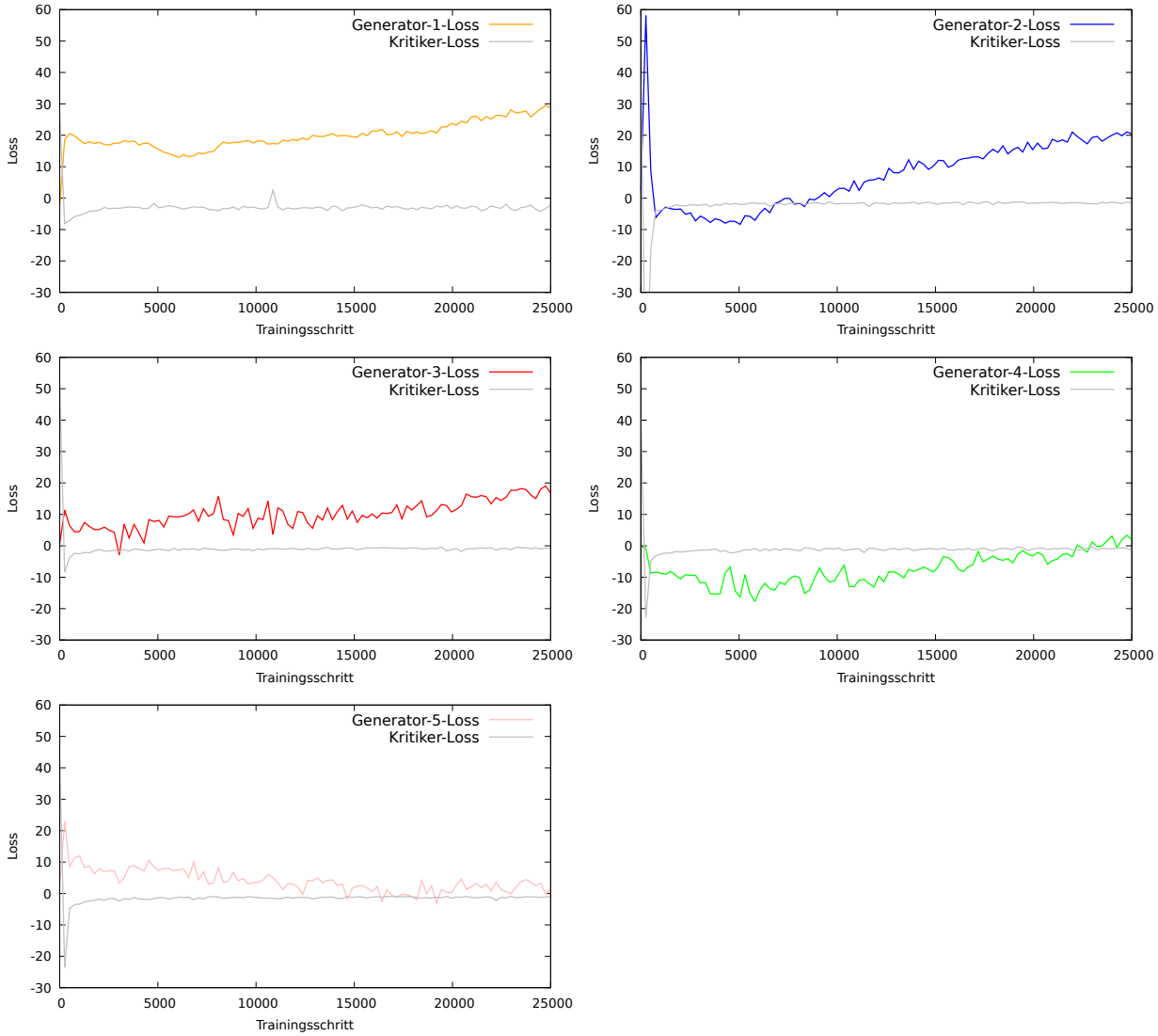


Abbildung 6.1: Verlauf der *Loss-Werte* beim Training im Rahmen der FF1.

Die Implementierung des hier durchgeführten Trainings ist im *Repository*[1] hinterlegt. Das Training einer Konfiguration beträgt mit eine *NVIDIA GTX1080* im Durschnitt 8,5 Stunden.

[1] https://github.com/batonfabi/master_thesis/blob/submission/kapitel_6/ff1/training_wgans.ipynb

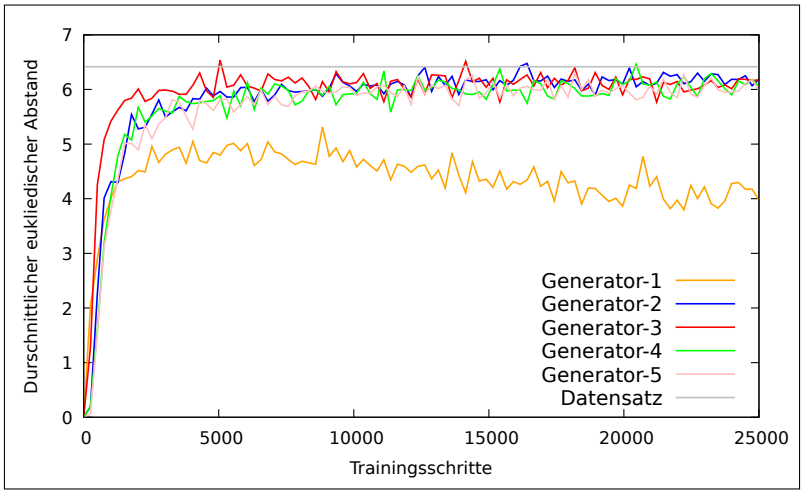


Abbildung 6.2: Durchschnittlicher geringster *euklidischer Abstand* des Trainings der *Generatoren* im Rahmen der FF1.

Ein steigender Wert deutet auf eine höhere Diversität der generierten Daten hin. Ausgenommen von *Generator* 1 nähern sich alle Konfigurationen dem Wert des Trainingsdatensatzes sehr nahe an.

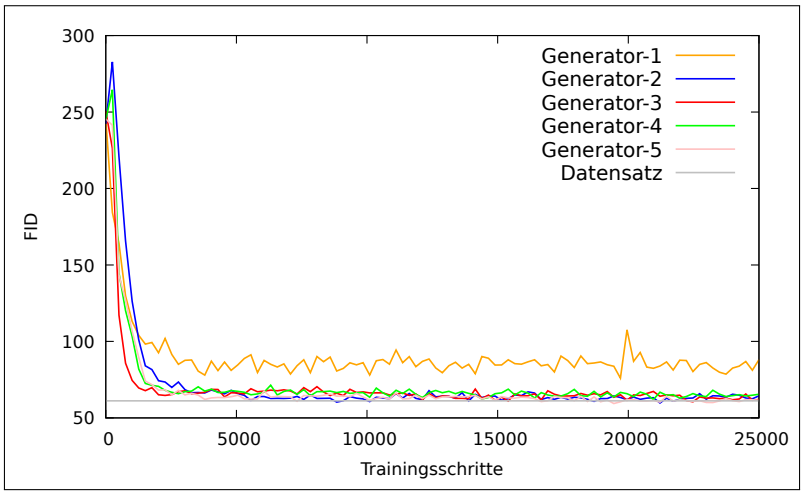


Abbildung 6.3: Verlauf des *FID-Scores* während des Trainings der *Generatoren* im Rahmen der FF1.

Der *FID-Score* weist Ähnlichkeiten zum oben beschriebenen durchschnittlichen geringsten *euklidischen Abstand* auf. Die *Generatoren* nähern sich an den *FID-Score* des Datensatzes an, woraus sich ein Positives Lernverhalten schließen lässt.

Ergebnisse der generierten Daten

Bevor die generierten Daten vorgestellt werden, wird die Leserin bzw. der Leser gebeten, sich in die Rolle eines *Diskriminators* zu versetzen und die Beispiele aus Abbildung 6.4

zu begutachten. Sind Sie in der Lage zu erkennen, welche der Beispielbilder generiert sind und welche aus dem Trainingsdatensatz stammen?

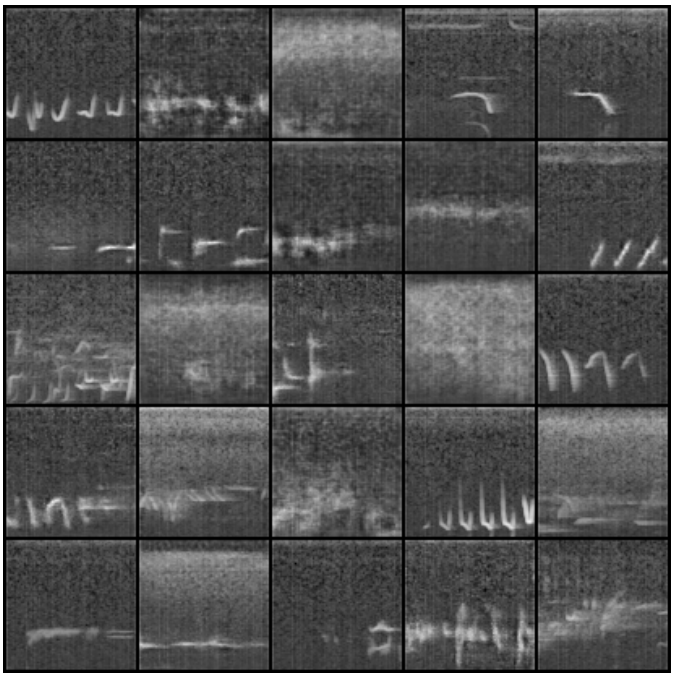


Abbildung 6.4: Exemplarische Darstellung von generierten und originalen *Mel-Spektrogrammen.*

Können Sie auseinanderhalten, welche der Bilder generiert und welche aus dem Trainingsdatensatz stammen? Die Auflösung ist in Abbildung 6.5 dargestellt.

Die Auflösung ist in Abbildung 6.5 dargestellt. Für die Erzeugung der generierten Daten kommt der *Generator* 4 zum Einsatz, der unterschiedlich lange trainiert wird. Die mit einer grünen eins markierten Bilder werden nach 1.200 Trainingsschritten und die mit einer weißen zwei markierten Bilder nach ca. 21.000 Trainingsschritten erzeugt. Die nicht markierten Beispiele stammen aus dem Trainingsdatensatz. Konnten Sie alle Exemplare richtig einordnen?

Anhand des Beispiels wird deutlich, dass der *Generator* 4 mit zunehmender Trainingsdauer bessere Ergebnisse liefert. Abbildung 6.6 zeigt exemplarisch die qualitative Entwicklung der generierten Bilder mit steigender Trainingsdauer. Das oberste Bild entspricht der generierten Qualität nach 600, das mittlere Bild nach 2.000 und das untere Bild nach 22.000 Trainingsschritten - das entspricht der Stelle, wo der *Loss* des *Generators* 4 am

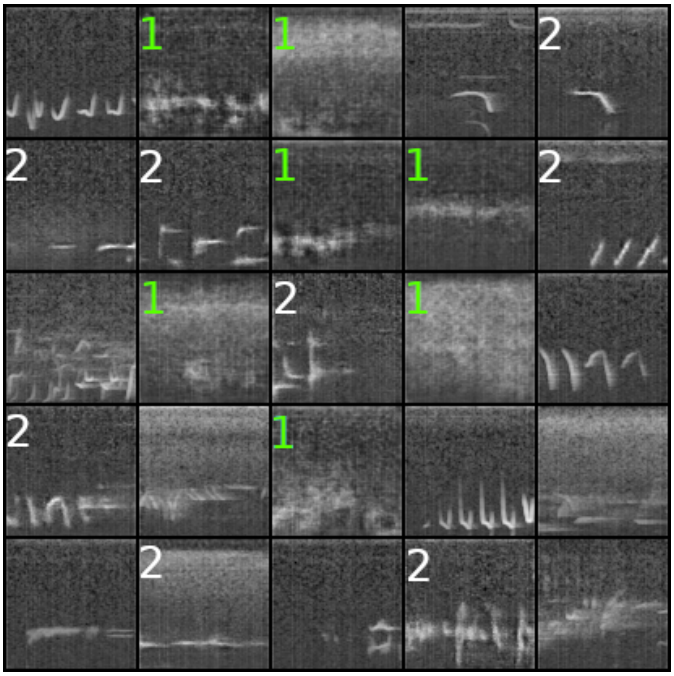


Abbildung 6.5: Auflösung der in Abbildung 6.4 dargestellten Bilder.

Die mit einer grünen eins markierten Bilder stammen von dem *Generator* 4 nach 1.200 Trainingsschritten. Die mit einer weißen zwei markierten Bilder stammen von demselben *Generator* nach ca. 21.000 Trainingsschritten. Die Bilder ohne Markierung stammen aus dem Trainingsdatensatz.

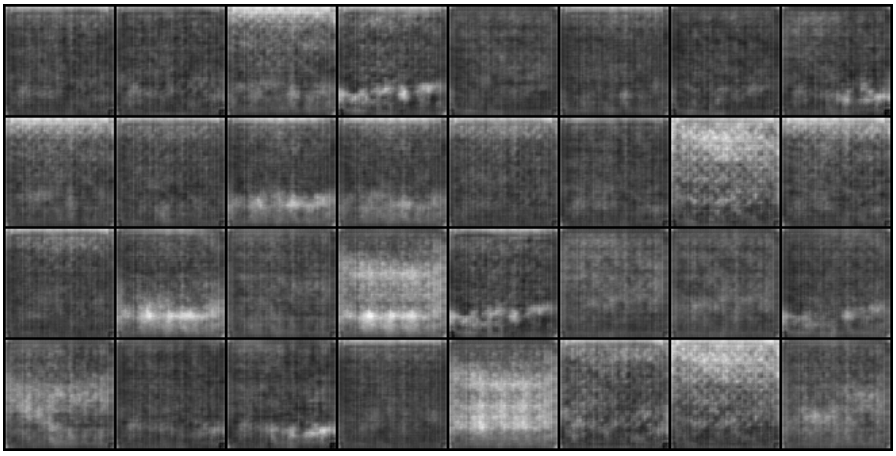
nächsten der null ist. Zum Vergleich wird in Abbildung 6.7 ein Auszug mit 32 Beispielen aus dem Trainingsdatensatz gezeigt.

