

Masterarbeit

Finn Christian Severin

Entwicklung von Deep Learning Methoden zur Kontrolle eines
Kurzzeitlasers

Finn Christian Severin

Entwicklung von Deep Learning Methoden zur Kontrolle eines Kurzzeitlasers

Masterarbeit eingereicht im Rahmen der Masterprüfung
im gemeinsamen Masterstudiengang Mikroelektronische Systeme
am Fachbereich Technik
der Fachhochschule Westküste
und
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Klaus Jünemann
Zweitgutachter: Prof. Dr. Rer. Nat. Kristina Schädler

Eingereicht am: 27. Januar 2025

Finn Christian Severin

Thema der Arbeit

Entwicklung von Deep Learning Methoden zur Kontrolle eines Kurzzeitalasers

Stichworte

Maschinelles Lernen, Deep Learning, Laserphysik, Ultrakurze Laserpulse

Kurzzusammenfassung

Entwicklung einer Deep Learning Methode zur Rekonstruktion von ultrakurzen Laserpulsen aus einem FROG-Trace

Finn Christian Severin

Title of Thesis

Development of Deep Learning methods to controll short pulse lasers

Keywords

Machine learning, Deep Learning, Laser physics, Ultra-short lasers

Abstract

Development of a Deep Learning method to reconstruct ultra short laser pulses from a FROG-trace

Inhaltsverzeichnis

Abbildungsverzeichnis	vii
Tabellenverzeichnis	ix
Abkürzungen	x
Symbolverzeichnis	xi
1 Einleitung	1
1.1 Motivation	1
1.2 Forschungsfragen	2
1.3 Verwendete Software	2
1.4 Aufbau der vorliegenden Arbeit	3
2 Ultrakurze Laserpulse	4
2.1 Fourierlimitierte Laserpulse und Dispersion	4
2.1.1 Fourierlimitierte Laserpulse	4
2.1.2 Dispersion	4
2.1.3 Das Zeit-Bandbreite-Produkt (TBD)	6
2.2 Autokorrelation	9
2.3 Frequency-Resolved-Optical-Gating (FROG)	11
2.4 Phase Retrieval und FROG-Fehler	17
2.5 Triviale Ambiguitäten	21
2.6 Hilbert-Transformation	24
3 Machine Learning	31
3.1 DenseNet	31
3.1.1 Grundlagen	31
3.1.2 Struktur des DenseNets	32
3.1.3 Anpassungen	34

3.2	Genutzter Datensatz	40
3.2.1	Struktur des Datensatzes	40
3.2.2	Vorverarbeitung	40
3.2.3	Einheitliche Vorverarbeitung für simulierte und experimentelle Daten	43
3.3	Rekonstruktion der Zeitprofile von Kurzzeitlasern mithilfe eines DenseNets	54
3.3.1	Verwendete Trainingsschleifen	54
3.3.2	Learning Rate Scheduling	60
3.4	Optimierer und Regularisierung	61
3.5	Verlustfunktion	62
3.5.1	Allgemeines	62
3.5.2	Gewichtete Mittlere Quadratische Abweichung (MSE) Funktion . .	64
3.5.3	FROG-Fehler	66
4	Auswertung der Trainingsergebnisse	68
4.1	Allgemeines	68
4.2	Einfluss des Schedulers auf die Vorhersage	68
4.3	Einfluss des TBD auf die Vorhersage	71
4.4	Einfluss von Rauschen auf die Vorhersage	74
4.5	Einfluss der Hyperparameter der Verlustfunktion auf die Vorhersage . . .	76
4.5.1	Einfluss der Hyperparameter der Verlustfunktion beim überwachten Trainieren ohne FROG-Fehler	76
4.5.2	Einfluss der Hyperparameter der Verlustfunktion beim überwachten Trainieren mit FROG-Fehler	80
4.5.3	Vorhersage bei unüberwachten Trainieren mit FROG-Fehler . . .	81
4.6	Einfluss der Anzahl an Trainingsepochen auf die Vorhersage	81
4.7	Vergleich mit der Software von Swamp-Optics	82
5	Ausblick	85
	Literaturverzeichnis	87
A	Anhang	90
A.1	Listings	90
A.2	Abbildungen	92
A.3	Hyperparameter	94
A.4	Verwendete Hilfsmittel	102

Selbstständigkeitserklärung

103

Abbildungsverzeichnis

2.1	Schematische Darstellung von Dispersionseffekten. Ein fourierlimitierter Laserpuls (orange) wandert durch ein dispersives Medium und ist danach verbreitert und geschirpt (rot)	5
2.2	Zeitliche Breite und spektrale Bandbreite eines Gausspuls im Zeit- (blau) und Frequenzbereich (orange)	8
2.3	Experimenteller Aufbau der Messung der Intensitätsautokorrelation mit Second Harmonic Generation (SHG)-Kristall	9
2.4	Experimenteller Aufbau der Messung eines FROG-Traces mit SHG-Kristall	11
2.5	SHG-FROG Spektrogramm mit Verzögerungs- und Wellenlängenachse . .	15
2.6	Schematische Darstellung eines Algorithmus zum Phase Retrieval	18
2.7	Vergleich zweier Spektrogramme. Das zweite Spektrogramm wurde aus dem Zielsignal des ersten erzeugt	19
2.8	Darstellung der Schritte der diskreten Hilbert-Transformation	28
2.9	Vergleich eines Zielsignals mit einem durch die Hilbert-Transformation erzeugten analytischen Signal	29
3.1	Ein $L = 5$ Schichten großer DenseBlock mit $k_0 = 5$ Eingangskanälen und einer Wachstumsrate von $k = 4$	34
3.2	Struktur eines DenseNet bestehend aus 3 DenseBlocks mit dazwischenliegenden Übergangsschichten, welche die Größe der Merkmalskarten ändert	34
3.3	Funktionsverlauf einer Rectified Linear Unit (ReLU)-Aktivierungsfunktion	36
3.4	Funktionsverlauf einer Tangens Hyperbolicus (tanh)-Aktivierungsfunktion	37
3.5	Experimentell ermitteltes Spektrogramm	44
3.6	Vergleich eines experimentell bestimmten Spektrogramms vor und nach seiner Vorverarbeitung. Links in linearer und rechts in logarithmischer Darstellung	53
3.7	Beispielverlauf der Lernrate bei Verwendung eines 1Cycle Lernratenschedulers	60

4.1	Vergleich des Trainings- und Testfehlers beim Training mit konstanter Lernrate, linear ansteigender Lernrate und 1Cycle Lernratenscheduler . . .	69
4.2	Boxplots des Testfehlers beim Training mit konstanter Lernrate, linear ansteigender Lernrate und 1Cycle Lernratenscheduler	70
4.3	Histogramm der TBD-Werte im vorliegenden Datensatz	71
4.4	Testfehler eines neuronalen Netzes über den TBD-Wert von 100 Spektrogrammen sowie Approximation als lineare Gleichung	72
4.5	Boxplots des Testfehlers beim Training mit einem maximalen TBD-Wert von 20, 1,063 und 0,5514	73
4.6	Unterschiedlich stark verrauschte Spektrogramme	74
4.7	Boxplots des Testfehlers beim Training mit einem Anstieg des Rauschens der Spektrogramme in 4 Stufen	75
4.8	Vergleich eines Zielsignals mit einer Voraussage bei gleicher Gewichtung von Real- und Imaginärteil	77
4.10	Trainingsfehler bei Nutzung des MSE der Phase	78
4.9	Vergleich eines Zielsignals mit einer Voraussage bei gleicher Gewichtung von Intensität und Realteil	79
4.11	Boxplots des Testfehlers beim überwachten Training mit Gewicht des FROG-Fehlers von 1,5 und 10	80
4.12	Boxplots des Testfehlers bei unterschiedlicher Trainingslänge	82
4.13	Boxplot des Testfehlers bei Nutzung des bisher besten Modells	83
4.14	Vergleich eines Zielsignals mit einer Voraussage bei Nutzung des bisher besten Modells	84
A.1	Vergleich eines Zielsignals mit einer Voraussage bei unüberwachtem Lernen ohne vortrainiertes Netz	92
A.2	Beispielpuls nach unüberwachtem Training auf einem vortrainierten Modell	93

Tabellenverzeichnis

3.1	Größe der Eingangs- und Ausgangsmerkmalskarten des angepassten DenseNet sowie Position der Aktivierungsfunktionen	35
3.2	Standardabweichung und Mittelwert für die Normalisierung von jedem Kanal der Spektrogramme für das verwendete DenseNet	43
A.1	Hyperparameter des Trainings zum Vergleich des Effekts des Rauschens	94
A.2	Hyperparameter des Trainings zum Vergleich von Lernratenschedulern	95
A.3	Hyperparameter des Trainings zum Vergleich des Effekts des TBD-Werts	96
A.4	Hyperparameter zum Ermitteln eines Testfehlers, der unabhängig von den Hyperparametern des Trainings ist	97
A.5	Hyperparameter des Trainings zum Vergleich ihrer Auswirkung auf das überwachte Lernen	98
A.6	Hyperparameter des Trainings zum Vergleich ihrer Auswirkung des FROG-Fehlers beim überwachten Lernen	99
A.7	Hyperparameter des Trainings zum Vergleich der Auswirkung der Epochenanzahl	100
A.8	Hyperparameter des bisher besten Modells	101
A.9	Verwendete Hilfsmittel und Werkzeuge	102

Abkürzungen

DTFT Diskrete Fouriertransformation.

FFT Schnelle Fouriertransformation.

FROG Frequency-Resolved-Optical-Gating.

FSR FROG-Abtastrate.

FWHM Full-Width-at-Half-Maximum.

GDD Gruppenlaufzeitdispersion.

MSE Mittlere Quadratische Abweichung.

ReLU Rectified Linear Unit.

RMS Root-Mean-Square.

SGD Statistischer Gradientenabstieg.

SHG Second Harmonic Generation.

SNR Signal Rausch Abstand.

tanh Tangens Hyperbolicus.

TBD Zeit-Bandbreite-Produkt.

Symbolverzeichnis

$\Delta\lambda_{FWHM}$ Bandbreite eines Pulses.

$\Delta\omega_{RMS}$ Bandbreite eines Pulses.

$\Delta\omega_{FWHM}$ Bandbreite eines Pulses.

TBD_{RMS} Zeit-Bandbreite-Produkt.

TBD Zeit-Bandbreite-Produkt.

β Exponentieller Abfall.

c_n Frequenzspezifische Konstante.

λ_w Gewichts zurücknahme.

g Gradienten.

τ_{gr} Gruppenlaufzeit.

ϵ Konstante, um Division durch Null zu verhindern.

ω Kreisfrequenz.

α Lernrate.

c Lichtgeschwindigkeit.

c_0 Lichtgeschwindigkeit im Vakuum.

λ_c Mittenwellenlänge.

m Erster Moment.

v Zweiter Moment.

θ Parameter.

ϕ Phase.

n_{refrak} refraktiver Index.

$\Delta\tau$ Schrittweite in Verzögerungsrichtung.

$\Delta\lambda$ Schrittweite in Wellenlängenrichtung.

I_{FROG} Gemessenes Spektrogramm.

$I_{FROG}^{(k)}$ Rückgewonnenes Spektrogramm.

m Steigung eines Signals.

$\Delta\tau_{RMS}$ Zeitliche Breite eines Pulses.

$\Delta\tau_{FWHM}$ Zeitliche Breite eines Pulses.

1 Einleitung

1.1 Motivation

Kurzzeitleaser sind mit einer Dauer von wenigen Femtosekunden ($1 \text{ fs} = 1 \cdot 10^{-15} \text{ s}$) die kürzesten von Menschen erzeugten Ereignisse und gehören zu den kürzesten Ereignissen im Universum. Das Messen von biologischen Prozessen, wie dem Faltungsvorgang von Proteinen, aber auch physikalische Vorgänge finden auf einer vergleichbaren zeitlichen Skala statt. Dies macht es unmöglich, sie mithilfe von beispielsweise elektrischen Signalen zu messen und zu charakterisieren. Denn zur Messung eines Ereignisses wird immer ein noch kürzeres Ereignis benötigt. Einzig ultrakurze Laserpulse selbst sind kurz genug, um sich für das Messen solcher Prozesse zu eignen. Es ist hierbei jedoch wichtig, dass die genaue Dauer, Intensität und Phase des Pulses bekannt ist, da dieser andernfalls nicht zur korrekten Charakterisierung der zu messenden Ereignisse verwendet werden kann. [20, S. 2]

Dies führt jedoch zu einem Dilemma. Wenn Kurzzeitleaser die kürzesten Ereignisse sind und man zum Messen eines Ereignisses immer ein noch kürzeres benötigt, können diese nicht korrekt charakterisiert werden. Hier wird das so genannte FROG-Verfahren relevant. In diesem wird der ultrakurze Laserpuls genutzt, um sich selbst zu messen. Das Ergebnis von FROG liefert ein Spektrogramm, aus welchem mithilfe eines Algorithmus die Intensität und Phase eines Laserpulses zurückgewonnen werden können. [20, S. 4]

Bisher werden für die Rekonstruktion iterative Verfahren verwendet, die eine hohe Berechnungsdauer aufweisen. Ein Beispiel hierfür ist die Software zur FROG-Rekonstruktion von Swamp-Optics. In den letzten Jahren ist es jedoch zu einer rasanten Entwicklung im Bereich des maschinellen Lernens gekommen. Hier könnte das Potential bestehen, mit einem entsprechend trainierten neuronalen Netz in einem Schritt das Zeitprofil eines Pulses aus dessen Spektrogramm vorauszusagen. Methoden hierzu sollen in der vorliegenden Arbeit untersucht werden.