Für die Vollständigkeit sind in Unterabschnitt A.3.1 generierte Beispieldaten der *Generatoren* 1, 2, 3 und 5 dargestellt, die zu den Zeitpunkten erzeugt wurden, an denen ihre *Loss-Werte* am nächsten der null liegen. Anhand der Beispiele wird deutlich, dass alle *Generatoren* in der Lage sind Daten zu generieren, die erkennbare Ähnlichkeiten mit dem Trainingsdatensatz aufweisen.

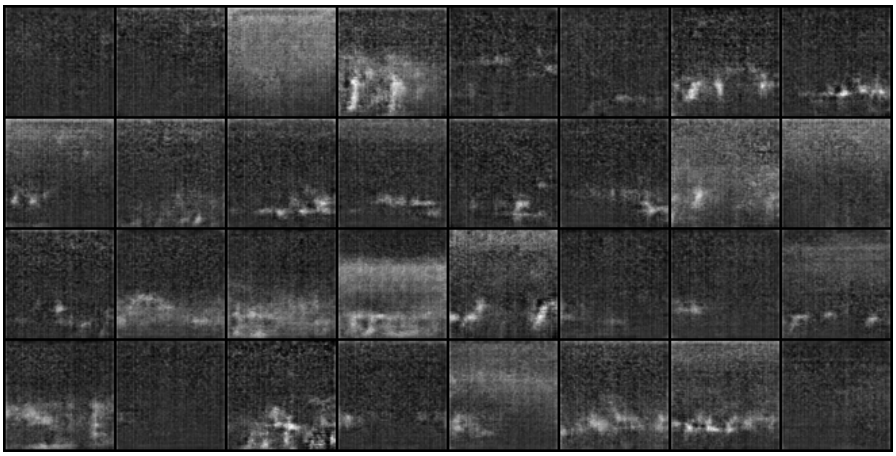
Auf Grundlage der hier dargestellten Ergebnisse wird die Forschungsfrage, ob *WGANs* in der Lage sind Vogelgesang in Form von *Mel-Spektrogrammen* abzubilden, mit einem Ja beantwortet. Eine genauere Analyse der Qualität erfolgt mit der Beantwortung der FF4 und im Rahmen der qualitativen Analyse in Abschnitt 6.2.

Der folgende Abschnitt beschäftigt sich mit der zweiten Forschungsfrage und prüft anhand eines Versuchs, ob der vortrainierte *Vocoder Mel-Spektrogramme* von Vogelgesang in entsprechende Signale in *Wellenform* umwandeln kann.

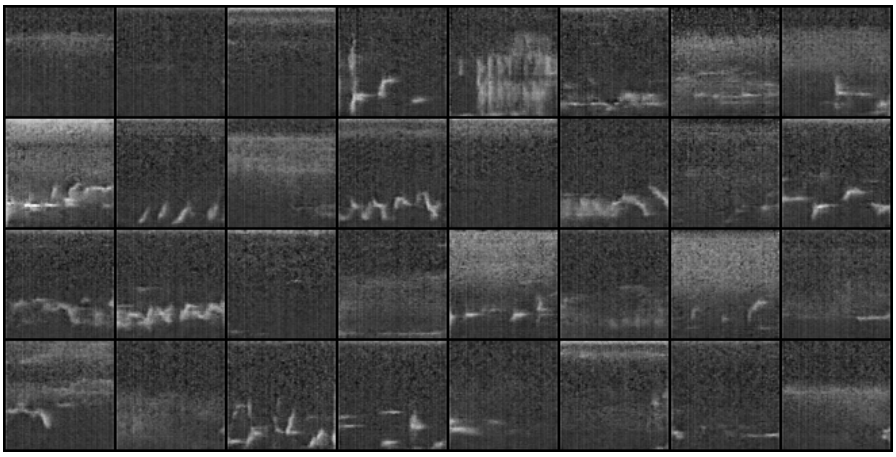
Fazit Der direkte Vergleich der generierten Daten der trainierten *Modelle* zeigt erkennbare Ähnlichkeiten mit den Daten aus dem Trainingsdatensatz auf, sodass die Forschungsfrage, ob *WGANs* in der Lage sind, Vogelgesang in Form von *Mel-Spektrogrammen* abzubilden, mit Ja beantwortet wird.



Generierte Beispiele nach 600 Trainingsschritten.



Generierte Beispiele nach ca. 2.000 Trainingsschritten.



Generierte Beispiele nach ca. 22.000 Trainingsschritten.

Abbildung 6.6: Verlauf der Qualitätssteigerung der von *Generator* 4 trainierten Daten mit steigenden Trainingsschritten.

Anhand der hier aufgeführten Beispiele wird deutlich, dass der *Generator* 4 mit steigender Zahl der Trainingsschritte zunehmend bessere Ergebnisse liefert.

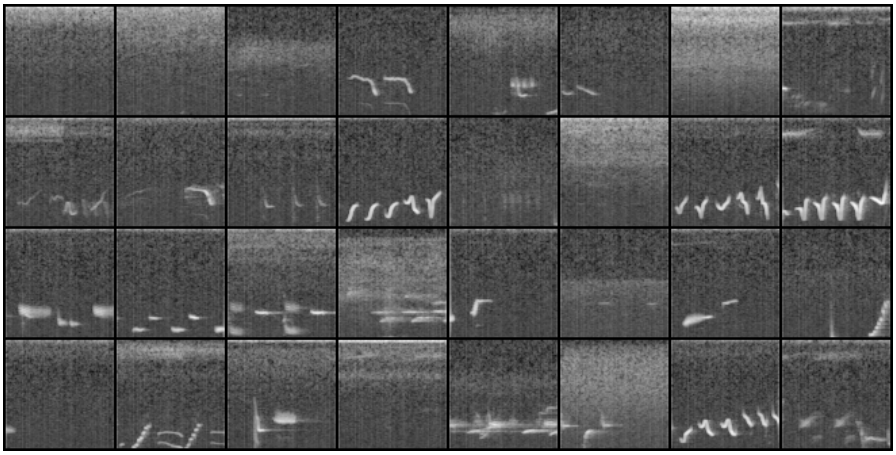


Abbildung 6.7: Auszug von 32 Beispieldaten aus dem Trainingsdatensatz.

Der direkte Vergleich der Daten aus dem Trainingsdatensatz mit den Beispielen aus Abbildung 6.6 weist erkennbare Ähnlichkeiten auf. *WGANs* sind somit in der Lage, ohne eine Aussage über die Qualität zu treffen, *Mel-Spektrogramme* von Vogelgesang abzubilden.

6.1.2 Eignung des vortrainierten Vocoders zur Synthese von Vogelgesang (FF2)

Frage: Ist ein vortrainierter *Vocoder*, der für die Sprachsynthese entwickelt ist, für die Umwandlung von *Mel-Spektrogrammen* mit Vogelgesang in *Wellenform* ohne signifikantem Qualitätsverlust geeignet?

Für die Evaluierung dieser Forschungsfrage werden manuell *Mel-Spektrogramme* aus Beispielen des Trainingsdatensatzes generiert und mithilfe des vortrainierten *Vocoders* (NVIDIA, 2020) zurück in Audiosignale umgewandelt. Anschließend wird vom Autor das rekonstruierte Audiosignal mit dem originalen Audiosignal verglichen und in die folgende fünfstufige Skala eingeordnet:

1. Der Vogelgesang im rekonstruierten Audiosignal ist klarer zu erkennen als im Original, z.B. wenn das Rauschen im Hintergrund leiser ist.
2. Der Vogelgesang im rekonstruierten Audiosignal ist identisch mit dem Vogelgesang der originalen Aufzeichnung. Der Autor empfindet keinen Qualitätsunterschied.
3. Das rekonstruierte Audiosignal unterscheidet sich leicht vom Original, z. B. wenn die Lautstärke des rekonstruierten Audiosignals von dem Original abweicht oder Hintergrundgeräusche anders wahrgenommen werden. Die in diese Kategorie eingeordneten Beispiele weisen keine signifikanten Unterschiede im Vogelgesang auf.
4. Der Vogelgesang im rekonstruierten Audiosignal ist erkennbar, weist aber negative Qualitätsunterschiede auf.
5. Der Vogelgesang im rekonstruierten Audiosignal ist nicht zu erkennen.

Dieser Versuch wird, um gleichzeitig die in Unterabschnitt 5.1.2 beschriebene Plausibilitätsprüfung durchzuführen, leicht erweitert. Die manuell erzeugten *Mel-Spektrogramme* werden zusätzlich von dem *Skalierer* in den Bereich von [-1; 1] normalisiert und anschließend wieder in die ursprüngliche Form zurückgewandelt. Das Ergebnis wird anschließend vom *Vocoder* genutzt, um das Signal in *Wellenform* zu erzeugen.

Um eine höhere Signaldiversität abzudecken, werden in diesem Versuch mehr Vogelarten verwendet, als die zwei in Abschnitt 5.2 aufgeführten Arten. Es werden sechs Beispiele der folgenden Arten betrachtet: Schwarzspecht, Teichrohrsänger, Grünfink, Buntspecht, Kohlmeise, Fichtenkreuzschnabel, Schilfrohrsänger, Singdrossel und der Fitis. Dieser Datensatz wurde zufällig ausgesucht. Die Länge der Beispieldaten beträgt ca. 1 Sekunde.

Die *Mel-Spektrogramme* werden mit den in Abschnitt 5.1.1 beschriebenen Parameter berechnet.

Die Tabelle 6.3 fasst das Ergebnis der 54 Beispiele zusammen, die in dem zu der vorliegenden Arbeit zugehörigen Repository[2] nachgehört werden können.

Tabelle 6.3: Ergebnis der Bewertung der rekonstruierten Signale.

Bewertungskategorie	1.	2.	3.	4.	5.
Anzahl Beispiele	7	26	20	1	0

Von den 54 Beispieldaten fällt nur eines in die vierte Kategorie, die angibt, dass der Vogelgesang im rekonstruierten Signal unter mangelnder Qualität leidet. Dieser Ausreißer kann vernachlässigt werden, sodass die *Forschungsfrage 2* mit einem Ja beantwortet wird.

Um der Leserin bzw. dem Leser ein bildliches Verständnis davon zu liefern, wie die Signale sich voneinander unterscheiden, wird im Folgenden ein Vertreter jeder identifizierten Kategorie (links Kategorie eins bis rechts Kategorie vier) als *Mel-Sepektrogramm* aufgeführt. Oben sind die Originaldaten und unten die jeweiligen Rekonstruktionen dargestellt.

Nachfolgend werden die Ergebnisse generierter Daten im Zusammenspiel der beiden Komponenten *WGAN* und *Vocoder* untersucht.

Fazit: Die manuelle Untersuchung hat ergeben, dass der auf Sprachsynthese vortrainierte *Vocoder* (NVIDIA, 2020) sich für das Generieren von Vogelgesang eignet.

6.1.3 Verifizierung der Architektur (FF3)

Frage: Ermöglicht das Zusammenspiel von *WGAN* und *Vocoder* die Generierung von Vogelgesang in Wellenform?

Dieser Abschnitt beschäftigt sich mit dem generierten Vogelgesang, der mit der zusammengesetzten Architektur der Komponenten *WGAN*, *Skalierer* und *Vocoder* erzeugt wird. Für die Evaluierung der Forschungsfrage werden Audiosignale generiert und von dem Autor dahingehend evaluiert, ob die generierten Signale nach Vogelgesang klingen.

[2] https://github.com/batonfabi/master_thesis/blob/submission/kapitel_6/ff2/ff2.ipynb

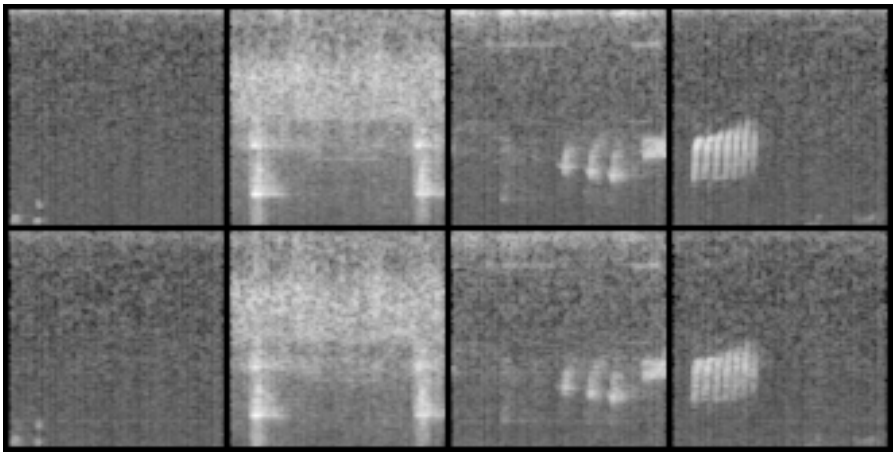


Abbildung 6.8: Bildliche Darstellung von Beispielen der vier identifizierten Kategorien zur Bewertung des rekonstruierten Audiosignals.

Bildliche Darstellung von Beispielen von identifizierten Vertretern der oben beschriebenen Kategorien eins bis vier. Die obere Zeile beinhaltet die originalen Beispiele und die untere Zeile die entsprechenden Rekonstruktionen. Links befindet sich die erste Kategorie und rechts die vierte. Mit dem bloßen Auge erkennt man nur leichte Unterschiede. Das letzte Bild unten rechts weist im Gegensatz zum Original leichte Verschwommenheit auf.

Als *WGAN-Modell* wird die in Abschnitt 6.1 beschriebene Konfiguration 5 mit ca. 21.000 Trainingsschritten verwendet. Die generierten Audiosignale können im *Jupyter Notebook*[3] oder als gespeicherte Audiosignale[4] angehört werden. Darüberhinaus bietet das *Jupiter Notebook* im letzten *Codeblock* die Möglichkeit, eigene Beispiele zu generieren.

Die generierten Beispiele beinhalten deutlich hörbare Ähnlichkeiten mit dem Vogelgesang aus dem Trainingsdatensatz. Die *Forschungsfrage 3* wird mit *Ja* beantwortet. Eine genauere Analyse der erzielten Qualität ist nicht Teil dieses Abschnittes und erfolgt in Abschnitt 6.2.

Nachfolgend wird der Frage nachgegangen, ob der *FID-Score* sich für die Beurteilung der Qualität des Vogelgesangs eignet.

Fazit: Die im *Jupyter Notebook* aufgeführten Ergebnisse zeigen deutlich, dass Vogelgesang mit der in Abschnitt 5.1 vorgestellten Architektur erfolgreich generiert wird.

[3] https://github.com/batonfabi/master_thesis/blob/submission/kapitel_6/ff3/wgan_with_vocoder.ipynb
[4] https://github.com/batonfabi/master_thesis/tree/submission/kapitel_6/ff3/examples

6.1.4 Eignung des FID-Scores zur Evaluation von generiertem Vogelgesang (FF4)

Frage: Ist der *FID-Score* im Kontext der Synthese von Vogelgesang über *Mel-Spektrogramme* ein geeigneter Indikator für ihre Qualität (FF4)?

Dieser Abschnitt untersucht die Aussagekraft des *FID-Scores* zur Bewertung der *Mel-Spektrogramme* von Vogelgesang. Dazu werden im ersten Schritt die in Abschnitt 6.1 erzielten Ergebnisse genauer untersucht und die generierten Daten mit dem Verlauf des *FID-Scores* verglichen. Anschließend werden die zwischengespeicherten Zustände der Konfigurationen während des Trainings aus Abschnitt 6.1 genutzt, um die *FID-Scores* genauer zu berechnen. Auf Grundlage der genaueren Ergebnisse wird im nächsten Schritt geprüft, ob sich unterschiedliche *FID-Scores* in der Qualität der generierten Daten widerspiegeln.

Die Abbildung 6.9 stellt den Verlauf des *FID-Scores* aus Abbildung 6.3 vergrößert dar. Hier ist erkennbar, dass die *Generatoren* 2, 3 und 5 zwischen dem 20.000 und 25.000 Trainingsschritt den durchschnittlichen *FID-Score* des Trainingsdatensatzes an einigen Stellen unterschreiten. Unter der Annahme, dass der *FID-Score* ein gutes Maß zur Bewertung der generierten *Mel-Spektrogramme* darstellt, müssten die Ergebnisse der *Generatoren* an diesen Stellen äquivalent zu den Daten aus dem Trainingsdatensatz sein.

Der direkte Vergleich der zu diesem Zeitpunkt generierten *Mel-Spektogramme* des *Generators* 5 in Abbildung 6.10 mit Beispieldaten aus dem Trainingsdatensatz in Abbildung 6.11 kann diese Annahme nicht bestätigen. In den generierten Daten finden sich beispielsweise keine sauberen *Bögen* und *Rundungen*, wie sie in den ersten zwei Beispielen aus Abbildung 6.11 zu erkennen sind.

Die in Abbildung 6.9 dargestellten *FID-Scores* werden mit *Batchgrößen* von 40 Elementen berechnet. Um sicherzustellen, dass die oben beschriebene Analyse aufgrund ungenauer Berechnungen zu keinem falschen Ergebnis führt, werden die zwischengespeicherten Zustände der *Generatoren* aus Abschnitt 6.1 verwendet, um ihren *FID-Score* genauer zu berechnen. Die Abbildung 6.12 stellt den Verlauf des *FID-Scores* unter Verwendung einer *Batchgröße* mit 400 Elementen dar. Hier wird der Durchschnitt aus je 20 Berechnungen ermittelt.

Der in Abbildung 6.12 dargestellte Verlauf des *FID-Scores* stimmt besser mit dem zuvor beschriebenen optischen Vergleich des Ergebnisses von *Generator* 5 mit dem Trainings-

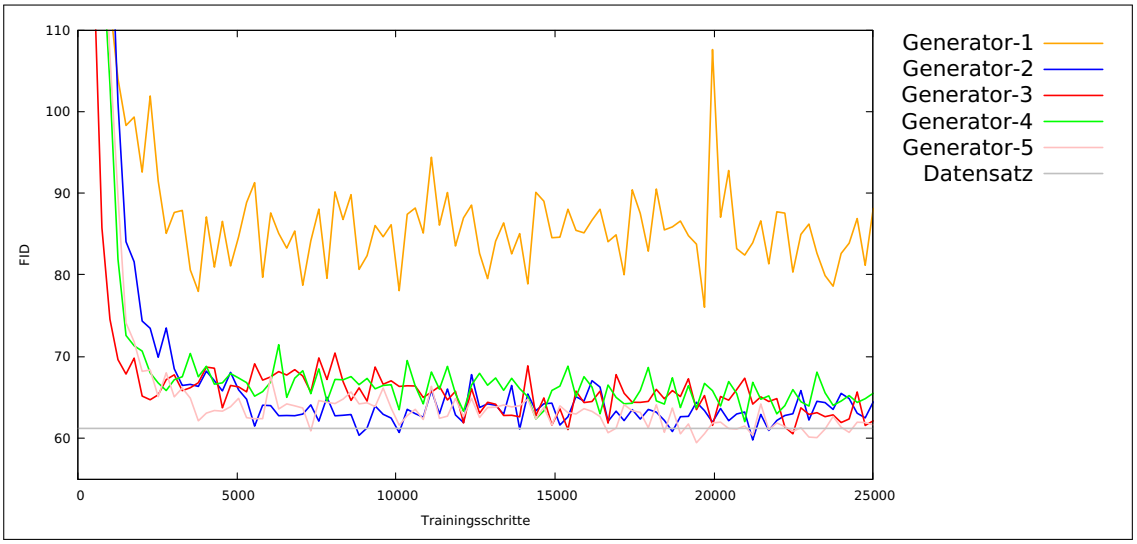


Abbildung 6.9: Vergrößerter Verlauf des *FID-Scores* von dem Training aus Abschnitt 6.1.

Die *Generatoren* 2, 3 und 5 unterschreiten zwischen den Trainingsschritten 20.000 und 25.000 den durchschnittlichen *FID-Score* des Trainingsdatensatzes. Ein Vergleich der generierten Daten des *Generators* 5 zu diesem Zeitpunkt weist dennoch Qualitätsunterschiede auf (vgl. Abbildung 6.10 und Abbildung 6.11).

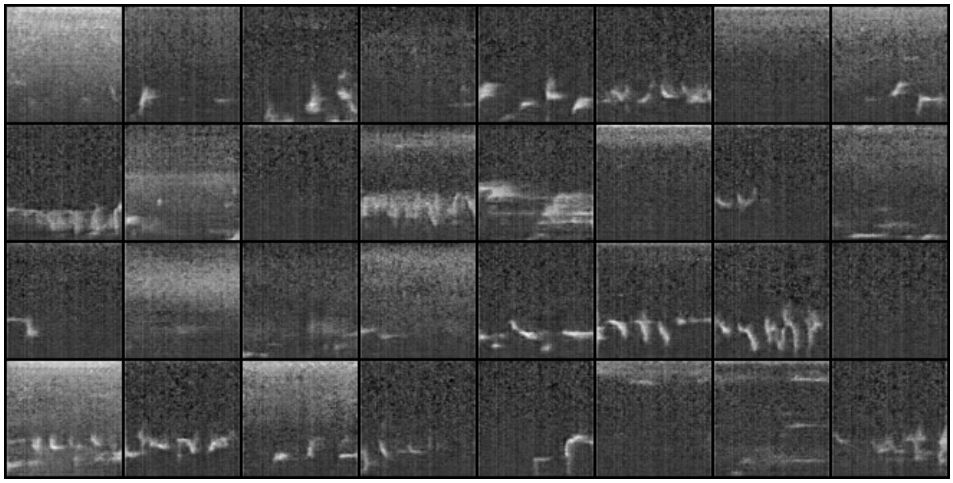


Abbildung 6.10: Beispiele von generierten Daten des *Generators* 5 aus Abschnitt 6.1 nach 23.097 Trainingsschritten.

Der errechnete *FID-Score* liegt zu diesem Zeitpunkt unter dem Durchschnitt des Trainingsdatensatzes. Unter Annahme, dass der *FID-Score* ein gutes Maß für die Qualität der *Mel-Spektrogramme* von Vogelgesang ist, müsste die Qualität der hier generierten Daten dem Trainingsdatensatz (vgl. Abbildung 6.11) gleichen.

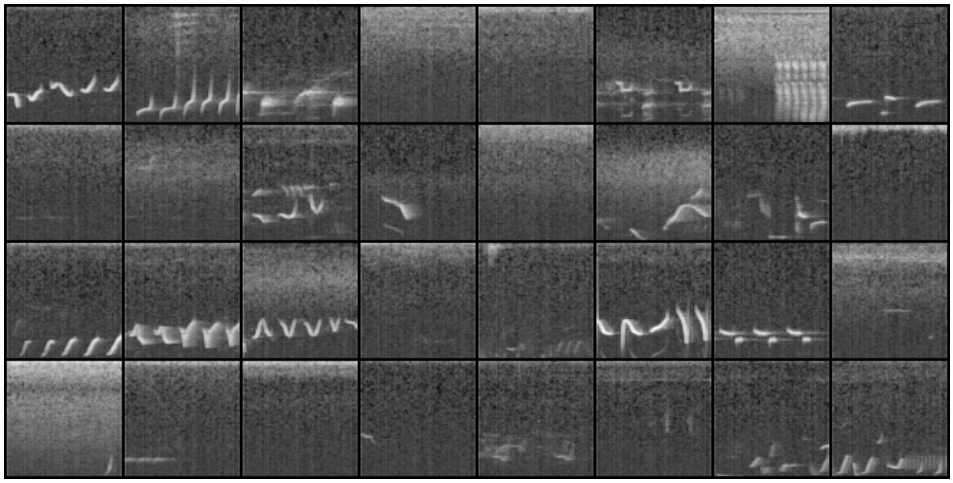


Abbildung 6.11: Beispieldaten aus dem Trainingsdatensatz.

datensatz überein als der aus Abbildung 6.9. Keine Konfiguration der *Generatoren* erreicht den durchschnittlichen *FID-Score* des Trainingsdatensatzes, was sich mit der oben aufgeführten Diskrepanz zwischen den generierten und originalen *Mel-Spektrogrammen* deckt.

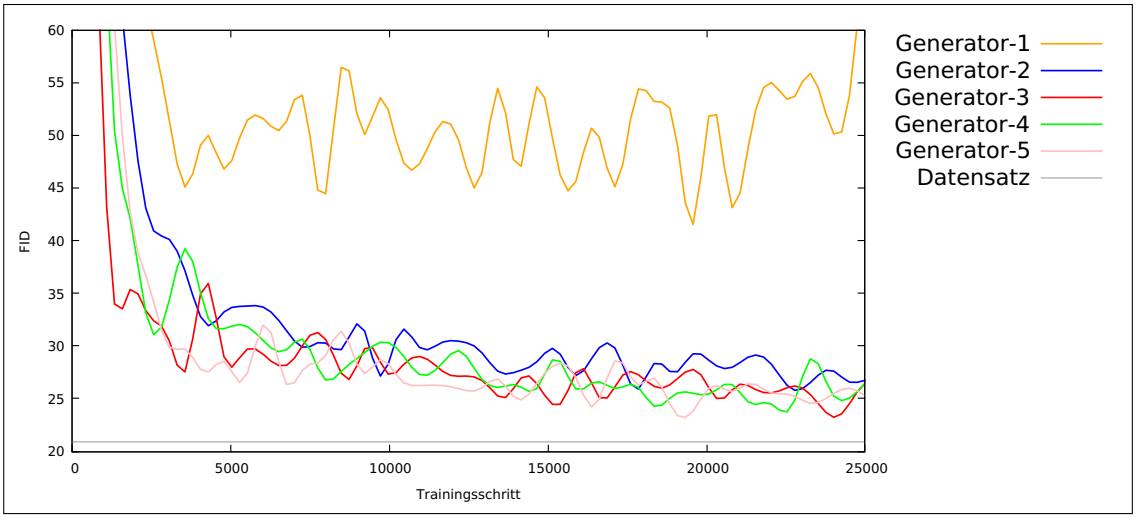


Abbildung 6.12: *FID-Score* der *Generatoren* 1 bis 5, der anhand einer *Batchgröße* von 400 Beispielen berechnet wurde.

Abschließend wird überprüft, ob die errechneten *FID-Scores* der einzelnen *Modelle* mit ihren Ergebnissen untereinander korrelieren. Dazu werden drei Zustände unterschiedlicher Konfigurationen gewählt, die einen ähnlichen *FID-Score* aufweisen und eine Konfi-

guration, dessen *FID-Score* deutlich höher ist. Die Erwartung ist, dass die *Modelle* mit ähnlichem *FID-Score* ähnliche Ergebnisse liefern, wohingegen das *Modell* mit höherem *FID-Score* schlechtere Ergebnisse erzeugt.

Um diese Hypothese zu überprüfen, werden die folgenden Konfigurationen gewählt (sortiert nach absteigendem *FID-Score*):

- *Generator* 1 bei 21.373 Trainingsschritten und einem *FID-Score* von 49,89.
- *Generator* 4 bei 25.097 Trainingsschritten und einem *FID-Score* von 26,80.
- *Generator* 2 bei 24.497 Trainingsschritten und einem *FID-Score* von 26,57.
- *Generator* 3 bei 22.163 Trainingsschritten und einem *FID-Score* von 26,16.

Mit den *Generatoren* wird je ein Datensatz mit 1.000 zufälligen *Mel-Sepektrogrammen* generiert. Anschließend werden von dem Autor vier charakteristische Bilder aus dem Trainingsdatensatz entnommen und auf ihrer Grundlage die fünf *Nearest Neighbours* mithilfe der *euklidischen Distanz* aus jedem der von den *Generatoren* erzeugten Datensätze ermittelt. Es wird erwartet, dass die Qualität der generierten Daten der *Generatoren* 2, 3 und 4 untereinander deutlich ähnlicher sind als mit den Daten von *Generator* 1.

Die nachfolgenden Abbildungen 6.13, 6.14 und 6.15 und 6.16 zeigen die entsprechenden Ergebnisse. Das oberste Bild stammt aus dem Trainingsdatensatz, die darunter folgenden Reihen sind die fünf *Nearest Neighbours* (von links nach rechts) des *Generators* 1 in der ersten Zeile, *Generators* 4 in der zweiten Zeile, *Generators* 2 in der dritten Zeile und *Generators* 3 in der letzten Zeile. Diese Reihenfolge entspricht dem absteigenden *FID-Score* der *Generatoren*.