1.2 Forschungsfragen

In dieser Arbeit soll untersucht werden, wie maschinelles Lernen zur Charakterisierung eines Pulses aus dessen durch das FROG-Verfahren erhaltene Spektrogramm genutzt werden kann. Hierzu sollen die folgenden Forschungsfragen untersucht werden:

- Wie kann ein neuronales Netz zur Vorhersage des Signalverlaufs eines ultrakurzen Laserpulses auf künstlich erstellten Spektrogrammen und den zugehörigen Signalen trainiert werden?
- Wie kann eine Verlustfunktion zum Trainieren eines solchen Netzes aussehen?
- Welcher Ansatz kann verfolgt werden, um ein mit künstlichen Daten vortrainiertes neuronales Netz auf experimentell ermittelten Spektrogrammen zu trainieren, ohne den entsprechenden Signalverlauf zu kennen?

1.3 Verwendete Software

Der Quellcode für diese Arbeit ist in **Python** geschrieben, einer der populärsten Programmiersprachen im Bereich der Datenverarbeitung und des maschinellen Lernens [8]. Die Bibliothek **PyTorch** stellt hierzu viele Funktionen und Klassen zur einfachen Umsetzung von maschinellem Lernen zur Verfügung. Sie wird in dieser Arbeit genutzt, da sie einen guten Kompromiss aus Simplizität und Modifizierbarkeit bildet. PyTorch implementiert außerdem viele Funktionen aus anderen Bibliotheken. In einigen Fällen werden diese benötigt, da z. B. ihr Gradient bestimmt werden muss. Dies ist mit anderen Implementierungen nicht immer möglich [13]. Zum Erstellen von Graphen wird die Bibliothek **Matplotlib** verwendet. Sie stellt viele Methoden zur Verfügung, um Daten schnell und einfach zu visualisieren und ist die populärste Bibliothek zur Visualisierung von Daten in Python [19]. **Pandas** wird genutzt, um leicht mit Textdateien zu interagieren und Informationen über diese zu ermitteln [11]. Die Bibliothek **NumPy** wird eingesetzt, um Matrizenrechnung durchzuführen. Sie stellt viele Funktionen aus der linearen Algebra, aber auch Funktionen wie die Schnelle Fouriertransformation (FFT) zur Verfügung [10]. **SciPy** stellt viele Funktionen zur Verfügung. In dieser Arbeit wird die Bibliothek z. B. für Interpolation oder die Verarbeitung von Signalen genutzt [16]. Des Weiteren wird die bereits erwähnte FROG-Rekonstruktionssoftware von Swamp-Optics genutzt, um die Vorhersageergebnisse zu validieren.

1.4 Aufbau der vorliegenden Arbeit

Die vorliegende Arbeit ist wie folgt strukturiert. Zunächst sollen in Kapitel 2 einige zum Verständnis nötigen Grundlagen der Laserphysik dargelegt werden. Hier werden außerdem einige relevante Funktionen, z. B. zum Erstellen eines FROG-Traces erörtert. Anschließend werden in Kapitel 3 Deep Learning Ansätze zur Rekonstruktion von Zeitprofilen aus Spektrogrammen erläutert. Hierzu wird die Struktur der gewählten Modellarchitektur thematisiert sowie der Datensatz und seine Vorverarbeitung beschrieben. Außerdem werden die Variationen des Trainingsvorgangs und die im Training verwendete Verlustfunktion erklärt. Kapitel 4 zeigt die Ergebnisse der Trainingsvorgänge auf. Zuletzt fasst Kapitel 5 ein Fazit und gibt einen Ausblick, wie die Untersuchung weitergeführt werden kann.

2 Ultrakurze Laserpulse

2.1 Fourierlimitierte Laserpulse und Dispersion

2.1.1 Fourierlimitierte Laserpulse

Ein Laserpuls sei gegeben. Dieser emittiert Licht von nahezu einer Wellenlänge. Hierbei handelt es sich jedoch um eine starke Vereinfachung. Bei der Fouriertransformation des Laserpulses wird deutlich, dass auch dieser eine spektrale Bandbreite hat. [12, S. 2.2]

Der gegebene Puls besitze eine feste Beziehung zwischen den Phasen der einzelnen Frequenzanteile. Da c_n eine für jede Frequenz spezifische Konstante ist, muss die Ableitung der Phase des Laserpulses ϕ frequenzunabhängig sein. Ist die negative Gruppenlaufzeit des Laserpulses τ_{gr} frequenzunabhängig, wie in Gleichungen 2.1 und 2.2 gezeigt, so tritt im Mittel konstruktive Interferenz auf und die einzelnen Frequenzkomponenten treffen gleichzeitig ein. Man spricht von einem fourierlimitierten Laserpuls. Dieser ist zeitlich so kurz, wie es seine Bandbreite zulässt. [12, S. 2.4 ff.]

$$\phi(\omega) = c_n \cdot \omega \quad (2.1)$$

$$-\tau_{gr} = \frac{d\phi(\omega)}{d\omega} = c_n \quad (2.2)$$

2.1.2 Dispersion

Licht hat in jedem Medium eine andere Geschwindigkeit. Dividiert man die Lichtgeschwindigkeit im Vakuum c_0 durch die Lichtgeschwindigkeit in einem Medium c , so erhält man den refraktiven Brechungsindex n_{refrak} . Dies ist in Gleichung 2.3 dargestellt. In der Regel gilt $c < c_0$ und damit $n_{refrak} > 1$. Außerdem ist der Brechungsindex eine Funktion der Frequenz $n_{refrak} = n_{refrak}(\omega)$. [12, S. 3.1 f.]

$$n(\omega) = \frac{c_0}{c} \quad (2.3)$$

$$c(\omega) = \frac{c_0}{n_{refrak}(\omega)} \quad (2.4)$$

Dies ist die physikalische Grundlage für die sogenannte chromatische Dispersion. Wenn Licht durch ein Material durchdringt, bewegen sich seine verschiedenen Farbanteile mit unterschiedlicher Geschwindigkeit. Reisen die hochfrequenten Anteile langsamer durch das Material, spricht man von normaler Dispersion. Reisen sie jedoch schneller, spricht man von anomaler Dispersion. Beide Effekte führen zu einer Verbreiterung des Laserpulses. Um die Geschwindigkeit einer bestimmten Lichtfrequenz in einem Medium zu beschreiben, verwendet man die sogenannte Phasenlaufzeit. Diese kann wie in Gleichung 2.4 bestimmt werden. Im Gegensatz zur Gruppenlaufzeit stellt sie nicht die Geschwindigkeit des gesamten Pulses dar, sondern die einer einzelnen Frequenz. [12, S. 3.2]

Kurzzeitlaser haben aufgrund ihrer kurzen Dauer eine große Bandbreite. Dies führt zu einer wesentlich höheren Empfindlichkeit für Dispersionseffekte als bei regulären Laserpulsen. Wenn ein Laserpuls durch ein Medium mit normaler Dispersion geschickt wird, werden sowohl die niedrigen roten als auch die hohen blauen Frequenzanteile gebremst. Hiervon sind die blauen Frequenzanteile stärker betroffen. Da die niedrigen Frequenzen schneller ankommen, konzentrieren sich die roten Frequenzanteile am Anfang des Pulses. Blaue Frequenzanteile hingegen bewegen sich langsamer, da sie eine höhere Frequenz haben. Sie konzentrieren sich am Ende des Laserpulses. Hier ist deutlich zu erkennen, dass die Farben des Pulses nicht mehr gleichzeitig eintreffen und es zu einer Verbreiterung des Impulses kommt, wie in Abbildung 2.1 dargestellt. Da in diesem Fall die Frequenz mit der Zeit zunimmt, spricht man von einem gechirpten Puls. [12, S. 3.3 ff.]

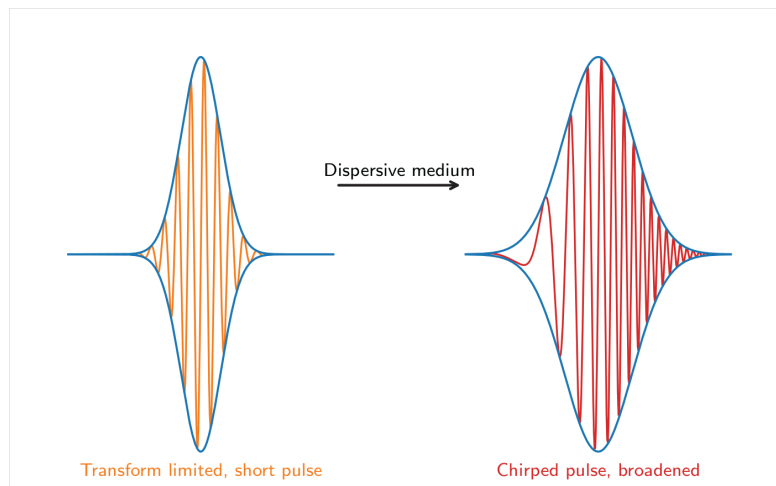


Abbildung 2.1: Schematische Darstellung von Dispersionseffekten. Ein fourierlimitierter Laserpuls (orange) wandert durch ein dispersives Medium und ist danach verbreitert und gechirpt (rot)

Quelle: [12, S. 3.4]

In Gleichung 2.5 wird das elektrische Feld eines Laserpulses dargestellt, wobei es sich im Wesentlichen um die Fouriertransformation des Laserpulses im Frequenzbereich $\varepsilon(\omega)$ handelt. Diese Darstellung wird gewählt, da jede Frequenz des Pulses durch den refraktiven Index n_{refrak} eine unterschiedliche spektrale Phase $\phi(\omega)$ akkumuliert. Dies ist in Gleichung 2.6 durch die Multiplikation mit dem Faktor $e^{j\phi\omega}$ dargestellt. Zur Untersuchung des Effekts der spektralen Phase wird die Taylor-Reihe um die Frequenz ω_0 entwickelt, wie in Gleichung 2.7 gezeigt. [12, S. 3.6 f.]

$$E(t) = \frac{1}{\sqrt{2}} \int_{-\infty}^{\infty} \varepsilon(\omega) e^{j\omega t} d\omega \quad (2.5)$$

$$E_{out}(t) = \frac{1}{\sqrt{2}} \int_{-\infty}^{\infty} \varepsilon(\omega) e^{j(\omega t + \phi(\omega))} d\omega \quad (2.6)$$

$$\phi(\omega) = \phi(\omega_0) + \left. \frac{\partial \phi}{\partial \Omega} \right|_{\omega=\omega_0} (\omega - \omega_0) + \frac{1}{2} \left. \frac{\partial^2 \phi}{\partial \Omega^2} \right|_{\omega=\omega_0} (\omega - \omega_0)^2 + \dots \quad (2.7)$$

Im Folgenden werden die Effekte der Terme nullter bis dritter Ordnung beschrieben, während sich Terme höherer Ordnung vernachlässigen lassen. Der Term nullter Ordnung $\phi(\omega_0) = \phi^{(0)}$ stellt die Phase dar, welche sich über der mittleren Frequenz ω_0 akkumuliert hat. Sie führt zur Hüllkurve des Pulses. Der Term erster Ordnung $\left. \frac{\partial \phi}{\partial \omega} \right|_{\omega=\omega_0} = \phi^{(1)}$ stellt die Gruppenlaufzeit τ_{gr} dar, welche angibt, wie lange ein Puls benötigt, um das dispersive Medium zu durchdringen. Der Term zweiter Ordnung $\left. \frac{\partial^2 \phi}{\partial \omega^2} \right|_{\omega=\omega_0} = \phi^{(2)}$ ist der Term niedrigster Ordnung, der zu einer zeitlichen Verbreiterung des Pulses durch Dispersion führt. Dieser Effekt wird als Gruppenlaufzeitdispersion (GDD) bezeichnet. Die Existenz dieses Terms impliziert, dass die Phasendifferenz zwischen den Frequenzanteilen von der Frequenz abhängt. Dies bedeutet, dass es sich nicht um einen fourierlimitierten Puls handelt. Terme höherer Ordnung tragen ebenfalls zur zeitlichen Verbreiterung des Pulses bei, jedoch in einem deutlich geringeren Maße. [12, S. 3.7]