Unter Betrachtung der Ergebnisse wird festgehalten, dass *Generator* 1 schlechtere Ergebnisse liefert als die anderen *Generatoren*. Eine klare Differenzierung der Qualität zwischen *Generator* 2, 3 und 4 ist nicht immer möglich. Die Ursache könnte in den Messfehlern des *FID-Scores* liegen.

Anhand der hier vorgelegten Untersuchungen kann abschließend zusammengefasst werden, dass sich der *FID-Score* als Maß für ein positives Trainingsverhalten eignet. Darüber hinaus ist erkennbar, dass *Generatoren* mit deutlich niedrigerem *FID-Score* bessere Ergebnisse liefern als *Generatoren* mit einem höheren *FID-Score*. Eine Differenzierung zwischen *Generatoren*, deren *FID-Score* sich in der Nachkommastelle unterscheidet, ist nicht festzustellen.

Da die Berechnung des *FID-Score* rechenintensiv ist, ist eine nachträgliche Berechnung des *FID-Scores* anhand von gespeicherten Zuständen der *Generatoren* empfohlen, um so das Training zu beschleunigen.

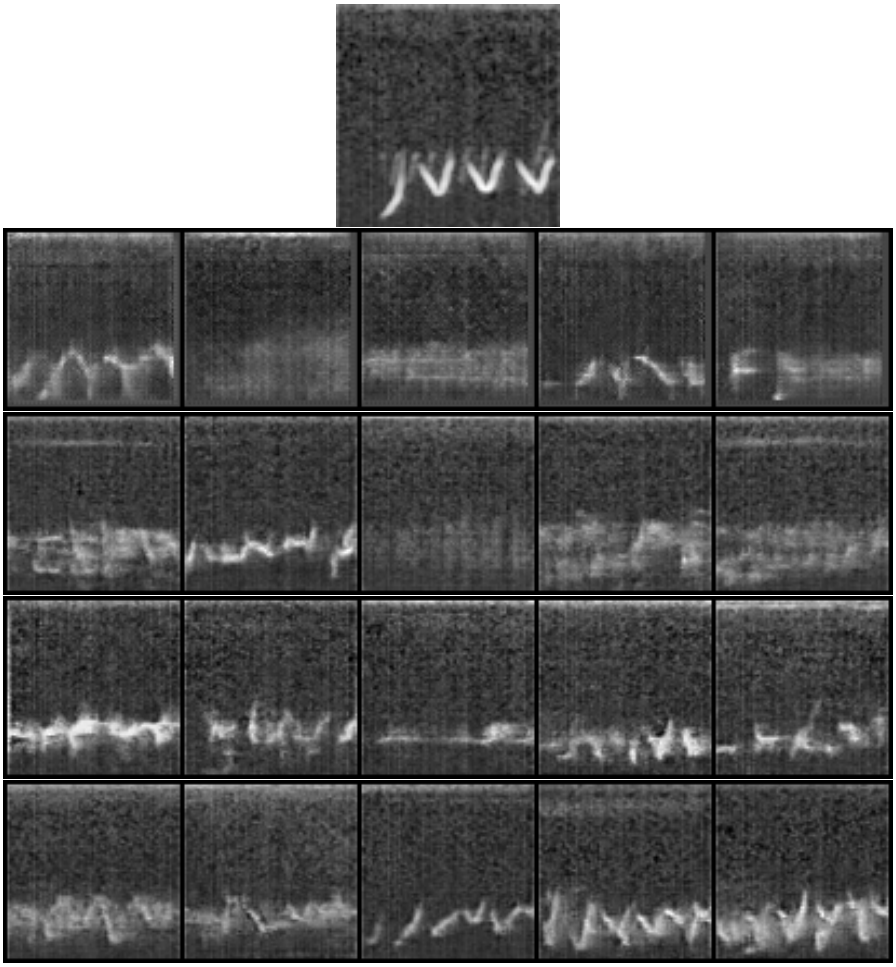


Abbildung 6.13: Beispiel 1 der *Nearest Neighbours* der oben beschriebenen *Generatoren* 1, 2, 3 und 4

Das oberste Bild stellt das gesuchte Beispiel aus dem Trainingsdatensatz dar. In jeder Reihe sind die fünf *Nearest Neighbours* der *Generatoren* dargestellt. Die erste Reihe gehört zu *Generator* 1, die zweite zu *Generator* 4, die dritte zu *Generator* 2 und die letzte zu *Generator* 3.

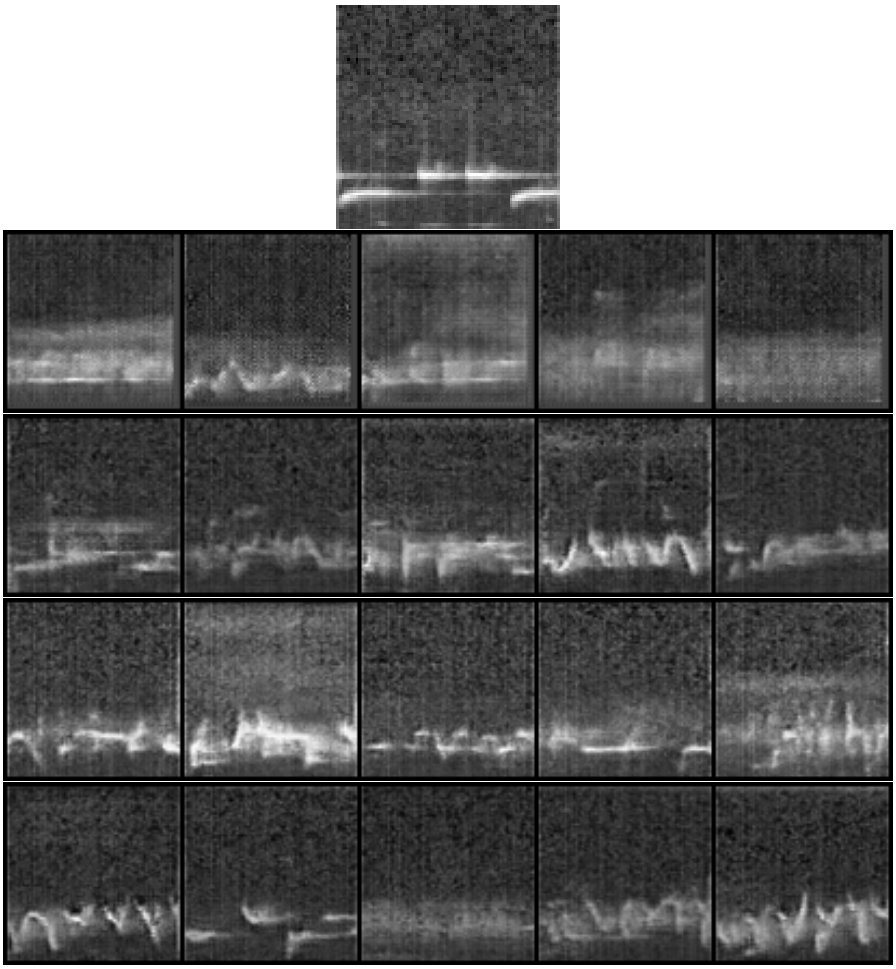


Abbildung 6.14: Beispiel 2 der *Nearest Neighbours* der oben beschriebenen *Generatoren* 1, 2, 3 und 4

Das oberste Bild stellt das gesuchte Beispiel aus dem Trainingsdatensatz dar. In jeder Reihe sind die fünf *Nearest Neighbours* der *Generatoren* dargestellt. Die erste Reihe gehört zu *Generator* 1, die zweite zu *Generator* 4, die dritte zu *Generator* 2 und die letzte zu *Generator* 3.

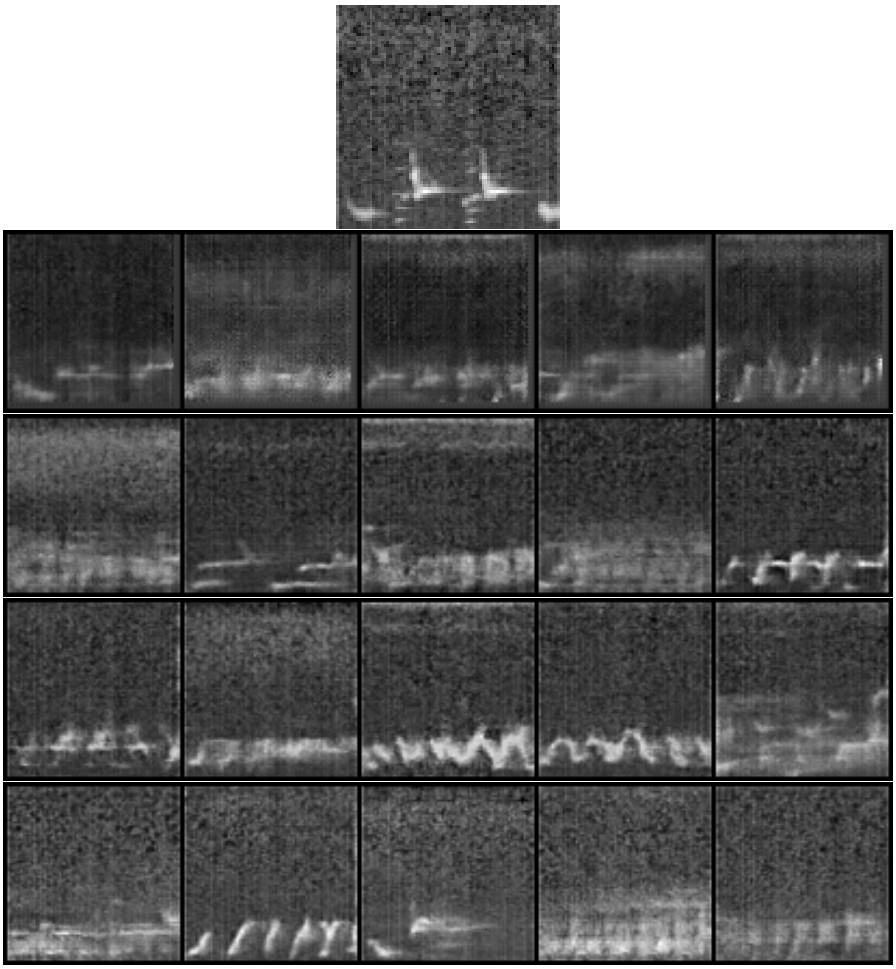


Abbildung 6.15: Beispiel 3 der *Nearest Neighbours* der oben beschriebenen *Generatoren* 1, 2, 3 und 4

Das oberste Bild stellt das gesuchte Beispiel aus dem Trainingsdatensatz dar. In jeder Reihe sind die fünf *Nearest Neighbours* der *Generatoren* dargestellt. Die erste Reihe gehört zu *Generator* 1, die zweite zu *Generator* 4, die dritte zu *Generator* 2 und die letzte zu *Generator* 3.

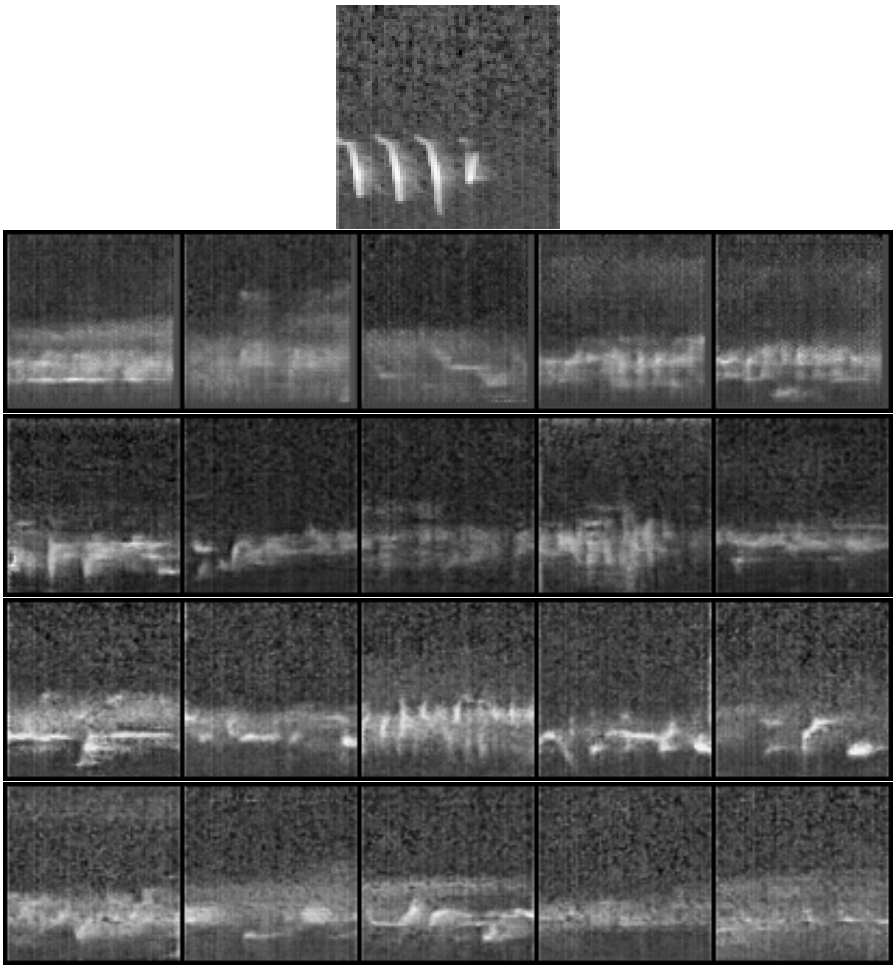


Abbildung 6.16: Beispiel 4 der *Nearest Neighbours* der oben beschriebenen *Generatoren* 1, 2, 3 und 4

Das oberste Bild stellt das gesuchte Beispiel aus dem Trainingsdatensatz dar. In jeder Reihe sind die fünf *Nearest Neighbours* der *Generatoren* dargestellt. Die erste Reihe gehört zu *Generator* 1, die zweite zu *Generator* 4, die dritte zu *Generator* 2 und die letzte zu *Generator* 3.

Fazit: Die hier aufgestellten Untersuchungen haben ergeben, dass der *FID-Score* mit einer steigenden Qualität der *Mel-Spektrogramme* korreliert. Für den Vergleich von *Modellen* mit einem sich in der Nachkommastelle unterscheidenden *FID-Score* wird eine manuelle Analyse empfohlen.

6.2 Qualitative Modell-Evaluation

Dieser Abschnitt widmet sich der qualitativen Analyse über den Vergleich von Daten aus dem Trainingsdatensatz mit ihren generierten *Nearest Neighbours*. Als Grundlage dient das *Modell* mit dem niedrigsten *FID-Score*, welches im Rahmen der FF1 trainiert wurde. Es handelt sich hierbei um den *Generator* 3, der mit ca. 24.000 Trainingsschritten den *FID-Score* von 23,24 erreicht. Die hier dargestellte Analyse umfasst im ersten Abschnitt ein *Nearest Neighbour* Vergleich von Beispielen aus dem Trainingsdatensatz mit generierten Daten.

Anschließend folgt ein Beispiel, in dem *Morphing* zwischen verschiedenen Vogelgesängen durchgeführt wird. Das ist zwar keine Methode für die Qualitative Analyse von *GANs*, es bietet aber Erkenntnisse über den Aufbau des *latent spaces*.

6.2.1 Nearest Neighbour Analyse

Für die *Nearest Neighbour* Analyse werden von beiden Vogelarten des Trainingsdatensatzes - dem Fitis und der Kohlmeise - je sechs charakteristische Muster in den *Mel-Spektrogrammen* ihres Gesangs identifiziert. Zusätzlich werden vier charakteristische Muster eines Zaunkönigs genommen, dessen Gesang sich in einer ähnlichen Frequenz befindet wie die Daten aus dem Trainingsdatensatz. Die charakteristischen Muster werden subjektiv vom Autor ermittelt.

Mithilfe des oben genannten *Modells* wird eine Datenmenge mit 10.000 zufälligen Beispielen erzeugt. Anschließend werden für jedes der identifizierten Muster fünf *Nearest Neighbours* innerhalb dieser generierten Datenmenge ermittelt.

Die Abbildungen 6.17, 6.18, 6.19 zeigen die entsprechenden Ergebnisse. Links des grünen Streifens befinden sich die *Mel-Spektrogramme* der originalen Datensätze, rechts davon sind die fünf *Nearest Neighbours* aus dem Datensatz der 10.000 vom *Generator* zufällig generierten Daten.

Abbildung 6.17 fasst die Ergebnisse der Kohlmeise und Abbildung 6.18 des Fitis zusammen. Abbildung 6.19 zeigt Muster vom Zaunkönig, welcher nicht Teil des Trainingsdatensatzes ist. An der Beobachtung der *Nearest Neighbours* der generierten Daten lassen sich die folgenden Feststellungen machen:

- Das *Modell* ist in der Lage weniger komplexe Muster wie das erste oder fünfte Beispiel aus Abbildung 6.17 so gut zu imitieren, dass eine Unterscheidung zwischen generierten und originalen Daten mit bloßem Auge nicht möglich ist. Komplexe Muster, wie Beispiel drei der Abbildung 6.17, werden deutlich schlechter synthetisiert. Das kann einerseits an einem nicht optimal konfigurierten *Modell* liegen. Andererseits kann eine Unterrepräsentation dieser Muster innerhalb des Trainingsdatensatzes zu diesem Ergebnis führen. Um diese Frage abschließend zu beantworten, sind weitere Untersuchungen nötig, die nicht Teil der vorliegenden Arbeit sind.
- Die Quantität der identifizierten *Nearest Neighbours* hat keine Anzeichen auf ein *Mode Collapse* des *Modells* - also den Zustand, in dem das *Modell* eine geringe Anzahl von unterschiedlichen Daten generiert. Jedes der ermittelten *Nearest Neighbours* unterscheidet sich voneinander - abgesehen derjenigen, die kein Signal aufweisen.
- Muster, die nicht Teil des Datensatzes sind, lassen sich in dem vorliegenden Trainingskontext nicht wiederfinden. Ein größerer Datensatz mit diverseren Daten könnte zu einem anderen Ergebnis führen.

In dem *Repository*[5] ist für jedes der hier aufgeführten Beispiele eine Audiodatei hinterlegt. Die Beispiele sind nach dem folgenden Format abgelegt, deren Nummerierung mit einer null beginnt.

- (Vogelname)-original-(Beispielnummer).wav für das gesuchte Muster
- (Vogelname)-(Beispielnummer)-nn-(n).wav für den n-ten *Nearest Neighbour*

Beispielsweise sind die Audiodateien für das Original und den dritten *Nearest Neighbour* der dritten Reihe aus Abbildung 6.18 unter fitis-original-2.wav und fitis-2-nn-2.wav zu finden.

[5] https://github.com/batonfabi/master_thesis/tree/submission/kapitel_6/explore_model/nn_examples

Zusätzlich dazu befindet sich im *Repository* ein *Jupyter Notebook*[6], in dem dieser Versuch nachgestellt ist. Hier kann eine neue Datenmenge aus 10.000 Beispielen generiert und die *Nearest Neighbours* zu den hier aufgeführten Beispielen innerhalb dieser Datenmenge abgespielt werden.

Der akustische Vergleich der Originaldaten mit den generierten Daten zeigt, dass die generierten Daten dem Original an einigen Stellen stärker ähneln, als es der visuelle Vergleich vermuten lässt. Ein Beispiel dafür findet sich in den Audiodaten der Reihe drei aus Abbildung 6.18.

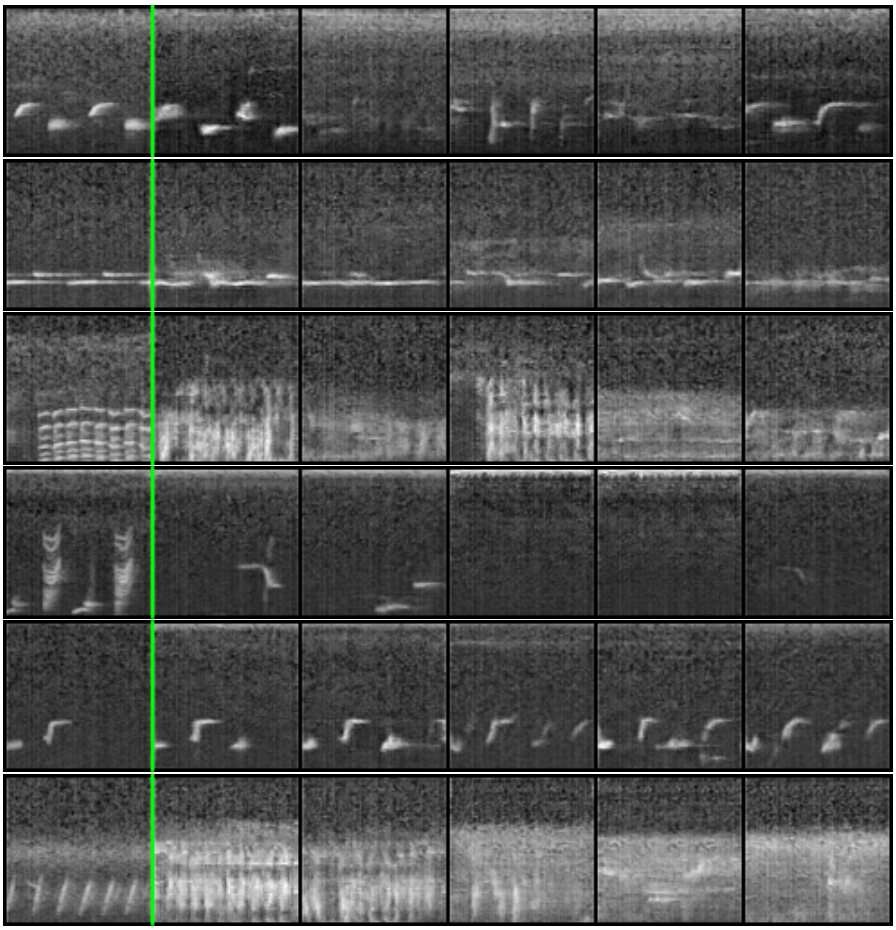


Abbildung 6.17: *Nearest Neighbours* von sechs charakteristischen Mustern der Kohlmeise.

Links vom grünen Trenner ist das Original aus dem Datensatz abgebildet. Rechts davon befinden sich die fünf *Nearest Neighbours* der vom *Generator* erzeugten Daten.

[6] https://github.com/batonfabi/master_thesis/blob/submission/kapitel_6/explore_model/nearest_neighbours_and_morphing.ipynb

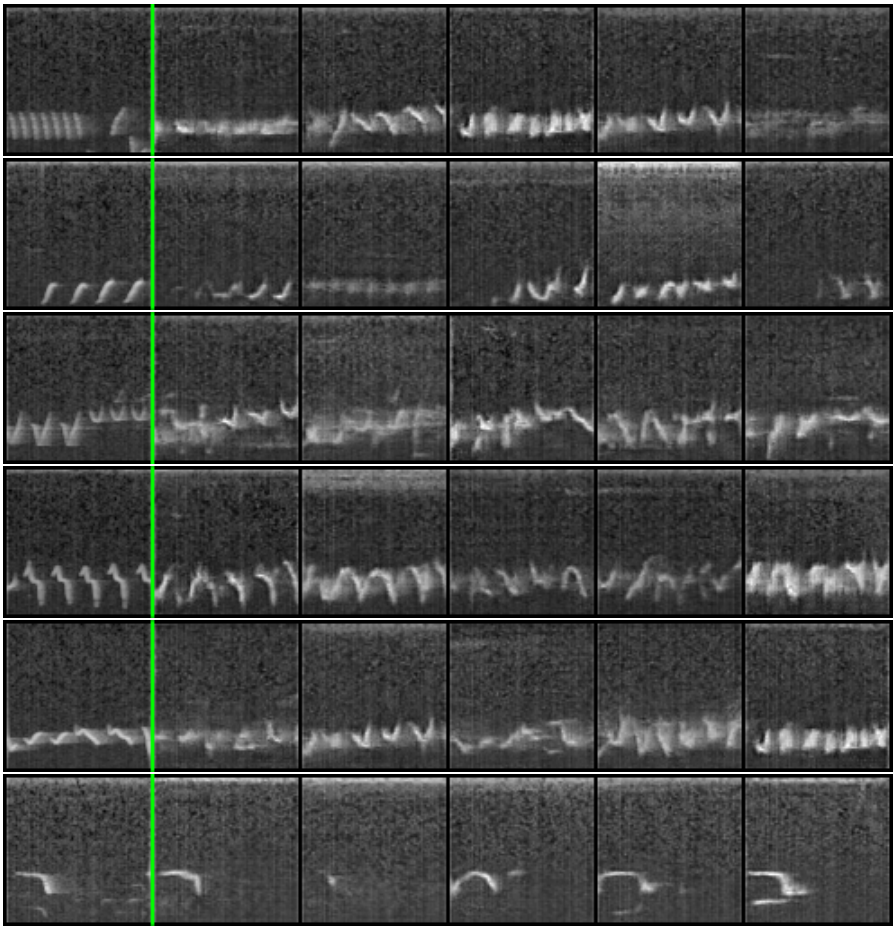


Abbildung 6.18: *Nearest Neighbours* von sechs charakteristischen Mustern des Fitis.

Links vom grünen Trenner ist das Original aus dem Datensatz abgebildet. Rechts davon befinden sich die fünf *Nearest Neighbours* der vom *Generator* erzeugten Daten.

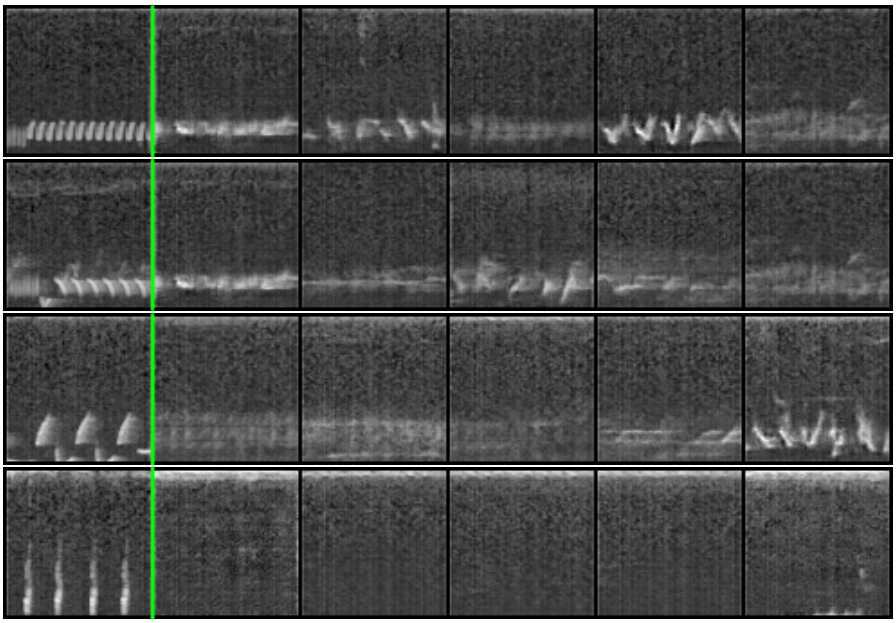


Abbildung 6.19: *Nearest Neighbours* von vier charakteristischen Mustern des Zaunkönigs.

Der Zaunkönig ist nicht Teil des Trainingsdatensatzes und wird vom *Generator* nicht synthetisiert. Links vom grünen Trenner ist das Original aus dem Datensatz abgebildet. Rechts davon befinden sich die fünf *Nearest Neighbours* der vom *Generator* erzeugten Daten.