2.1.3 Das TBD

Die in dieser Arbeit diskutierten Laserpulse können gut als Gausspulse approximiert werden. Die zeitliche Ausbreitung eines solchen Pulses wird über den so genannten Full-Width-at-Half-Maximum (FWHM)-Wert $\Delta\tau_{FWHM}$ beschrieben. Er gibt die zeitliche Differenz zwischen dem ersten und letzten Punkt des Pulses bei der Hälfte seiner maximalen Amplitude an. Die spektrale Bandbreite des Pulses wird über den FWHM-Wert von dessen Spektrum $\Delta\omega_{FWHM}$ definiert. Er beschreibt die Differenz zwischen der ersten und letzten Frequenz des Pulses, der die der Hälfte seiner maximalen Amplitude

groß ist. Die Bestimmung dieser Werte wird in Abbildung 2.2 grafisch dargestellt. In Listing 1 wird eine Pythonfunktion dargestellt, die den FWHM-Wert eines Signals berechnet. In den Zeilen 3 und 5 wird zunächst die maximale Amplitude und dann deren Hälfte bestimmt. Die Bestimmung der Nullstellen erfolgt in Zeile 8 über eine Funktion, die im Anhang in Listing 23 dargestellt ist. Im Fall von mehreren Nullstellen wird die Differenz aus der kleinsten und der größten Nullstelle ermittelt und als FWHM-Wert zurückgegeben.

```
1 def calcFWHM(signal, axis):
2     # find the maximum value of signal
3     max_signal = np.max(signal.numpy())
4     # subtract the half of the maximum from yd
5     signal_half_maximum = signal - max_signal / 2
6
7     # get the zero crossings relative to half its maximum
8     zero_crossings = zeroCrossings(axis, signal_half_maximum)
9
10    # if there are less than 2 zero crossings, return -1 and print a message
11    if len(zero_crossings) < 2:
12        logger.info("FWHM cannot be calculated, since there are fewer than 2
13            ↪ half-maximum points")
14        return -1
15    else:
16        # calculate the full width half maximum as the difference between the highest
17            ↪ and lowest zero crossing
18        fwhm = np.max(zero_crossings) - np.min(zero_crossings)
19        return fwhm
```

Quelle: Eigene Darstellung, Auszug aus `./modules/preprocessing.py`

Zusätzlich werden noch die Variablen $\Delta\tau_{RMS}$ und $\Delta\omega_{RMS}$ definiert. Sie stellen das Moment zweiter Ordnung dar, also die Streuung der Varianz, in Bezug auf die mittlere Ankunftszeit oder -frequenz des Pulses. Hierbei wird immer der auf Eins normalisierte Puls betrachtet. Ihre vorrangige Verwendung liegt in der theoretischen Betrachtung. Ihre Herleitung ist in den Gleichungen 2.9 und 2.11 abgebildet. Das Produkt von $\Delta\tau_{FWHM}$ und $\Delta\omega_{FWHM}$, wie in Gleichung 2.8 dargestellt, wird als TBD bezeichnet. Die Multiplikation von $\Delta\tau_{RMS}$ und $\Delta\omega_{RMS}$ führt zu einem TBD_{RMS} -Wert der stets größer oder

gleich 0,25 ist. Die Ermittlung von TBD_{RMS} ist in Gleichung 2.13 veranschaulicht. Das TBD stellt folglich ein Maß für die Komplexität des Pulses dar, d. h. ein Puls mit geringem TBD ist weniger komplex als ein Puls mit hohem TBD. Während die regulären FWHM-Werte Teile des Pulses unter der Hälfte der Amplitude vollkommen ignorieren, werden diese von den Root-Mean-Square (RMS)-Werten berücksichtigt. Dies macht sie zu einem guten Indikator bei Pulsen, die eine hohe Breite unterhalb der halben Amplitude aufweisen. [20, S. 29 ff.]

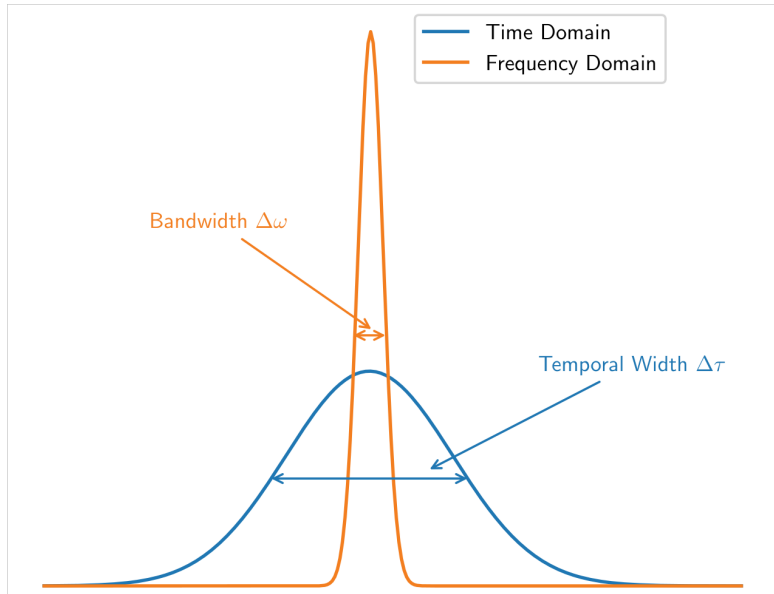


Abbildung 2.2: Zeitliche Breite und spektrale Bandbreite eines Gausspuls im Zeit- (blau) und Frequenzbereich (orange)

Quelle: [12, S. 2.2]

$$TBD = \Delta\tau_{FWHM} \Delta\omega_{FWHM} \quad (2.8)$$

$$\Delta\tau_{RMS} = \left\langle t - \langle t \rangle^2 \right\rangle = \langle t^2 \rangle - \langle t \rangle^2 \quad (2.9)$$

$$\text{mit: } \langle t^n \rangle = \int_{-\infty}^{\infty} t^n I(t) dt \quad (2.10)$$

$$\Delta\omega_{RMS} = \left\langle \omega - \langle \omega \rangle^2 \right\rangle = \langle \omega^2 \rangle - \langle \omega \rangle^2 \quad (2.11)$$

$$\text{mit: } \langle \omega^n \rangle = \int_{-\infty}^{\infty} \omega^n I(\omega) d\omega \quad (2.12)$$

$$TBD_{RMS} = \Delta\tau_{RMS} \Delta\omega_{RMS} \quad (2.13)$$

2.2 Autokorrelation

In diesem Abschnitt wird die Charakterisierung eines Laserpulses mithilfe der Autokorrelation erörtert, da diese die Grundlage für das Erstellen von FROG-Traces bildet. Zur Messung eines Ereignisses wird ein mindestens genau so kurzes Ereignis benötigt. Da Femtosekunden-Laserpulse jedoch zu den kürzesten Ereignissen überhaupt gehören, lassen sie sich nicht auf konventionelle Arten, z. B. durch elektrische Bauteile messen. Dies liegt daran, dass die Laufzeiten der Bauteile deutlich größer sind als die betrachteten Pulse selbst. Daher werden im Folgenden Verfahren verwendet, bei denen ein Puls selbst zu seiner Messung verwendet wird. [12, S. 6.3 ff.]

Zur Messung der Intensität des Pulses, definiert als das Quadrat der Amplitude $I = A^2$, des Pulses über die Zeit, wird die Autokorrelation der Intensität $A^{(2)}(\tau)$ genutzt. Zu diesem Zweck wird der Puls $E(t)$ mittels eines Beam Splitters in zwei Pulse aufgeteilt. Einer dieser beiden Pulse wird dabei mit einer variablen Verzögerung τ relativ zum anderen Puls verzögert. Im Anschluss werden der ursprüngliche Puls $E(t)$ und der verzögerte Puls $E(t - \tau)$ in einem nicht-linearen optischen Medium, welches unmittelbar auf die Einwirkung reagiert, überlagert. In der vorliegenden Arbeit wird ein SHG-Kristall verwendet, welcher zusätzlich die Frequenz des Laserpulses verdoppelt. Abschließend wird das resultierende Licht mithilfe eines Detektors gemessen. Der beschriebene Versuchsaufbau wird in Abbildung 2.3 grafisch dargestellt. Wie in Gleichung 2.14 dargestellt, ist das entstehende E-Feld proportional zum Produkt der zueinander verzögerten E-Felder. Die Intensität des Feldes ist proportional zum Produkt der Intensitäten der beiden Pulse, wie in Gleichung 2.15 gezeigt. [20, S. 65]

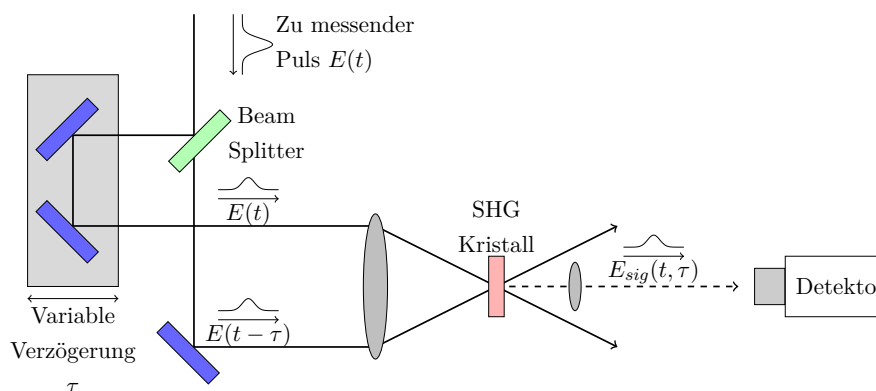


Abbildung 2.3: Experimenteller Aufbau der Messung der Intensitätsautokorrelation mit SHG-Kristall

Quelle: Eigene Darstellung nach [20, S. 65]

$$E_{sig}^{SHG}(t, \tau) \propto E(t) \cdot E(t - \tau) \quad (2.14)$$

$$I_{sig}^{SHG}(t, \tau) \propto I(t) \cdot I(t - \tau) \quad (2.15)$$

Aufgrund der unzureichenden Zeitaufösung von Detektoren zur Messung von $I_{sig}^{SHG}(t, \tau)$, wird das Integral über die Verzögerung τ gemessen. Dieses Messverfahren wird als Intensitätsautokorrelation bezeichnet und ist in Gleichung 2.16 dargestellt. Im Folgenden wird sie als Autokorrelation bezeichnet. [20, S. 66]

Die Autokorrelation erreicht ihr Maximum bei $\tau = 0$, da hier die Überlappung beider Pulse maximal ist. Zudem ist sie achsensymmetrisch um den Punkt $\tau = 0$, wie in den Gleichungen 2.17 und 2.18 gezeigt. [20, S. 67]

$$A^{(2)}(\tau) = \int_{-\infty}^{\infty} I(t) \cdot I(t - \tau) d\tau \quad (2.16)$$

dargestellt mit: $t = t - \tau$

$$A^{(2)}(\tau) = \int_{-\infty}^{\infty} I(t + \tau) \cdot I(t) d\tau \quad (2.17)$$

$$= A^{(2)}(-\tau) \quad (2.18)$$

Die Anwendung des in Gleichung 2.19 dargestellten Autokorrelationstheorems ergibt, wie in Gleichung 2.21 dargestellt, dass die Fouriertransformierte der Autokorrelation dem Betragsquadrat der Intensität entspricht. Daraus folgt, dass die Fouriertransformierte der Autokorrelation realwertig und positiv ist. Es konnte jedoch gezeigt werden, dass sich aus der Fouriertransformierten der Autokorrelation die Intensität und Phase des Signals nicht zurückgewinnen lassen. [20, S. 67]

$$|\tilde{E}(\omega)|^2 = \mathcal{F} \left\{ \int_{-\infty}^{\infty} E(t) \cdot E^*(t - \tau) d\tau \right\} \quad (2.19)$$

$$|\tilde{I}(\omega)|^2 = \mathcal{F} \left\{ \int_{-\infty}^{\infty} I(t) \cdot I^*(t - \tau) d\tau \right\} \quad (2.20)$$

$$\tilde{A}^{(2)}(\omega) = |\tilde{I}(\omega)|^2 \quad (2.21)$$

2.3 FROG

Die Rückgewinnung von Intensität und Phase wird durch das so genannte FROG-Verfahren erreicht. Dieses unterscheidet sich von der Autokorrelation dadurch, dass nicht nur die Intensität über die Zeit, sondern die instantane Frequenz über die Zeit gemessen wird. Infolgedessen ergibt sich ein Spektrogramm $\Sigma_g(\omega, \tau)$. Die Spalten eines Spektrogramms sind, wie in Gleichung 2.22 dargestellt, das Spektrum des Produkts aus Signal $E(t)$ und einer Gate-Funktion $g(t - \tau)$ bei jedem Verzögerungswert τ . [20, S. 102]

$$\Sigma_g^E(\omega, \tau) = \left| \int_{-\infty}^{\infty} E(t) \cdot g(t - \tau) \cdot e^{-j\omega t} dt \right|^2 \quad (2.22)$$

Da für die Gate-Funktion ein Ereignis benötigt wird, dessen Dauer höchstens der des Signals selbst entspricht, soll im Folgenden analog zur Autokorrelation das Signal mit sich selbst gegatet werden. Anstatt lediglich die Intensität des Signals zu messen, wird die instantane Frequenz mit einem Spektrometer am Ende des Messaufbaus gemessen, wie in Abbildung 2.4 dargestellt ist. [20, S. 103 f.]

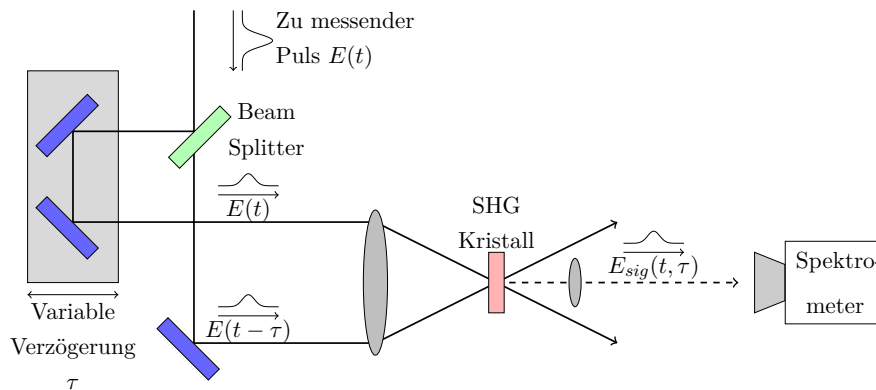


Abbildung 2.4: Experimenteller Aufbau der Messung eines FROG-Traces mit SHG-Kristall

Quelle: Eigene Darstellung nach [20, S. 127]

Wird erneut ein SHG-Kristall verwendet, ergibt sich somit der FROG-Trace $I_{FROG}^{SHG}(\omega, \tau)$ gemäß Gleichung 2.23. Der SHG-FROG weist im Vergleich zu anderen FROG-Verfahren den Vorteil einer hohen Sensitivität auf, sodass Pulse mit einer Amplitude von bis zu 1 pJ gemessen werden können. Darüber hinaus nimmt der Signal Rausch Abstand (SNR) beim SHG-FROG den geringsten Wert an, da die Frequenzverdopplung durch den SHG-Kristall dazu führt, dass sich die Ursprungssignale herausfiltern lassen. Ein signifikanter

Nachteil besteht jedoch darin, dass das resultierende Spektrogramm wie bei der Autokorrelation achsensymmetrisch um den Verzögerungspunkt $\tau = 0$ liegt. Dies bedeutet, dass ein Puls zu demselben FROG-Trace führt wie seine zeitliche Spiegelung. [20, S. 127 f.]

$$I_{FROG}^{SHG}(\omega, \tau) = \left| \int_{-\infty}^{\infty} E(t) \cdot E(t - \tau) \cdot e^{-j\omega t} dt \right|^2 \quad (2.23)$$

Im Gegensatz zur Autokorrelation sind FROG-Traces, abgesehen von den in Kapitel 2.5 erläuterten Ambiguitäten, eindeutig bestimmbar. Das impliziert, dass sowohl die Intensität als auch die Phase des E-Feldes nahezu eindeutig rekonstruiert werden können. Die erläuterten Ambiguitäten besitzen in der Optik keine physikalische Bedeutung und können daher entfernt werden. [20, S. 105 f.]

Im Listing 2 ist der verwendete Code zum Erstellen der Spektrogramme aus dem analytischen Signal angegeben. In Zeile 14 wird ein Vektor erstellt, der Indizes zwischen $-\frac{N}{2}$ und $\frac{N}{2}$ umfasst, wobei N die Größe des Spektrogramms in beide Dimensionen beschreibt. Dieser wird mit *delta_tau*, welches der Sampling-Frequenz entspricht, multipliziert, um eine Zeitachse zu erzeugen. In Zeile 20 wird ein Tensor erstellt, welcher die in Gleichung 2.23 gezeigte Exponentialkomponente $e^{-j\omega t}$ repräsentiert. Da es durch den SHG-Kristall zu einer Frequenzverdopplung kommt, wird in dieser Zeile auch die Kreisfrequenz verdoppelt. In Zeile 23 wird ein leerer Tensor der Größe $N \times N$ erzeugt, welcher das Spektrogramm initiiert. Ab Zeile 26 beginnt eine Schleife, welche über die verschiedenen Verzögerungswerte und die Spalten des Spektrogramms iteriert. Da die Spalten *matrix_index* des Spektrogramms die Autokorrelation des Signals zu einer definierten Verzögerung *delay_index* repräsentieren, wird diese nun für jede der Spalten berechnet. In Zeile 28 wird hierzu zunächst ein Ringbuffer genutzt, um das Signal um *delay_index* Stellen zu verschieben. Dieses Signal wird mit dem Ursprungssignal und dem oben erzeugten *shift_factor* multipliziert. In Zeile 32 wird anschließend das resultierende Signal mithilfe der schnellen Fouriertransformation in die Frequenzebene transformiert, um die Autokorrelation zu erzeugen. Die Funktion *fftshift* wird genutzt, um die korrekte Reihenfolge der Frequenzwerte zu bestimmen. Zu diesem Zeitpunkt stellt das Spektrogramm jedoch die komplexe Amplitude des Signals dar. Um die Intensität zu erhalten, wird zunächst elementweise der Betrag des Spektrogramms gebildet und dieser anschließend quadriert.

```
1 def createSHGmat(analytical_signal, delta_tau, wCenter):
2     # get correct device
3     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
4     # get the amount of samples of the analytical_signal and rename it
5     N = len(analytical_signal)
6     E = analytical_signal.to(device)
7
8     # double the frequency because of the SHG-matrix
9     wCenter2 = 2 * wCenter
10
11    # create a tensor storing indices -N/2 to N/2
12    start = -N // 2
13    end = N // 2
14    delay_index_tensor = torch.arange(start, end, dtype=torch.float32).to(device)
15
16    # create time tensor (t)
17    time_axis = delta_tau * delay_index_tensor
18
19    # calculate shift factor  $e^{-j 2\omega_c t}$ 
20    shift_factor = torch.exp(-1j * wCenter2 * time_axis).to(device)
21
22    # initialize empty SHG-matrix
23    shg_matrix = torch.zeros((N, N), dtype=torch.complex128).to(device)
24
25    # increment over the
26    for (matrix_index, delay_index) in enumerate(delay_index_tensor):
27        # Shift E-Field for the current delay index
28        E_shifted = helper.circshift(E, delay_index)
29        # Calculate the the argument of the fft  $E(t)E(t-\tau)*shift\_factor$ 
30        argument = E * E_shifted * shift_factor
31        # current matrix index is the shifted fft of  $E(t)E(t-\tau)*shift\_factor$ 
32        shg_matrix[matrix_index, :] = fftshift(fft(argument))
```

33

34 `return shg_matrix`

Listing 2: Funktion zum Erstellen einer SHG-Matrix aus dem analytischen Signal, der Differenz zwischen den Verzögerungswerten $\Delta\tau$ und der Mittenfrequenz

Quelle: Eigene Darstellung, Auszug aus `./modules/loss.py`

Das resultierende Spektrogramm weist eine Verzögerungs- und eine Frequenzachse auf. Hierbei ist anzumerken, dass die vorliegenden Daten anstatt einer Frequenzachse eine Wellenlängenachse besitzen, sodass eine entsprechende Konvertierung des Spektrogramms erforderlich ist. Zu diesem Zweck wird Gleichung 2.25 verwendet, wobei der resultierende Code in Listing 3 dargestellt ist. Es wird hierbei genutzt, dass das Spektrogramm als Anreihung der Autokorrelation für jeden Verzögerungswert aufgefasst werden kann. Der Funktion werden sowohl das Spektrogramm als auch die zugehörige Frequenzachse übergeben. Zunächst wird in Zeile 13 die Frequenzachse durch den in Gleichung 2.24 dargestellten Zusammenhang in eine Wellenlängenachse konvertiert. Da der Abstand zwischen den Wellenlängenwerten $\Delta\lambda$ jedoch konstant sein soll, werden in den folgenden Zeilen zunächst der kleinste und größte Wellenlängenwert bestimmt. Anschließend wird ein Tensor erstellt, der die äquidistante Wellenlängenachse darstellt.

$$\lambda = \frac{2\pi c_0}{\omega} \quad \text{mit } c_0 = 299792458 \frac{m}{s} \quad (2.24)$$

$$S_\lambda = S_\omega \left(\frac{2\pi c_0}{\lambda} \right) \frac{2\pi c_0}{\lambda^2} \quad (2.25)$$

Ab Zeile 21 wird eine Iteration über die Zeilen des Spektrogramms vorgenommen, wobei jede dieser Zeilen, wie in Gleichung 2.25 dargestellt, in den Wellenlängenbereich konvertiert wird. Dies ist in Zeile 23 des Listings veranschaulicht. [20, S. 15]

Im Anschluss werden die Punkte mithilfe der zuvor erstellten Wellenlängen- und Ursprungsfrequenzachse interpoliert, sodass das resultierende Spektrogramm equidistant auf der Wellenlängenachse ist. Die Funktion zur Interpolation wird in Kapitel 3.2.2 genauer beschrieben. Ein resultierendes Beispielspektrogramm ist in Abbildung 2.5 gezeigt.

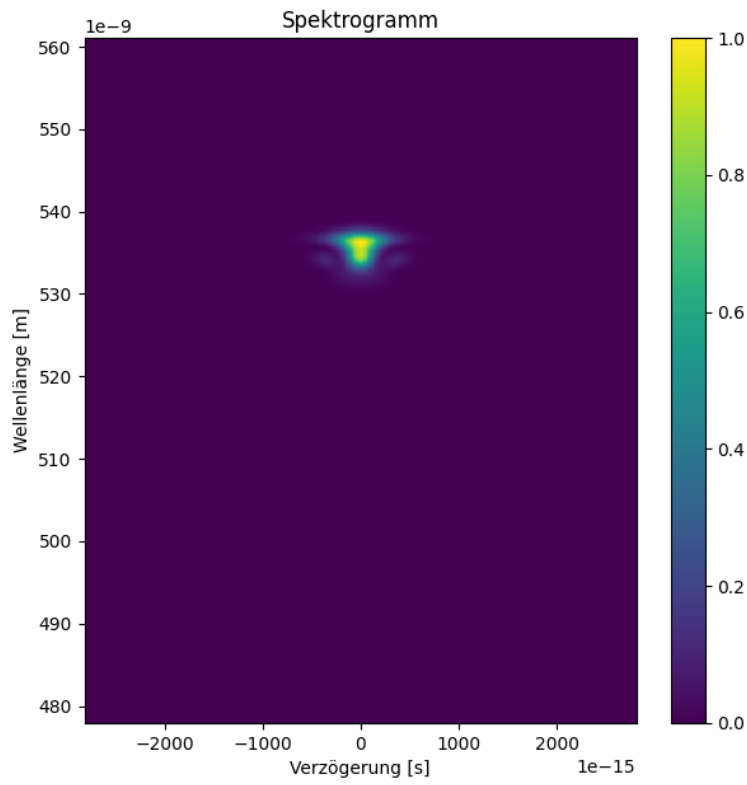


Abbildung 2.5: SHG-FROG Spektrogramm mit Verzögerungs- und Wellenlängenachse
Quelle: Eigene Darstellung

```
1 def intensityMatrixFreq2Wavelength(frequency_axis, freq_intensity_matrix):
2     device = freq_intensity_matrix.device
3     frequency_axis = frequency_axis.to(device)
4
5     length = len(frequency_axis)
6     # print(f"length = {length}")
7     assert length == freq_intensity_matrix.size(1)
8
9     # initialize output matrix with zeros in the shape of the input matrix
10    wavelength_intensity_matrix = torch.zeros_like(freq_intensity_matrix)
11
```

```

12     # calculate the wavelength_axis and flip it for decending order
13     wavelength_axis = c.c2pi / frequency_axis.flip(0)
14     # get the equidistant wavelength axis
15     wavelength_min = wavelength_axis[0]
16     wavelength_max = wavelength_axis[-1]
17     wavelength_axis_equidistant = torch.linspace(wavelength_min, wavelength_max,
18     ↪     length)
19
20     # calculate wavelength intensity matrix (2.17 Trebino)
21     # itterate over each row
22     for i, Sw in enumerate(freq_intensity_matrix):
23         # Element wise operation (Sw .* wavelength_axis.^2 / c2p)
24         Sl = torch.flip(Sw.to(device) * (frequency_axis ** 2) / c.c2pi, dims=[0])
25
26         # perform linear interpolation
27         interpolated = piecewiseLinearInterpolation(wavelength_axis, Sl,
28         ↪         wavelength_axis_equidistant)
29
30         # assign interpolated values to freq_intensity_matrix
31         wavelength_intensity_matrix[i, :] = interpolated
32
33     return wavelength_axis_equidistant, wavelength_intensity_matrix