6.2.2 Morphing

Dieser Abschnitt präsentiert exemplarisch das Ergebnis vom *Morphing* von verschiedenen Ausgangsvektoren mit der in Abschnitt 2.2 beschriebenen Formel $\vec{x}_{neu} = \vec{x}_1 \cdot (1-\alpha) + \vec{x}_2 \cdot \alpha$. Dafür werden zehn *Mel-Spektrogramme* verwendet, zwischen welchen schrittweise interpoliert wird. Die Abbildung 6.20 zeigt das Vorgehen exemplarisch. Hier stellt das *Mel-Spektrogramm* oben links das Ausgangssignal dar. Es wird in fünf Schritten zum nächsten *Mel-Spektrogramm* in Pfeilrichtung übergegangen. Das nächste Bild, rechts vom Startbild, stellt anschließend den Ausgangspunkt dar und es wird erneut in fünf Schritten in Pfeilrichtung interpoliert. Dieser Vorgang wird wiederholt, bis beim Vektor des Start *Mel-Spektrogramm* angekommen wird. Das daraus resultierende Audiosignal ist im *Repository*[7] zu finden.

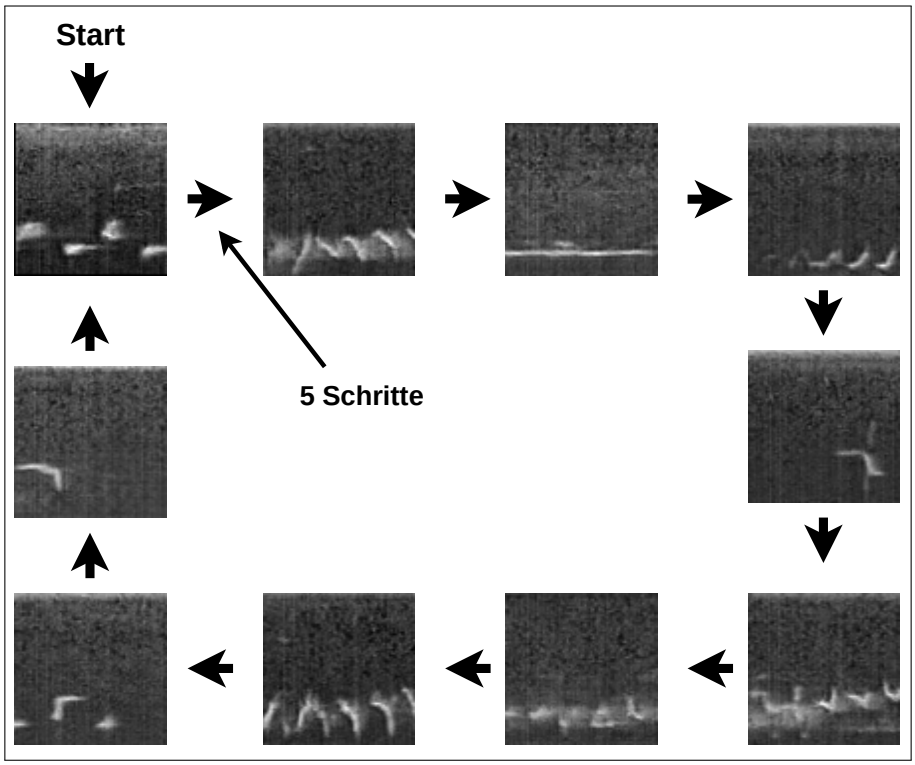


Abbildung 6.20: Exemplarisches Vorgehen beim *Morphing*.

Die folgende Abbildung 6.21 zeigt den schrittweisen Übergang dieser Bilder. Anhand des Beispiels wird sichtbar und hörbar, dass im *latent space* naheliegende Vektoren ähnliche

[7] https://github.com/batonfabi/master_thesis/blob/submission/kapitel_6/explore_model/nn_examples/morph.wav

Mel-Spektrogramme erzeugen. Diese Eigenschaft spricht für ein gut erstelltes *latent space* durch das *WGAN* (vgl. Abschnitt 2.2).

Es folgt eine Zusammenfassung der in diesem Abschnitt erzielten Ergebnisse.

6.3 Zusammenfassung der Ergebnisse

In diesem Kapitel wurde die Funktionalität der in Abschnitt 5.1 vorgestellten Architektur zur Synthese von Vogelgesang geprüft. Dabei wurden mit der Beantwortung der FF1 und FF2 die einzelnen Komponenten isoliert betrachtet und anschließend mit der Beantwortung von FF3 die Ergebnisse ihres Zusammenspiels analysiert. Sowohl die einzelnen Komponenten als auch die zusammengesetzte Architektur eignen sich für die Lösung der vorliegenden Problemstellung - dem Generieren von Vogelgesang.

Weiterhin haben die erzielten Ergebnisse gezeigt, dass der *FID-Score* einen guten Indikator für die Qualität der generierten *Mel-Spektrogramme* darstellt. Die steigende Qualität der generierten Daten resultierte in einem sinkenden *FID-Score*. Für den Vergleich von *Modellen*, dessen *FID-Score* sich nur in der Nachkommastelle unterscheidenden, wird eine manuelle Analyse empfohlen.

Anhand einer qualitativen Analyse der generierten *Mel-Spektrogramme*, die von dem *Modell* mit dem niedrigsten *FID-Score* erzeugt wurden, wurde gezeigt, dass die Signale aus dem Trainingsdatensatz unterschiedlich gut synthetisiert werden. In einigen Beispielen war keine Unterscheidung zwischen dem originalen und generierten Audiosignal möglich. Komplexe Muster wurden mit deutlicherem Qualitätsunterschied synthetisiert.

Im folgenden Abschnitt wird das Fazit der Arbeit gezogen und ein Ausblick über weitere Forschungsansätze gegeben.

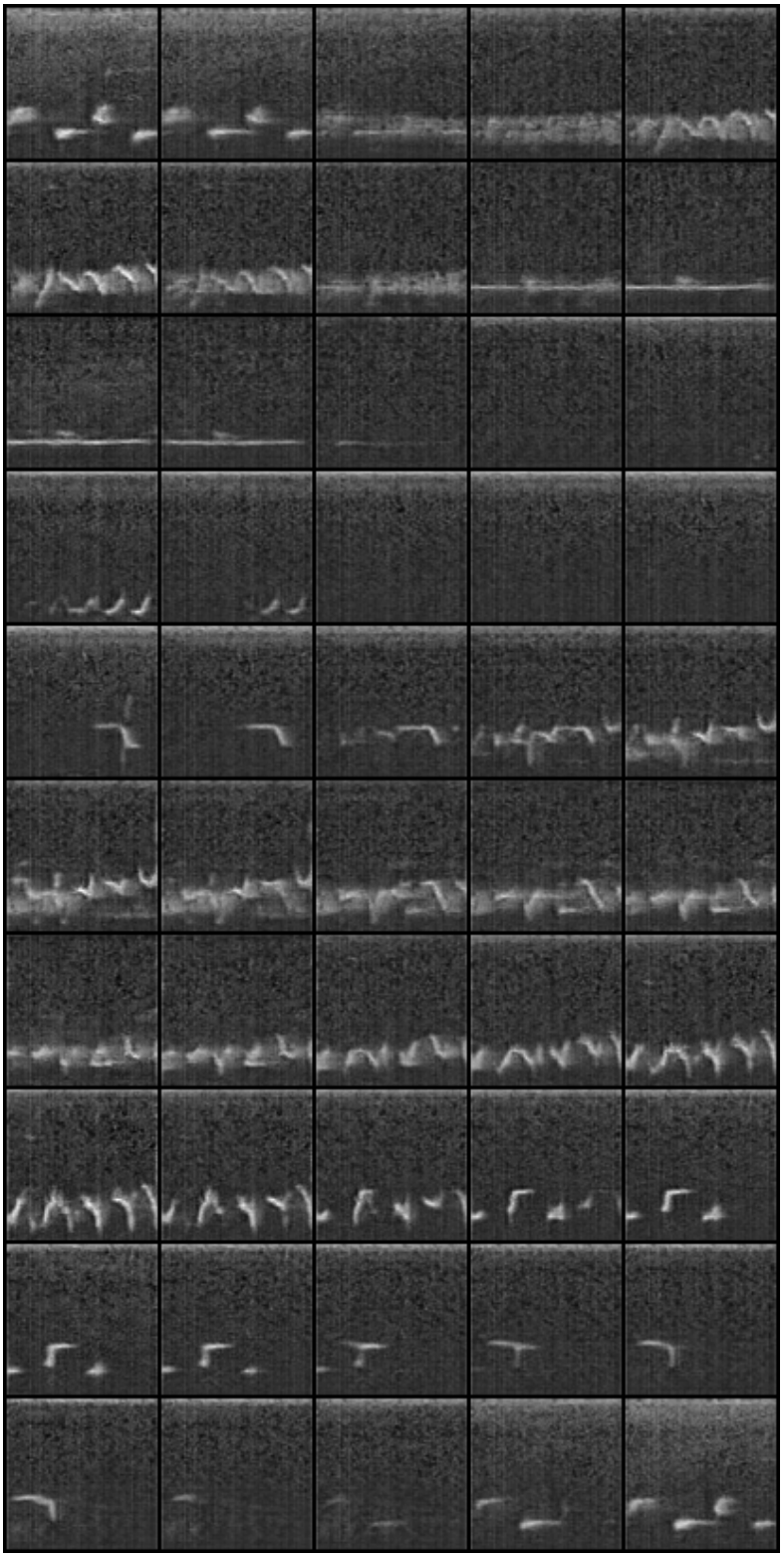


Abbildung 6.21: Bildliche Darstellung des Ergebnisses des *Morphings*.

7 Fazit und Ausblick

Dieser Abschnitt fasst im Fazit die Erkenntnisse der vorliegenden Arbeit zusammen und gibt anschließend einen Ausblick über mögliche Verbesserungen und weiterführende Forschungsansätze.

7.1 Fazit

Der Kern dieser Arbeit liegt in der Konzeption, Implementierung und Evaluation einer Architektur mit *Neuronalen Netzen* für die Synthese von Vogelgesang.

Im Bereich der *Neuronalen Netze* gibt es unterschiedliche Ansätze Audiosignale zu synthetisieren. Es wurde eine kurze Übersicht der gängigen Methoden vorgestellt, woraus sich *Wasserstein-GANs* von Arjovsky et al. (2017) als Fundament der vorliegenden Arbeit herauskristallisiert haben.

Um ihre Funktionsweise zu beschreiben, wurde zunächst auf die Grundlagen der *Generative Adversarial Networks (GANs)* (Goodfellow et al., 2014) und ihre Herausforderungen beim Training eingegangen. Aufbauend auf den Herausforderungen wurden die Optimierungen vorgestellt, die mit den *Wasserstein-GANs* einhergehen und gleichzeitig ihr Trainingsverfahren detailliert beschrieben. Abschließend wurden quantitative und qualitative Metriken vorgestellt, die für die Evaluation der Ergebnisse zum Einsatz kommen.

In Bezug auf die Darstellungsformen von Audiosignalen wurden *Spektogramme* und *Mel-Spektrogramme* behandelt. *Mel-Spektrogramme* sind auf das subjektive Hörempfinden der Menschen angepasst und wurden als bevorzugte Repräsentation der zu generierenden Vogelgesänge gewählt. Mit dieser Entscheidung entstand die zusätzliche Anforderung ein Vorgehen zu erarbeiten, die *Mel-Spektrogramme* in Audiosignale in *Wellenform* umzuwandeln.

Mit der Erörterung verwandter Arbeiten wurden zwei Themenfelder untersucht. Erstens wurden zwei Architekturen behandelt, die auf ähnliche Weise wie in der vorliegenden Arbeit Audiosynthese durchführen (Donahue et al., 2018). Zweitens wurde aus den Ergebnissen von Shen et al. (2017) ein Konzept übernommen, der die Umwandlung von *Mel-Spektogrammen* in *Wellenform* ermöglicht. Es wird ein sogenannter *Vocoder* eingesetzt, der im Rahmen der Sprachsynthese entwickelt und trainiert wurde (Prenger et al., 2018). Der für die Sprachsynthese vortrainierte *Vocoder* bildet, neben dem *WGAN*, den zweiten Baustein der hier vorgestellten Architektur.

Aus der Zielstellung der vorliegenden Arbeit und der beschriebenen Grundlagen wurden Forschungsfragen erarbeitet, die das Vorgehen der Arbeit ausrichten. Dabei stand die Fragestellung, ob die einzelnen Komponenten und ihr Zusammenspiel die Zielstellung, Vogelgesang zu synthetisieren, umsetzen können. Zusätzlich widmete sich eine Frage der quantitativen Evaluationsmöglichkeit von generierten *Mel-Spektogrammen* mithilfe des *FID-Scores*. An den Forschungsfragen angeknüpft wurden Anforderungen aufgestellt, die sich an die Versuchsdurchführung und Implementierung richten. Dabei spielten Gesichtspunkte wie die Protokollierung des Trainingsverlaufs und flexible Anpassbarkeit in Bezug auf die Trainingskonfiguration und die Architektur eine tragende Rolle.

Auf Basis der Zielstellung der vorliegenden Arbeit und der definierten Anforderungen wurde die Architektur zur Synthese von Vogelgesang entwickelt, die maßgeblich aus den drei Komponenten: *WGAN*, *Skalierer* und *Vocoder* besteht. Darüberhinaus wurde ein Trainingsdatensatz beschafft, die Trainingsdurchführung vorbereitet und alle notwendigen Komponenten implementiert.

Die Evaluation bezieht sich im ersten Schritt auf die aufgestellten Forschungsfragen. Diese wurden sukzessiv beantwortet, indem Versuche durchgeführt, verschiedene Konfigurationen der erarbeiteten Architektur trainiert und die resultierenden Ergebnisse interpretiert wurden. Nach der Beantwortung der Forschungsfragen wurde das *Modell* mit dem niedrigsten *FID-Score* einer qualitativen Analyse unterzogen. Dazu wurde aus einer Menge von generierten Daten *Nearest Neighbours* zu typischen Mustern in den *Mel-Spektogrammen* aus dem Trainingsdatensatz gesucht und diese visuell und akustisch miteinander verglichen. Abschließend wurde exemplarisch ein *Morphing* zwischen verschiedenen Vektoren durchgeführt und das erzeugte *latent space* grob untersucht.

Mit dieser Arbeit wurde gezeigt, dass die einzelnen Komponenten der Architektur für sich isoliert ihre Aufgaben erfolgreich umsetzen können: Der *WGAN* ist in der Lage Vogelgesang in Form von *Mel-Spektrogrammen* zu synthetisieren und der *Vocoder* kann

manuell erzeugte *Mel-Spektrogramme* von Vogelgesang ohne signifikantem Qualitätsverlust in *Wellenform* umwandeln.