```

Listing 3: Funktion zum Konvertieren einer SHG-Matrix mit einer Frequenzachse zu einer SHG-Matrix mit einer Wellenlängenachse
 Quelle: Eigene Darstellung, Auszug aus `./modules/loss.py`

Damit ein abgetastetes Signal fehlerfrei wiederhergestellt werden kann, müssen mindestens zwei Abtastpunkte je Periode des Signals existieren. Die Abtastfrequenz muss also größer als die doppelte Frequenz des Signals sein. Man spricht von der Nyquistfrequenz. Für die Rückgewinnung eines Laserpulses aus einem Spektrogramm gilt hingegen ein noch strengeres Kriterium. So muss der Trace inmitten von Nullen liegen, damit er rekonstruiert werden kann. Da dies streng genommen nie der Fall ist, werden alle Intensitätswerte unter 10^{-4} als Null betrachtet. Ist dies der Fall, spricht man davon, dass die FROG-Abtastrate (FSR) erfüllt wird. Wird ein Signal mit der Nyquistfrequenz abgetastet, hat das hieraus resultierende Spektrogramm bis zu seinem Rand signifikante

Intensitäten. Somit muss die FSR immer über der Nyquistfrequenz eines Signals liegen. Damit sie eingehalten wird, müssen die in Gleichung 2.26 und 2.27 gezeigten Kriterien erfüllt sein. [20, S. 212 ff.]

$$\delta\tau \leq \frac{\lambda_c^2}{6,3c_0\Delta\lambda_{FWHM}} \quad (2.26)$$

$$\delta\tau \geq \frac{4,5\Delta\tau_{FWHM}}{N} \quad (2.27)$$

2.4 Phase Retrieval und FROG-Fehler

Die Rekonstruktion des komplexen E-Feldes $E(t)$ aus einem FROG-Trace $I_{FROG}(\omega, \tau)$ wird Phase Retrieval genannt und i. d. R. durch verschiedene sogenannte FROG Algorithmen erreicht. Zunächst wird ein E-Feld $E(t)$ geraten. Anschließend wird dieses mit der entsprechenden Gleichung 2.23 in ein Signalfeld $E_{sig}(t, \tau)$ transformiert. Im Anschluss wird eine Fouriertransformation durchgeführt, um das Signalfeld in den Frequenzbereich zu wandeln. Aus diesem Signalfeld $\tilde{E}'_{sig}(\omega, \tau)$ wird durch Vergleich mit dem gemessenen FROG-Trace $I_{FROG}(\omega, \tau)$ ein verbessertes $\tilde{E}'_{sig}(\omega, \tau)$ erstellt, wobei sich die Vorgehensweise je nach gewähltem Algorithmus unterscheidet. Im Anschluss wird $\tilde{E}'_{sig}(\omega, \tau)$ invers fouriertransformiert, um zu einem neuen komplexen $E'_{sig}(t, \tau)$ zu gelangen. Daraus wird nun ein neues E-Feld $E(t)$ berechnet und eine neue Iteration beginnt. Dabei besteht das Ziel darin mit jeder Iteration des Algorithmus näher an das tatsächliche E-Feld zu gelangen. Dieser Vorgang ist in Abbildung 2.6 schematisch dargestellt.

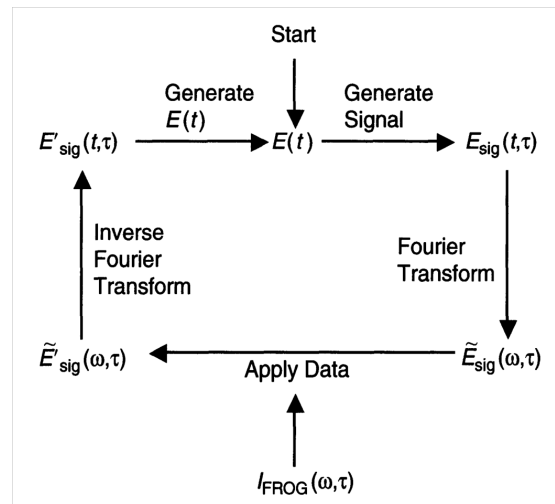


Abbildung 2.6: Schematische Darstellung eines Algorithmus zum Phase Retrieval
Quelle: [20, S. 158]

Die Güte einer solchen Rekonstruktion wird durch den sogenannten FROG-Fehler bewertet. Er beschreibt die MSE zwischen dem gemessenen FROG-Trace I_{FROG} und der k -ten Iteration des rückgewonnenen FROG-Traces $I_{FROG}^{(k)}$ [20, S. 160]. Die Konstante μ wird verwendet, um diesen Fehler möglichst klein zu halten und wird wie in Gleichung 2.29 ermittelt [20, S. 160][2, S. 497]. Dieser ist für die k -te Iteration wie in Gleichung 2.28 definiert. Der Vergleich zweier Spektrogramme ist in Abbildung 2.7 abgebildet. [20, S. 183 ff.]

$$G^{(k)} = \sqrt{\frac{1}{N \cdot M} \sum_{i,j=1}^{M,N} |I_{FROG}(\omega_i, \tau_j) - \mu I_{FROG}^{(k)}(\omega_i, \tau_j)|^2} \quad (2.28)$$

$$\mu = \frac{\sum_{i,j=1}^{M,N} I_{FROG}(\omega_i, \tau_j) \cdot I_{FROG}^{(k)}(\omega_i, \tau_j)}{\sum_{i,j=1}^{M,N} I_{FROG}^{(k)}(\omega_i, \tau_j)^2} \quad (2.29)$$

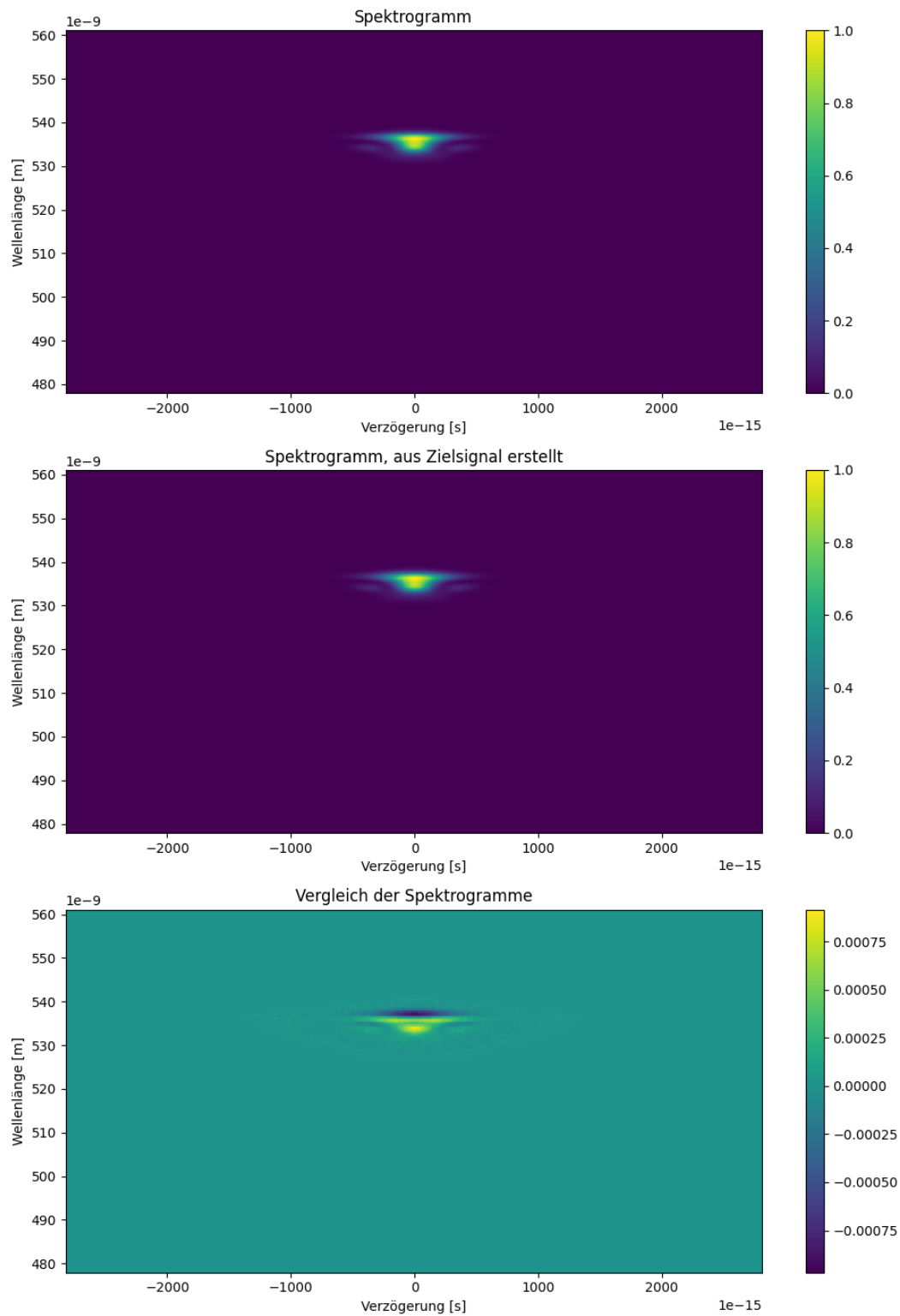


Abbildung 2.7: Vergleich zweier Spektrumgramme. Das zweite Spektrumgramm wurde aus dem Zielsignal des ersten erzeugt

Quelle: Eigene Darstellung

Die Implementierung dieser Gleichungen in eine Python-Funktion wird in Listing 4 gezeigt. Die Variablen I_k und I_m entsprechen hierbei $I_{FROG}^{(k)}$ und I_{FROG} . In den Zeilen 10 bis 12 wird zunächst, analog zu Gleichung 2.29, der Faktor μ bestimmt. In Zeile 17 erfolgt die Berechnung der MSE für jeden Punkt der Spektrogramme unter Berücksichtigung von μ . Anschließend werden die einzelnen MSE-Werte addiert und mit dem Kehrwert der Punktzahl multipliziert, um den Fehler je nach Größe der Spektrogramme zu gewichten. Aus dem skalaren Wert, der sich daraus ergibt, wird anschließend die Wurzel gezogen, um den FROG-Fehler zu erhalten.

```
1 def calcFrogError(I_k, I_m):
2     # get correct device
3     device = I_k.device
4     # ensure all tensors are on the same device
5     I_m.to(device)
6     # get the shape of the measured spectrogram
7     M, N = I_m.shape
8
9     # calculate \mu [pypret gl. 13 (s. 497)]
10    mu_numerator = torch.sum(I_m * I_k)
11    mu_denominator = torch.sum(I_m * I_k)
12    mu = mu_numerator / mu_denominator
13
14    # calculate normalization factor
15    norm_factor = 1 / (M * N)
16    # calculate the magnitude squared difference between the spectrograms
17    mag_squared_difference = torch.abs(I_m - mu*I_k)**2
18    # calculate the FROG-Error
19    frog_error = torch.sqrt(norm_factor * torch.sum(mag_squared_difference))
20
21    return frog_error
```

Listing 4: Funktion zur Bestimmung des FROG-Fehlers aus einem gemessenen und einem vorhergesagten Spektrogramm