Anschließend wurde an dem Zusammenspiel der Kernkomponenten demonstriert, dass die erarbeitete Architektur erfolgreich Vogelgesang synthetisieren kann. Die qualitative Analyse des *Modells* mit dem geringstem *FID-Score* hat ergeben, dass einfache Muster in den *Mel-Spektrogrammen* der Vogelgesänge so gut imitiert werden können, dass eine Unterscheidung zwischen originalen und generierten Signalen ohne weiteres unmöglich ist. Bei komplexer Mustern hingegen wurden Qualitätsunterschiede zu Originaldaten festgestellt. Mithilfe eines beispielhaften *Morphings* wurde gezeigt, dass der vom *WGAN* erzeugte *latent space* ähnliche Abbildungen an ähnlichen Punkten projiziert.

Zusätzlich wurde aufgewiesen, dass der *FID-Score* sich als Indikator für die Qualität der generierten *Mel-Spektrogramme* eignet.

Die erzielten Ergebnisse und Implementierungen sind für weiterführende Forschungen in dem zu der Arbeit zugehörigen *Repository*[1] und der beigefügten CD hinterlegt.

7.2 Ausblick

Die Arbeit hat eine Architektur bestehend aus zwei *Neuronalen Netzen*, dem *WGAN* und dem *Vocoder*, zur Generierung von Vogelgesang über *Mel-Spektrogramme* vorgestellt. Das ist nach aktuellem Wissensstand die erste wissenschaftliche Auseinandersetzung, Vogelgesang in dieser Form zu generieren und bietet somit eine Basis, auf der weitere Forschung betrieben werden kann.

Die folgende Auflistung stellt mögliche Themengebiete für weiterführende Forschungen auf, die auf Grundlage der vorliegenden Arbeit durchgeführt werden können. Die Reihenfolge der Punkte trifft keine Aussage über die Priorität der Themengebiete.

Länge der generierten Audiosignale: Die in der vorliegenden Arbeit trainierten Konfigurationen des *WGANs* erzeugen *Mel-Spektrogramme* im Format $80 \times 80 \times 1$. Diese werden vom *Vocoder* in Audiosignale der Länge von ca. einer Sekunde umgewandelt. Eine mögliche weiterführende Forschung könnte sich mit der Verlängerung der generierten Audiosignale beschäftigen. Das könnte über veränderte Konfigurationen des *WGANs* erfolgen.

[1] https://github.com/batonfabi/master_thesis/tree/submission/

Diversität des Trainingdatensatzes: In der vorliegenden Arbeit wurde ein Datensatz verwendet, der aus zwei Vogelarten besteht. Eine weiterführende Forschungsfrage kann sich mit dem Zusammenhang zwischen der Diversität der Trainingsdaten und der daraus resultierenden Qualität der generierten Daten auseinandersetzen. Ist die Qualität der generierten Daten bei einem vielfältigen Datensatz höher als bei einem Datensatz, der weniger Unterscheidungen aufweist?

Vergleich der erzeugten Audioqualität mit ähnlichen Ansätzen: Donahue et al. (2018) stellen zwei verwandte Herangehensweisen vor, Audiosignale zu synthetisieren. Ein Ansatz generiert direkt die Signale in *Wellenform*, der zweite Ansatz generiert *Spektrogramme*, die anschließend mithilfe des *Griffin-Lim-Algorithmus* umgewandelt werden. Wie reiht sich der hier vorgestellte Ansatz in Bezug auf die generierte Audioqualität ein?

Optimierung des Trainingdatensatzes: Wie in Abschnitt 5.2 beschrieben, enthält der in der vorliegenden Arbeit verwendete Trainingsdatensatz Störgeräusche. Die genauere Beschäftigung mit dem Datensatz und ihrer Bereinigung könnte zu besseren Ergebnissen in der Synthese von Vogelgesang führen.

Arithmetik im *latent space*: Welche arithmetischen Operationen sind innerhalb des *latent spaces* von Vogelgesang möglich? Kann beispielsweise zwischen männlichen und weiblichen, jungen und ausgewachsenen Vögeln, Rufen und Gesängen interpoliert werden?

Verwendung anderer *GAN* Architekturen: Ist es möglich mithilfe anderer *GAN-Architekturen*, statt des *WGANs*, bessere Ergebnisse in der Qualität der generierten Signale zu erzielen?

Literaturverzeichnis

Martin Arjovsky, Soumith Chintala, und Léon Bottou. Wasserstein gan. 2017.

Ali Borji. Pros and cons of gan evaluation measures, 2018.

Andrew Brock, Jeff Donahue, und Karen Simonyan. Large scale gan training for high fidelity natural image synthesis, 2019.

Jason Brownlee. How do convolutional layers work in deep learning neural networks?, 2020a. URL https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/. (Zugegriffen: 23.12.2020).

Jason Brownlee. How to explore the gan latent space when generating faces, 2020b. URL https://machinelearningmastery.com/how-to-interpolate-and-perform-vector-arithmetic-with-faces-using-a-generative-adversarial-network/. (Zugegriffen: 23.12.2020).

Francois Chollet. *Deep Learning with Python*. Manning Publications Co., USA, 1st edition, 2017. ISBN 1617294438.

Chris Donahue, Julian McAuley, und Miller Puckette. Adversarial audio synthesis. 2018.

David Foster. *Generative Deep Learning*. O'Reilly Media, Inc., 2019. ISBN 9781492041948.

L. A. Gatys, A. S. Ecker, und M. Bethge. Image style transfer using convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2414–2423, 2016. doi: 10.1109/CVPR.2016.265.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, und Yoshua Bengio. Generative adversarial networks, 2014.

D. Griffin und Jae Lim. Signal estimation from modified short-time fourier transform. volume 8, pages 804–807. IEEE, 1983. doi: 10.1109/ICASSP.1983.1172092.

Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, und Aaron Courville. Improved training of wasserstein gans. 2017.

Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, und Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium, 2018.

Nabil Hewahi, Salman AlSaigal, und Sulaiman AlJanahi. Generation of music pieces using machine learning: long short-term memory neural networks approach. *Arab Journal of Basic and Applied Sciences*, 26(1):397–413, 2019. doi: 10.1080/25765299.2019.1649972. URL https://doi.org/10.1080/25765299.2019.1649972.

Nasser Kehtarnavaz. Chapter 7 - frequency domain processing. In Nasser Kehtarnavaz, editor, *Digital Signal Processing System Design (Second Edition)*, pages 175–196. Academic Press, Burlington, second edition edition, 2008. ISBN 978-0-12-374490-6. doi: https://doi.org/10.1016/B978-0-12-374490-6.00007-6. URL https://www.sciencedirect.com/science/article/pii/B9780123744906000076.

Asifullah Khan, Anabia Sohail, Umme Zahoora, und Aqsa Saeed Qureshi. A survey of the recent architectures of deep convolutional neural networks. 2019. doi: 10.1007/s10462-020-09825-6.

Diederik P. Kingma und Jimmy Ba. Adam: A method for stochastic optimization, 2017.

Diederik P Kingma und Max Welling. Auto-encoding variational bayes, 2014.

Kundan Kumar, Rithesh Kumar, Thibault de Boissiere, Lucas Gestin, Wei Zhen Teoh, Jose Sotelo, Alexandre de Brebisson, Yoshua Bengio, und Aaron Courville. Melgan: Generative adversarial networks for conditional waveform synthesis. 2019.

L. Latha und Coimbatore Tamilnadu. Efficient approach to normalization of multimodal biometric scores. *International Journal of Computer Applications (0975 - 8887)*, 32 (10):57–64, 2011.

Bryan Lim und Stefan Zohren. Time series forecasting with deep learning: A survey, 2020.

Bei Liu, Jianlong Fu, Makoto P. Kato, und Masatoshi Yoshikawa. Beyond narrative description. *Proceedings of the 26th ACM international conference on Multimedia*, Oct 2018. doi: 10.1145/3240508.3240587. URL http://dx.doi.org/10.1145/3240508.3240587.

Miriam. Introducing variational autoencoders (in prose and code), 2016. URL https://blog.fastforwardlabs.com/2016/08/12/introducing-variational-autoencoders-in-prose-and-code.html. (Zugegriffen: 23.12.2020).

Sudipto Mukherjee, Himanshu Asnani, Eugene Lin, und Sreeram Kannan. Clustergan : Latent space clustering in generative adversarial networks. 2018.

NVIDIA. Mel spectrogram, 2018. URL https://docs.nvidia.com/deeplearning/dali/user-guide/docs/examples/audio_processing/spectrogram.html. (Zugegriffen: 01.01.2021).

NVIDIA. Waveglow, 2020. URL https://pytorch.org/hub/nvidia_deeplearningexamples_waveglow/. (Zugegriffen: 23.02.2021).

Augustus Odena, Vincent Dumoulin, und Christopher Olah. Deconvolution and checkerboard artifacts. 2016.

D. O'Shaughnessy. *Hearing*, pages 109–139. 2000. doi: 10.1109/9780470546475.ch4.

Aravind Pai. Want to generate your own music using deep learning? here's a guide to do just that!, 2020. URL https://www.analyticsvidhya.com/blog/2020/01/how-to-perform-automatic-music-generation/. (Zugegriffen: 02.02.2021).

Ryan Prenger, Rafael Valle, und Bryan Catanzaro. Waveglow: A flow-based generative network for speech synthesis. 2018.

pytorch. Tanh, 2019. URL https://pytorch.org/docs/stable/generated/torch.nn.Tanh.html. (Zugegriffen: 20.04.2021).

Alec Radford, Luke Metz, und Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. 2015.

Danilo Jimenez Rezende, Shakir Mohamed, und Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models, 2014.

Tim Salimans, Ian J. Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, und Xi Chen. Improved techniques for training gans. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, und Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2226–2234,

2016. URL http://papers.nips.cc/paper/6125-improved-techniques-for-training-gans.

Jonathan Shen, Ruoming Pang, Ron J. Weiss, Mike Schuster, Navdeep Jaitly, Zongheng Yang, Zhifeng Chen, Yu Zhang, Yuxuan Wang, RJ Skerry-Ryan, Rif A. Saurous, Yannis Agiomyrgiannakis, und Yonghui Wu. Natural tts synthesis by conditioning wavenet on mel spectrogram predictions. 2017.

Yujun Shen, Jinjin Gu, Xiaoou Tang, und Bolei Zhou. Interpreting the latent space of gans for semantic face editing, 2020.

Barry Truax. Handbook for acoustic ecology, 1999. URL https://www.sfu.ca/sonic-studio-webdav/handbook/Mel.html. (Zugegriffen: 01.01.2021).

Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, und Koray Kavukcuoglu. Wavenet: A generative model for raw audio. 2016.

Aäron van den Oord, Yazhe Li, Igor Babuschkin, Karen Simonyan, Oriol Vinyals, Koray Kavukcuoglu, George van den Driessche, Edward Lockhart, Luis Carlos Cobo Rus, Florian Stimberg, Norman Casagrande, Dominik Grewe, Seb Noury, Sander Dieleman, Erich Elsen, Nal Kalchbrenner, Heiga Zen, Alexander Graves, Helen King, Thomas Walters, Dan Belov, und Demis Hassabis. Parallel wavenet: Fast high-fidelity speech synthesis. Technical report, Google Deepmind, 2017. URL https://arxiv.org/abs/1711.10433.

Lilian Weng. From gan to wgan. 2019.

A Anhang

A.1 Voruntersuchung

Im Rahmen der Konkretisierung der Zielstellung werden verschiedene Modelle untersucht, um ihre Erfolgschancen vorab einzuschätzen. Dabei werden nur Ansätze unter Betracht gezogen, die Daten mithilfe eines *latent spaces* generieren, da die *autoregressiven* Konzepte die Ressourcen der vorliegenden Arbeit überschreiten würde. Es wird hier kurz auf die zwei wichtigsten Versuche eingegangen. Um den Rahmen dieses Abschnittes zu bleiben, wird von Implementierungsdetails abgesehen.

A.1.1 Voruntersuchung von Variational Autoencoder

Bei der ersten untersuchten Methode handelt es sich um *VAE*. Dabei werden zwei Versuche durchgeführt, um das Potenzial der *VAEs* im Zusammenhang mit der Synthese von Vogelgesang zu überprüfen:

Versuch 1: Der erste Versuch konzentriert sich auf den vom *VAE* erzeugten *latent space*. Wie in Abschnitt 2.2 beschrieben, geht man bei einem gut trainierten *latent space* von einer deutlichen räumlichen Trennung der Punkte verschiedener Klassen aus. Diese erwünschte Eigenschaft wird untersucht, indem ein *VAE* mit einem Datensatz mit zwei verschiedenen Vogelarten trainiert und der *latent space* auf zwei Dimensionen begrenzt wird.

Die Ergebnisse diverser Durchläufe mit verschiedenen Konfigurationen des *VAEs* haben die erwartete Verteilung nicht erfüllt. Gleichzeitig bedeutet dieses Ergebnis nicht, dass *VAEs* ungeeignet für die vorliegende Aufgabe sind. Dieses Ergebnis kann aufgrund der reduzierten Dimensionalität des *latent spaces* resultieren. Um diese Fehlerquelle auszuschließen, wird folgender Versuch durchgeführt.

Versuch 2: Für den zweiten Versuch wird der *latent space* auf 100 Dimensionen erweitert. Der Fokus liegt diesmal auf der Qualität der Ausgabewerte. Für das Training wird ein Datensatz verwendet, der aus *Spektrogrammen* der Größe 128×128 besteht, welche sich mithilfe des *Griffin-Lim*-Algorithmus (Griffin und Lim, 1983) wieder in Wellenform mit akzeptabler Qualität umwandeln lassen.

Dieses Experiment wird mit unterschiedlichen Konfigurationen durchgeführt. Die Ergebnisse aller untersuchten Konfigurationen haben die gewünschte Qualität nicht erreichen können - die erzeugten Signale in *Wellenform* werden nur als Rauschen wahrgenommen.

Die untersuchten Architekturen und Trainingsdurchläufe orientieren sich an dem Aufbau von (Foster, 2019, S. 86ff) und sind entsprechend der Ein- und Ausgangsgrößen angepasst. Die hier aufgeführten Ergebnisse bilden keine abschließende Beurteilung für die Eignung der *VAEs* im Kontext der Synthese von Vogelgesang. Diese werden als richtungsweisende Grundlage für das weitere Vorgehen der vorliegenden Arbeit verwendet: Mit den Ergebnissen der Voruntersuchung werden *VAEs* für die Synthese von Vogelgesang ausgeschlossen, weil es vielversprechendere Konzepte gibt.

A.1.2 Voruntersuchung von GANs

Die zweite Voruntersuchung konzentriert sich auf die Audiosynthese mithilfe von *GANs*. Dabei wird direkt die Ausgabequalität untersucht und der erzeugte *latent space* des *GANs* außer Acht gelassen. Das wird damit begründet, dass die Analyse des vom *GAN* erzeugten *latent space* aufwändiger ist als beim *VAE*[1]. Für das Training des *GANs* wird der gleiche Datensatz verwendet wie bei der vorangehenden Untersuchung. Dieser besteht aus *Spektrogrammen* der Größe 128×128, die mithilfe des *Griffin-Lim*-Algorithmus in Wellenform umgewandelt werden können. Die untersuchte Architektur ist mit dem in Abschnitt 3.1 beschriebenen *SpecGAN* vergleichbar und liefert deutlich bessere Ergebnisse als der zuvor untersuchte *VAE*.

Das Ergebnis der Voruntersuchung zeigt, dass *GANs* im Kontext der Audiosynthese von Vogelgesang vielversprechender sind als *VAEs*. Aus diesem Grund wird in der vorliegenden Arbeit ein Ansatz, der im Kern aus einem *GAN* besteht, weiter verfolgt.

[1] Das liegt daran, dass mithilfe des *Encoders* des *VAEs* der *latent space* kartografiert werden kann, wohingegen es bei dem *GAN* diese Möglichkeit fehlt.

A.1.3 Aus der Voruntersuchung abgeleitete Schritte

Das weitere Vorgehen konzentriert sich auf die Optimierung der Qualität bei gleichzeitiger Verlängerung der generierten Signaldauer. Im weiteren Verlauf werden die folgenden Strategien verwendet, um diese Ziele zu erreichen:

- Die Reduzierung der Komplexität des *Modells* und der Problemstellung führt in der Regel zu einem stabileren Lernverhalten, einem erhöhten Lerntempo und einer besseren Qualität der generierten Daten.
- Die Kalibrierung der Hyperparameter des *Modells* auf das gegebene Problem kann die Qualität der generierten Daten zusätzlich erhöhen. Zu den Hyperparametern gehören die Filter- und Kernelgröße, Lernrate, Batchgröße und weitere.

Die Komplexität der Problemstellung lässt sich über das Einfließen von Domänenwissen bzw. der Vorverarbeitung des verwendeten Datensatzes erreichen. Die Vorverarbeitung des Datensatzes wird auch als *Merkmalsextraktion* bezeichnet, in dem entscheidende Merkmale hervorgehoben werden. Abschnitt 2.5 zeigt zwei Beispiele der *Merkmalextraktion* in Bezug auf Audioverarbeitung: die *Spektrogramme* und *Mel-Spektrogramme*. Sie heben den Frequenzbereich der in Wellenform vorliegenden Audiosignale hervor.

Als *Merkmalsextraktion* wird in der vorliegenden Arbeit im weiteren Verlauf die an das menschliche Hörempfinden angepasste *Mel-Spektrogramme* verwendet, was eine direkte Auswirkung auf die Architektur hat: es muss die Funktionalität berücksichtigt werden, die eine Umwandlung der *Mel-Spektrogramme* in Wellenform ermöglicht.

Die Kalibrierung der Hyperparameter erfolgt im Rahmen der *Versuchsdurchführung* in Kapitel 6 und wird dort im Detail beschrieben.

A.2 Implementierungsdetails

A.2.1 Gradient Panelty

```python
import torch
import torch.nn as nn


# Berechnung des gradient panelties (||∇_x̂ D_w(x̂)||_2 − 1)^2
def gradient_penalty(critic, real, fake, device="cpu"):

    BATCH_SIZE, C, H, W = real.shape
    epsilon = torch.rand((BATCH_SIZE, 1, 1, 1)).repeat(1, C, H, W).to(device)

    # Berechnung von x̂ ← εx + (1 − ε)x̃
    # hier heißt x̂ gleich interpolated_images
    interpolated_images = epsilon * real + fake * (1 - epsilon)

    # Berechnung von D_w(x̂)
    mixed_scores = critic(interpolated_images)

    # Berechnung des Gradienten D_w(x̂)
    gradient = torch.autograd.grad(
        inputs=interpolated_images,
        outputs=mixed_scores,
        grad_outputs=torch.ones_like(mixed_scores),
        create_graph=True,
        retain_graph=True,
    )[0]

    gradient = gradient.view(gradient.shape[0], -1)

    # Berechnung von ||∇_x̂ D_w(x̂)||_2
    gradient_norm = gradient.norm(2, dim=1)

    # Berechnung von (||∇_x̂ D_w(x̂)||_2 − 1)^2
    gradient_penalty = torch.mean((gradient_norm - 1) ** 2)
    return gradient_penalty
```

Python Quelltext 5: Implementierung der Berechnung des *gradient panelties*.

A.3 Evaluation

A.3.1 FF1

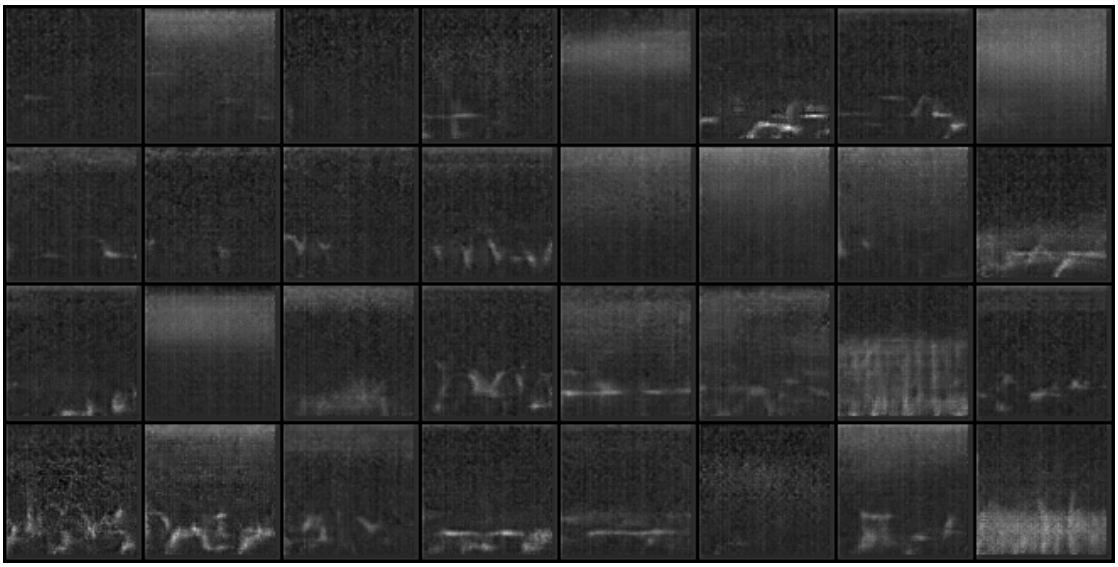


Abbildung A.1: Beispiele von generierten Daten des *Generators* 1 mit dem *Loss* am nächsten zur null.

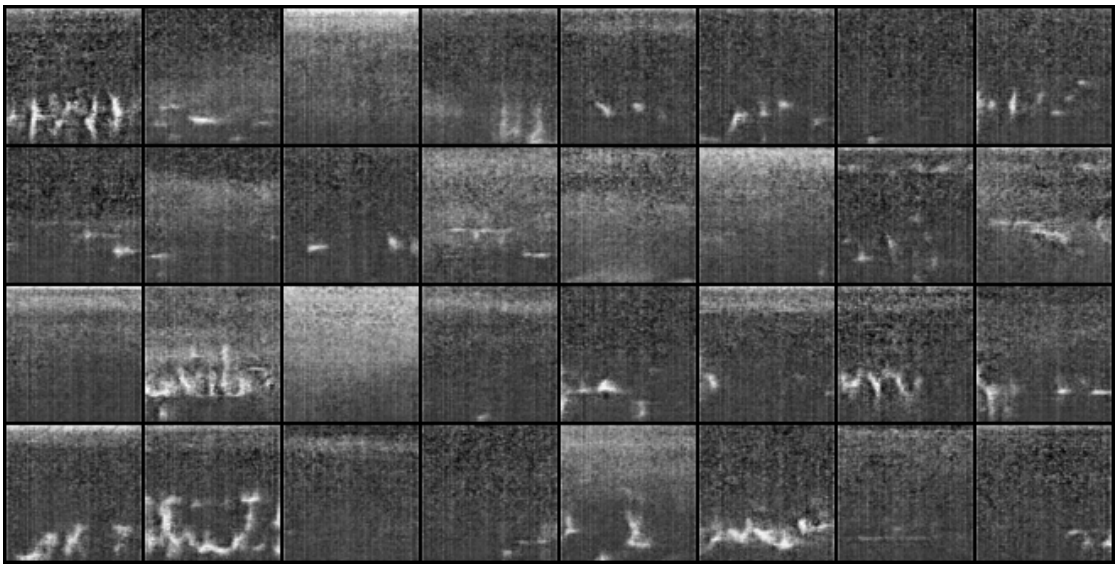


Abbildung A.2: Beispiele von generierten Daten des *Generators* 2 mit dem *Loss* am nächsten zur null.

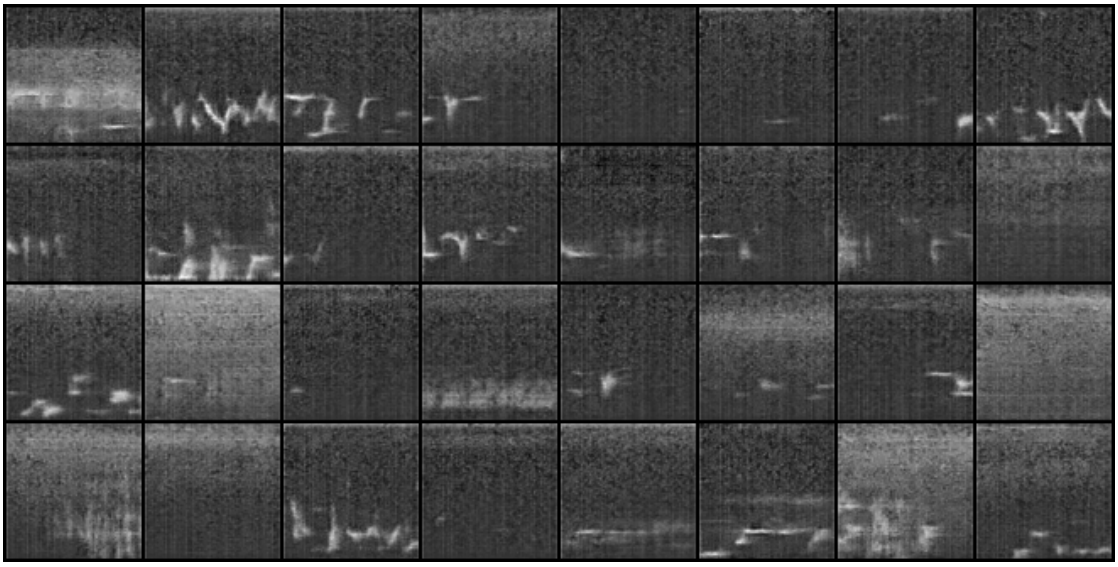


Abbildung A.3: Beispiele von generierten Daten des *Generators* 3 mit dem *Loss* am nächsten zur null.

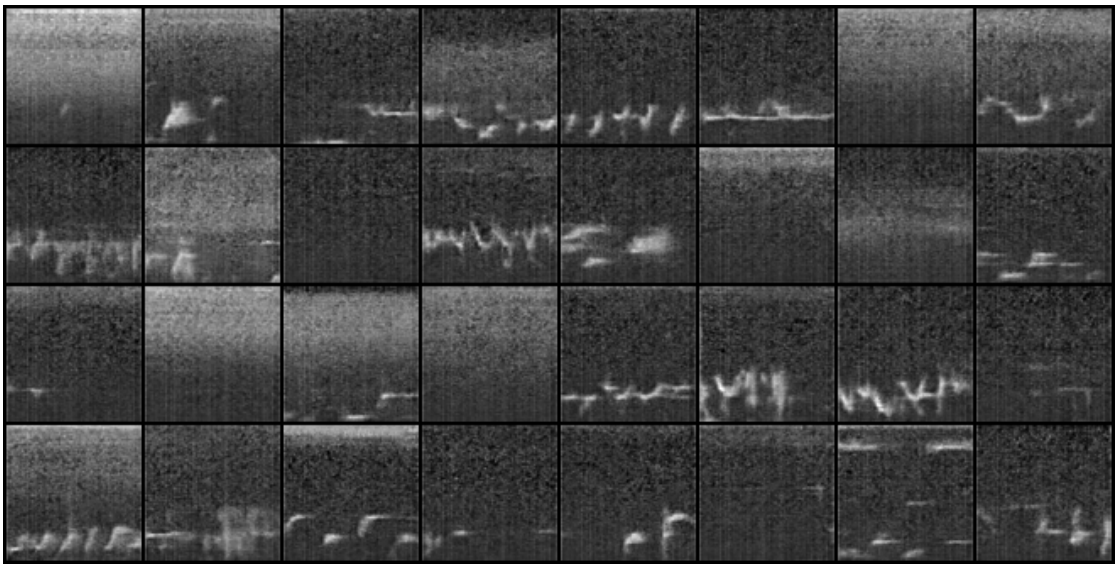


Abbildung A.4: Beispiele von generierten Daten des *Generators* 5 mit dem *Loss* am nächsten zur null.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Masterarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Synthetisierung von Audiosignalen mithilfe Neuronaler Netze am Beispiel von Vogelgesang

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original