Quelle: Eigene Darstellung, Auszug aus `./modules/loss.py`

2.5 Triviale Ambiguitäten

Die vorliegende Arbeit befasst sich mit dem Problem, mithilfe eines neuronalen Netzes einen Puls aus dem SHG-FROG-Trace zu rekonstruieren. Aufgrund sogenannter trivialer Ambiguitäten gibt es unendlich viele komplexe Zeitsignale $E(t)$, die auf das gleiche Spektrogramm führen. Um das Training des neuronalen Netzes für die Pulsrekonstruktion zu ermöglichen, sollten also zunächst diese Ambiguitäten aus den Zielsignalen entfernt werden, die das komplexe Zeitsignal enthalten. Die erste triviale Ambiguität ist eine Translation des Signals, wie in Gleichung 2.30 dargestellt. [20, S. 105]

$$E_{amb1}(t) = E(t - t_0) \tag{2.30}$$

Die Beseitigung dieser Ambiguität erfolgt durch die Zentrierung des Peaks der Intensität des Signals auf einen definierten Zeitpunkt. Die hierzu verwendete Python-Funktion ist in Listing 5 abgebildet. Zunächst wird in Zeile 3 die Intensität des Signals aus dessen Real- und Imaginärteil berechnet. In den Zeilen 6 und 7 wird anschließend der Index des Signaltensors bestimmt, der den höchsten Intensitätswert aufweist. Ein Offset wird definiert, indem der Index des Maximums vom mittleren Index des Tensors subtrahiert wird. In Zeile 10 wird schließlich das komplexe Signal um diesen Offset verschoben, wobei ein Ringbuffer genutzt wird.

```
1 def removeTranslationAmbiguity(complex_signal, center_index):
2     # calculate the intensity of the signal
3     intensity = complex_signal.real**2 + complex_signal.imag**2
4
5     # get the index of the highest intensity value and calculate the offset
6     peak_index = torch.argmax(intensity)
7     offset = center_index - peak_index
8
9     # shift the real and imaginary parts to center the peak
10    complex_signal_noambig = torch.roll(complex_signal, offset.item() )
11
12    return complex_signal_noambig
```

Quelle: Eigene Darstellung, Auszug aus `./modules/helper.py`

Die zweite triviale Ambiguität manifestiert sich bei der Spiegelung und anschließenden Konjugation des komplexen Signals, wie in Gleichung 2.31 beschrieben. Dies resultiert in einer gespiegelten Intensität des Signals $I(-t)$ sowie einer negativen gespiegelten Phase $-\phi(-t)$. [20, S. 105]

$$E_{amb2}(t) = E^*(-t) \quad (2.31)$$

Die zugehörige Funktion ist in Listing 6 dargestellt. Um die Ambiguität zu entfernen, wird zunächst die Summe der Intensitäten auf der linken bzw. rechten Seite des Pulses bestimmt. Dies ist in den Zeilen 4 und 5 zu sehen. Ist das linke Mittel kleiner als das rechte, wird das Signal komplex konjugiert und gespiegelt. Andernfalls bleibt das originale Signal erhalten. Durch den Vergleich der Summen soll bestimmt werden, auf welcher Seite das Gewicht des Pulses liegt. Durch die Konjugation sollte die linke Seite des Pulses immer den Hauptanteil der Intensität bilden.

```
1 def removeConjugationAmbiguity(complex_signal, center_index):
2     # calculate the intensity of the signal
3     intensity = complex_signal.real**2 + complex_signal.imag**2
4     intensity_sum_left = torch.sum(intensity[:center_index])
5     intensity_sum_right = torch.sum(intensity[center_index:])
6
7     # if the sum is smaller than 0:
8     if intensity_sum_left < intensity_sum_right:
9         # flip and conjugate the signal
10        complex_signal_noambig = torch.flip(complex_signal).conj()
11    else:
12        # do nothing
13        complex_signal_noambig = complex_signal
14
15    return complex_signal_noambig
```

Quelle: Eigene Darstellung, Auszug aus `./modules/helper.py`

Die letzte triviale Ambiguität stellt eine absolute Phasenverschiebung dar, die durch die Multiplikation des komplexen Zeitsignals mit einer absoluten Phasenverschiebung

charakterisiert ist. Diese ist in Gleichung 2.32 veranschaulicht. [20, S. 105]

$$E_{amb3}(t) = E(t) \cdot e^{j\phi_0} \quad (2.32)$$

Um die Ambiguität zu entfernen wird die Phase des Signals zu einem definierten Zeitpunkt bestimmt und anschließend vom Signal subtrahiert. Die hierzu genutzte Python-Funktion ist in Listing 7 abgebildet. In Zeile 6 wird die Phase in der Mitte des Signals bestimmt und anschließend von der Phase des Signals abgezogen. Abschließend werden in Zeile 14 und 15 aus der so entstehenden neuen Phase der Real- und Imaginärteil bestimmt und diese in Zeile 18 wieder zu einem komplexen Signal zusammengefügt.

```
1 def removePhaseShiftAmbiguity(complex_signal, center_index):
2     # calculate the phase of the signal [rad]
3     phase = torch.angle(complex_signal)
4
5     # get phase at center index
6     center_phase = phase[center_index]
7     # remove the absolute phase shift from the whole phase tensor
8     phase = phase - center_phase
9
10    # reconstruct real and imaginary parts
11    # get the magnitude
12    magnitude = torch.abs(complex_signal)
13    # calculate real part
14    real_part = magnitude * torch.cos(phase)
15    imag_part = magnitude * torch.sin(phase)
16
17    # create a new complex tensor
18    complex_signal_noambig = torch.complex(real_part, imag_part)
19
20    return complex_signal_noambig
```

Quelle: Eigene Darstellung, Auszug aus `./modules/helper.py`

2.6 Hilbert-Transformation

Die Fouriertransformierte $X(\omega)$ eines reellwertigen Signals $x(t)$ ist komplex und achsensymmetrisch zum Punkt $\omega = 0$. Daraus lässt sich ableiten, dass der Informationsgehalt der negativen Frequenzanteile redundant ist. Werden die negativen Frequenzanteile entfernt, die aus der Fouriertransformation folgen, spricht man von einem analytischen Signal. Dieses ist komplexwertig, hat aber weiterhin den gleichen spektralen Informationsgehalt wie das reelle Signal. [9]

Diese Eigenschaft kann genutzt werden, um die Vorhersage des E-Feldes eines Laserpulses zu vereinfachen. Ein reeller Laserpuls $E(t)$ führt über die Fouriertransformation zu einem komplexen Spektrum $E(\omega)$. Dieses erfüllt die Bedingung hermitisch zu sein, wie in Gleichung 2.33 gezeigt ist. Dies bedeutet, die gespiegelt konjugiert Komplexe des Spektrums entspricht dem Spektrum selbst. Daraus folgt, dass zur vollständigen Charakterisierung des E-Feldes eines Laserpulses der positive Teil des Spektrums genügt, wie in Gleichung 2.34 definiert. Wird dieses zurück in den Zeitbereich transformiert, ergibt sich ein komplexes Zeitsignal, welches die gleiche Intensität und Phase wie das ursprüngliche Signal aufweist. Es handelt sich also um ein analytisches Signal. [22, S. 7]

$$\tilde{E}(\omega) = \tilde{E}^*(-\omega) \quad (2.33)$$

$$\tilde{E}^+(\omega) = \begin{cases} \tilde{E}(\omega) & \text{für } \omega \geq 0 \\ 0 & \text{für } \omega < 0 \end{cases} \quad (2.34)$$

Es sei das Spektrum $X(\omega)$ eines reellen Signals $x(t)$ gegeben. Die Bestimmung des analytischen Signals $z(t)$ erfolgt zunächst durch Erzeugung dessen Spektrums $Z\omega$, wie in Gleichung 2.35 gezeigt. Hierbei werden alle negativen Frequenzen zu Null, während die positiven Frequenzanteile die doppelte Amplitude des Ursprungssignals erhalten, um den Energieinhalt zwischen den Signalen konstant zu halten. Wird nun das Spektrum $Z(\omega)$, wie in Gleichung 2.36, invers fouriertransformiert, ergibt sich das in Gleichung 2.37 gezeigte komplexe analytische Signal $z(t)$. [9]

$$X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t} dt$$

$$Z(\omega) = \begin{cases} X(0) & \text{für } \omega = 0 \\ 2X(\omega) & \text{für } \omega > 0 \\ 0 & \text{für } \omega < 0 \end{cases} \quad (2.35)$$

$$Z(\omega) \bullet \longleftrightarrow z(t) \quad (2.36)$$

$$z(t) = z_r(t) + z_i(t) \quad (2.37)$$

Wie in Gleichung 2.38 dargestellt, ist der Realteil des resultierenden Signals identisch zum Ausgangssignal $x(t)$. Der Imaginärteil des Signals wird in Gleichung 2.39 so definiert, dass er der Hilbert-Transformation HT des Ursprungssignals entspricht. [9]

$$z_r(t) = x(t) \quad z_i(t) = ? \quad (2.38)$$

$$z(t) = z_r(t) + z_i(t) = x(t) + j \text{HT}\{x(t)\} \quad (2.39)$$

Bei den gemessenen Signalen handelt es sich um diskrete Signale. Daher wird im Folgenden die Hilbert-Transformation für den diskreten Fall erläutert. Es wird ein reellwertiges Signal $x(t)$ mit einer Frequenz von $\frac{1}{T}$ abgetastet. Daraus folgt ein N-Abtastwerte langes diskretes Signal $x[n] = x(nT)$, dessen Diskrete Fouriertransformation (DTFT) in Gleichung 2.40 abgebildet ist. Es wird, analog zum kontinuierlichen Fall, aus dem diskreten Spektrum ein einseitiges erzeugt. Anschließend wird die inverse DTFT genommen, um das analytische Signal zu erzeugen. Die entsprechenden mathematischen Formeln werden in den Gleichungen 2.41 und 2.42 dargestellt. [9]

$$X(\omega) = T \sum_{n=0}^{N-1} x[n]e^{-j\omega nT} \quad (2.40)$$

$$Z[m] = \begin{cases} X[0] & \text{für } m = 0 \\ 2X[m] & \text{für } 1 \leq m \leq \frac{N}{2} - 1 \\ X[\frac{N}{2}] & \text{für } m = \frac{N}{2} \\ 0 & \text{für } \frac{N}{2} \leq m \leq N - 1 \end{cases} \quad (2.41)$$

$$\text{DTFT}_{\text{inv}}(Z[m]) = z[n] = z_r[n] + jz_i[n] \quad (2.42)$$

Die zur Implementierung der diskreten Hilbert-Transformation verwendete Funktion ist in Listing 8 dargestellt. Zunächst wird in Zeile 3 die FFT des reellen Signals erzeugt. Darauf aufbauend wird eine Maske erstellt, welche die negativen Frequenzkomponenten zu Null setzt. Hierbei wird zwischen einer geraden und ungeraden Anzahl an Stützwerten unterschieden, um eine eindeutige Unterscheidung zwischen den positiven und negativen Frequenzen zu ermöglichen. Die Maske wird in den Zeilen 10 bis 18 analog zu Gleichung 2.41 erzeugt und anschließend mit der FFT des Signals multipliziert, um den negativen Frequenzbereich zu unterdrücken. Abschließend wird die inverse FFT des so entstehenden Signals gebildet, um das analytische Signal zu erhalten. Eine grafische Darstellung der diskreten Hilbert-Transformation ist in Abbildung 2.8 gezeigt. Hierbei wurde aus dem Realteil eines analytischen Signals mit der Hilbert-Transformation der Imaginärteil bestimmt. Ein Vergleich des Ursprungssignals mit dem aus der Hilbert-Transformation folgenden analytischen Signal ist in Abbildung 2.9 gezeigt. Es ist zu erkennen, dass es nur im Randbereich zu einem geringen Fehler kommt, der zu vernachlässigen ist.

```
1 def hilbert(signal, plot=False):
2     N = signal.size(0) # Length of the input signal
3     signal_fft = trafo.fft(signal) # FFT of input signal
4
5     # Create the frequency mask for the hilbert transform
6     H = torch.zeros(N, dtype=torch.complex64, device=signal.device)
7
8     # if N is even
9     if N % 2 == 0:
10        N_half = N // 2 # Half of the signal (when N is even)
11        H[0] = 1 # DC component
12        H[N_half+1:] = 2 # Positive frequencies
13        H[N_half] = 1 # Nyquist frequency (only for even N)
14    else:
15        N_half = (N+1) // 2 # Half of the signal (when N is uneven)
16        H[0] = 1 # DC component
17        H[:N_half] = 2 # Positive frequencies
18    # apply the frequency mask
19    signal_fft_hilbert = signal_fft * H
20
21    # inverse FFT to get the analytical signal
22    analytical_signal = trafo.ifft(signal_fft_hilbert)
23
24    return analytical_signal
```

Quelle: Eigene Darstellung, Auszug aus `./modules/loss.py`

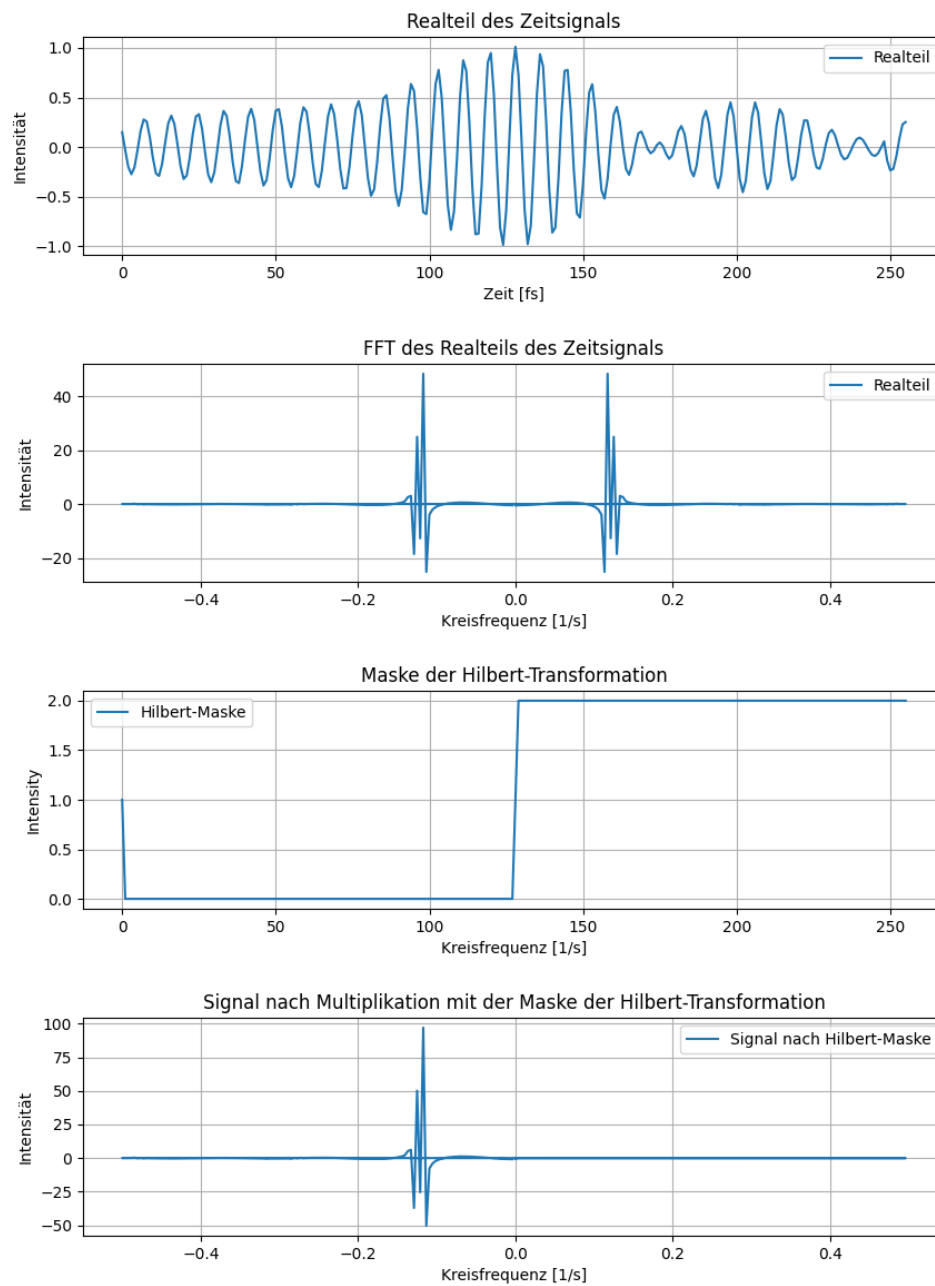


Abbildung 2.8: Darstellung der Schritte der diskreten Hilbert-Transformation
 Quelle: Eigene Darstellung

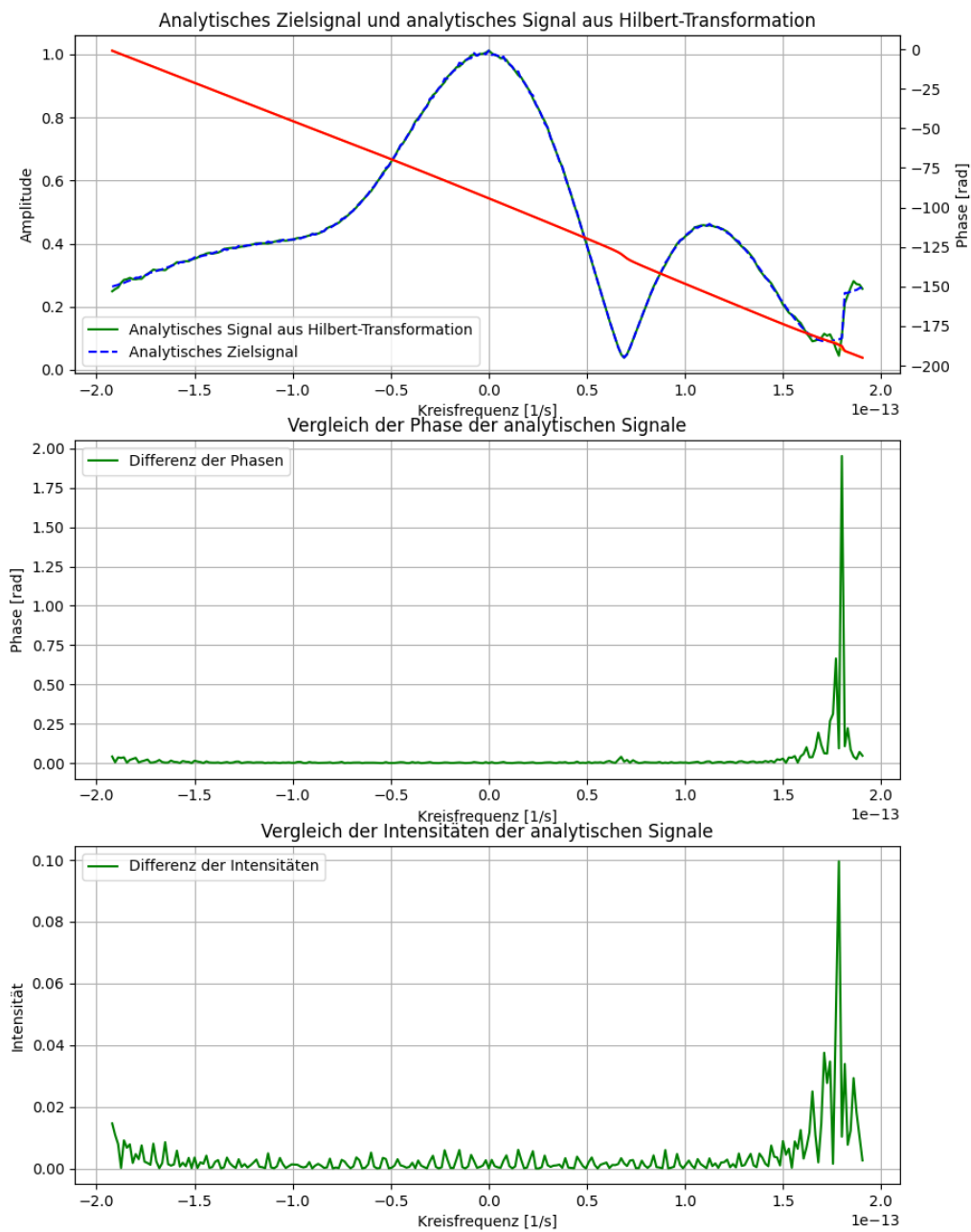


Abbildung 2.9: Vergleich eines Zielsignals mit einem durch die Hilbert-Transformation erzeugten analytischen Signal

Quelle: Eigene Darstellung

Die als Zielsignal verwendeten Zeitprofile aus dem Trainingsdatensatz liegen als analytische Signale vor. Es wird die Hilbert-Transformation genutzt, um aus dem Realteil der vorhergesagten Signals das gesamte analytische Signal zu berechnen. Dies hat eine Reduktion der Anzahl an vorherzusagenden Samples um die Hälfte zur Folge. Daraus ergibt sich ein weiterer Vorteil. Ohne Hilbert-Transformation muss das neuronale Netz zunächst den mathematischen Zusammenhang zwischen Real- und Imaginärteil lernen, was den Lernprozess erschwert. Es kann außerdem nicht garantiert werden, dass das Netz immer korrekt den korrespondierenden Imaginärteil konstruiert. Die Hilbert-Transformation hingegen erzeugt immer ein mathematisch korrektes analytisches Signal aus dem Realteil.

3 Machine Learning

3.1 DenseNet

3.1.1 Grundlagen

Da das Trainieren eines neuronalen Netzes mit vielen Schichten einen hohen Bedarf an Trainingsdaten und -zeit erfordert, soll ein bereits vortrainiertes neuronales Netz an die vorliegende Aufgabenstellung angepasst werden. SHG-Matrizen können als Bilder mit einem Farbkanal aufgefasst werden. Für Anwendungen, wie z. B. die Bildklassifizierung, sind hierbei sogenannte Faltungsnetze die vorherrschende Herangehensweise. [7, S. 1]

Faltungsnetze sind eine spezielle Kategorie neuronaler Netze, die in mindestens einer ihrer Schichten die Faltung von Matrizen anstelle ihrer Multiplikation verwenden. Sie weisen eine Spezialisierung auf Daten mit einer gitterartigen Topologie auf, wie z. B. die in der vorliegenden Arbeit behandelten Spektrogramme. Die Faltungsoperation wird in Gleichung 3.1 gezeigt. Hierbei wird x als Eingang und w als Kernel bezeichnet. Da es sich im Kontext des maschinellen Lernens um diskretisierte Probleme handelt, wird in Faltungsnetzen die in Gleichung 3.2 dargestellte diskrete Faltung genutzt. Diese lässt sich einfacher berechnen als die kontinuierliche Faltung. Es wird angenommen, dass alle Elemente außerhalb des Bildes gleich Null sind, sodass die Summe über einen endlichen Bereich gebildet werden kann. Für den zweidimensionalen Fall wird dies in Gleichung 3.3 gezeigt, wobei I das Eingangsbild und K den Kernel bezeichnet. [4, S. 326 ff.]

$$s(t) = (x * w)(t) = \int x(a)w(t - a)da \quad (3.1)$$

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^x (a)w(t - a) \quad (3.2)$$

$$s(i, j) = (K * I)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n) \quad (3.3)$$

Faltungsnetze weisen im Vergleich zu konventionellen neuronalen Netzen gewisse Vorteile auf. Konventionelle neuronale Netze basieren auf der Matrixmultiplikation, was bedeutet,

dass für jede Beziehung zwischen einem Eingangs- und Ausgangselement ein eigener Parameter erforderlich ist. Im Gegensatz dazu nutzen Faltungsnetze einen Kernel, der kleiner als die Eingangskarte ist, und verwenden nur die Elemente dieses Kernels als Parameter. Dieser Ansatz wird als *sparse interactions* bezeichnet.

Ein weiterer Vorteil von Faltungsnetzen ist das sogenannte *Parameter sharing*. Es beschreibt, dass die Elemente des Kernels auf jedem Element des Eingangs angewandt werden. Bei konventionellen neuronalen Netzen wird hingegen jeder Parameter exakt einmal verwendet. Ferner besitzt die Faltung die Eigenschaft der *Äquivarianz*. Sie beschreibt, dass eine Translation des Eingangs gleichermaßen auf dem Ausgang durchgeführt wird. Daher sind Faltungsnetze gut geeignet, um Strukturen unabhängig von ihrer Position in einem Bild zu erkennen. [4, S. 329-335]

Die Schicht eines Faltungsnetzes besteht aus drei Schritten. Zunächst wird die Faltung auf dem Eingang durchgeführt. Das Ergebnis der Faltung wird anschließend in die Detektor-Stufe geleitet, in der eine Aktivierungsfunktion angewandt wird, um signifikante Merkmale der Merkmalskarte hervorzuheben. Abschließend wird eine Poolingfunktion angewendet, was die Dimensionen der Merkmalskarte reduziert. Eine typische Poolingfunktion ist das sogenannte *max pooling*, bei dem der jeweils höchste Wert in einer rechteckigen Nachbarschaft ermittelt wird. Die Größe der Merkmalskarte ändert sich so je nach Größe der Nachbarschaft. [4, S. 335-339]

Für die vorliegende Aufgabenstellung soll ein sogenanntes DenseNet verwendet werden, welches eine Sonderform von Faltungsnetzen darstellt. Dieses wurde ausgewählt, da DenseNets in der Vergangenheit bei der Pulsrekonstruktion bereits gute Ergebnisse erzielt haben. [21, S. 2]

3.1.2 Struktur des DenseNets

Das DenseNet ist eine Netzarchitektur, die aus Faltungsschichten, sogenannten Dense Blocks und Poolingschichten besteht. Der Eingang des Netzes ist ein Bild x_0 . Das Netzwerk umfasst L Schichten, welche jeweils eine nicht-lineare Transformation $H_l(\cdot)$ durchführen. x_l bezeichnet hierbei den Ausgang jeder Schicht l , auch Merkmalskarte genannt. [7, S. 3]

In konventionellen Faltungsnetzen wird der Ausgang einer Schicht l als Eingang der Folgeschicht $l + 1$ verwendet. Dieser Übergang ist in der Gleichung 3.4 dargestellt. Die Netzarchitektur ResNet, welche in der Bildklassifizierung eine hohe Popularität besitzt, addiert zusätzlich die Einheitsfunktion zum Ausgang der nicht-linearen Transformation zu x_l . Dies ist in Gleichung 3.5 gezeigt. Diese Vorgehensweise resultiert einerseits in

einer stärkeren Interkonnektivität der Schichten, da der Gradient über die Einheitsfunktion direkt von späteren Schichten in frühere weitergegeben wird. Andererseits führt die Addition jedoch zu einer Dämpfung des Informationsflusses. [7, S. 3]

$$x_l = H_l(x_{l-1}) \tag{3.4}$$

$$x_l = H_l(x_{l-1}) + x_{l-1} \tag{3.5}$$

Um den Informationsfluss durch die Schichten zu optimieren, werden im DenseNet sämtliche vorhergehenden Merkmalskarten in jede neue Schicht weitergegeben. Zu diesem Zweck werden die Merkmalskarten aller Schichten $0, \dots, l - 1$ in einem Tensor X_{l-1} miteinander verknüpft. Dieser Vorgang ist in Gleichung 3.6 dargestellt. [7, S. 3]

$$x_l = H_l(X_{l-1}) \quad \text{mit } X_{l-1} = [x_0, x_1, \dots, x_{l-2}, x_{l-1}] \tag{3.6}$$

Darüber hinaus wird die Funktion H_l so definiert, dass zunächst eine Batch-Normalisierung, anschließend eine ReLU-Aktivierungsfunktion und letztlich eine Faltung mit einem 3×3 -Kernel durchgeführt wird. [7, S. 3]

Die Verkettung der Merkmalskarten ist nur unter der Voraussetzung möglich, dass ihre Größe zwischen den Faltungen konstant bleibt. Diese werden jedoch, wie zuvor gezeigt, in Faltungsnetzen durch die Poolingfunktionen beim sogenannten Downsampling reduziert. Um sowohl die beschriebene Struktur als auch Downsampling verwenden zu können, wird das DenseNet in mehrere sogenannte DenseBlocks unterteilt. Die Operation H_l erzeuge eine Merkmalskarte, die k Merkmale enthält. Die Schicht l weist demnach $k_0 + k \times (l - 1)$ Eingänge auf. k_0 bezeichnet hierbei die Anzahl der Kanäle von x_0 . Mit k wird die Wachstumsrate des Netzwerks bezeichnet. Ein einzelner DenseBlock ist in Abbildung 3.1 dargestellt.

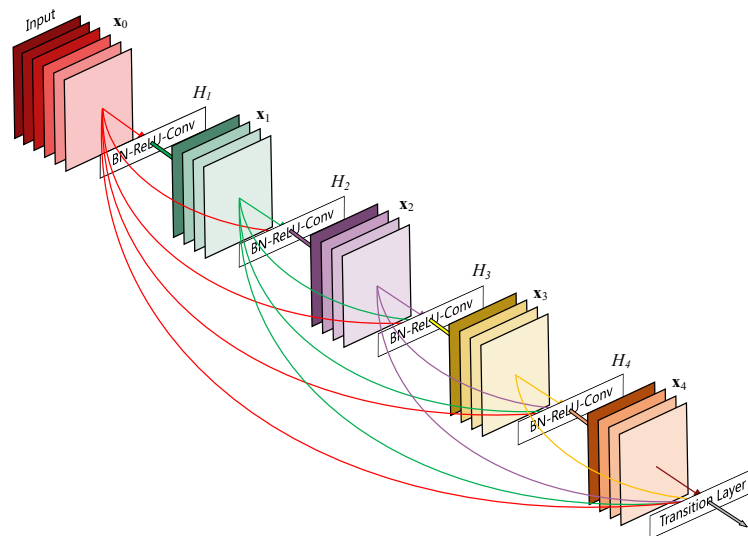


Abbildung 3.1: Ein $L = 5$ Schichten großer DenseBlock mit $k_0 = 5$ Eingangskanälen und einer Wachstumsrate von $k = 4$

Quelle: [7, S. 1]

Die Realisierung von Downsampling erfolgt durch eine Faltungsschicht mit einem 1×1 -Kernel sowie einer 2×2 -Poolingschicht zwischen den DenseBlocks. Diese werden auch als Übergangsschicht bezeichnet. Diese Struktur eines DenseNet wird in Abbildung 3.2 gezeigt. [7, S. 3]

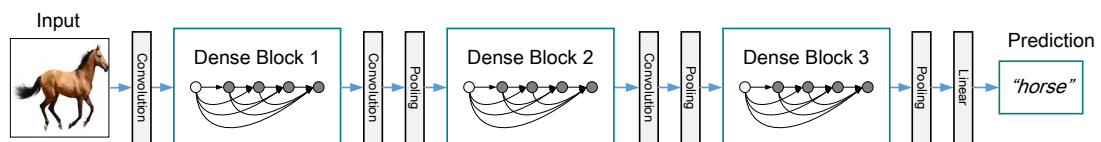


Abbildung 3.2: Struktur eines DenseNet bestehend aus 3 DenseBlocks mit dazwischenliegenden Übergangsschichten, welche die Größe der Merkmalskarten ändert

Quelle: [7, S. 3]

3.1.3 Anpassungen

Die zuvor beschriebene Architektur muss für die vorliegende Untersuchung angepasst werden. Die DenseNet Architektur ist in erster Linie für die Klassifizierung von Bildern konzipiert [7, S. 1]. Im vorliegenden Fall soll der Output des neuronalen Netzes eine Merkmalskarte sein. Sie enthält eine Anzahl an Fließkommawerten, die der Anzahl der

vorhergesagten Abtastwerte des Laserpulses entspricht. Die angepasste DenseNet-Klasse ist in Listing 9 gezeigt.

Zunächst wird ein DenseNet121 mit vortrainierten Gewichten geladen, wie in Zeile 14 des Listings ersichtlich. In Zeile 17 wird die Größe der Merkmalskarte vor der letzten Schicht des DenseNet ermittelt. An Stelle der letzten Schicht werden nun drei voll verbundene Schichten eingefügt, wie in den Zeilen 19 bis 23 zu sehen. Die Größe der Eingangs- und Ausgangsmerkmalskarten kann Tabelle 3.1 entnommen werden.

Tabelle 3.1: Größe der Eingangs- und Ausgangsmerkmalskarten des angepassten DenseNet sowie Position der Aktivierungsfunktionen

Quelle: Eigene Darstellung

	Eingangsmerkmalskarte	Ausgangsmerkmalskarte
Schicht 1	<i>num_features</i> = 1024	2048
ReLU-Aktivierungsfunktion		
Schicht 2	2048	1024
ReLU-Aktivierungsfunktion		
Schicht 3	1024	<i>num_outputs</i> = 256
tanh-Aktivierungsfunktion		

Zwischen diesen Schichten werden sowohl die ReLU- als auch die tanh-Aktivierungsfunktionen genutzt. Ihre Platzierung lässt sich ebenfalls aus Tabelle 3.1 entnehmen. Die ReLU-Aktivierungsfunktion ähnelt einer linearen Funktion, jedoch wird letztere für alle negativen Werte auf Null gesetzt. Ihre Definition ist in Gleichung 3.7 dargestellt, der zugehörige Funktionsverlauf in Abbildung 3.3. Die in PyTorch implementierte Variante der ReLU-Aktivierungsfunktion weist, anders als die mathematische Definition, auch bei negativen Werten einen Gradienten auf. Sie wird zwischen die neu eingefügten voll verbundenen Schichten platziert, da ReLU-Aktivierungsfunktionen im Vergleich zu anderen Aktivierungsfunktionen leicht zu optimieren sind. Dies ist in Listing 9 in den Zeilen 41 und 43 zu erkennen. [4, S. 189]

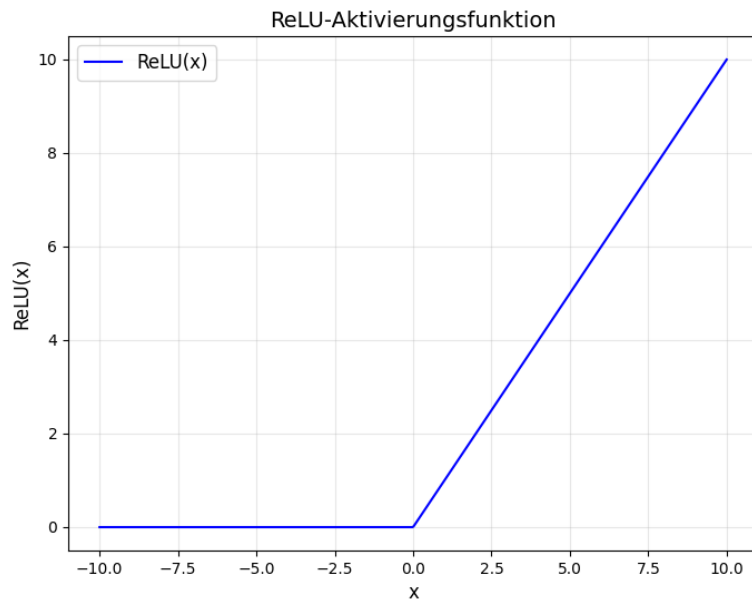


Abbildung 3.3: Funktionsverlauf einer ReLU-Aktivierungsfunktion
Quelle: Eigene Darstellung

Die tanh-Aktivierungsfunktion wird durch die tanh-Funktion definiert, wie in Gleichung 3.8 und Abbildung 3.4 gezeigt. Dabei wird der Ausgang der vorangehenden Schicht auf einen Wertebereich zwischen Eins und Null skaliert. Diese Skalierung entspricht dem Wertebereich des vorauszusagenden Pulses, weshalb hinter der letzten voll verbundenen Schicht eine tanh-Aktivierungsfunktion platziert wird. Dies ist in Zeile 47 des Listings abgebildet. [4, S. 191]

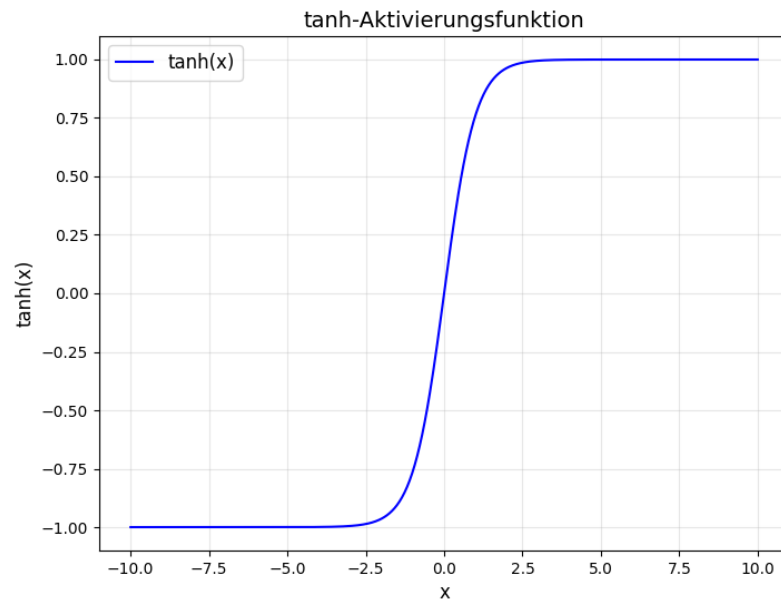


Abbildung 3.4: Funktionsverlauf einer tanh-Aktivierungsfunktion

Quelle: Eigene Darstellung

$$g_{ReLU}(x) = \text{ReLU}(x) = \begin{cases} 0 & \text{für } z < 0 \\ z & \text{für } z \geq 0 \end{cases} \quad (3.7)$$

$$g_{\tanh}(x) = \tanh(x) \quad (3.8)$$

Die Initialisierung der Gewichte der hinzugefügten Schichten erfolgt unter Berücksichtigung der Aktivierungsfunktion hinter der jeweiligen Schicht. Für Schichten, die sich vor einer tanh-Aktivierungsfunktion befinden, hat sich die sogenannte Xavier-Initialisierung etabliert [1]. Bei einer n -Merkmale großen Merkmalskarte ergeben sich die Gewichte $W_{i,j}$ aus U , einer uniformen Wahrscheinlichkeitsverteilung. Ihr Wertebereich liegt zwischen den Grenzen $-\left(\frac{1}{\sqrt{n}}\right)$ und $\left(\frac{1}{\sqrt{n}}\right)$. Dies ist in Gleichung 3.9 und in Zeile 28 von Listing 9 erkennbar. [3, S.251].

Für Schichten, die sich vor ReLU-Aktivierungsfunktionen befinden, hat sich hingegen die Kaiming-Initialisierung etabliert [1]. Hier werden die Gewichte mit einer Gauss-Verteilung G initialisiert, wobei der Mittelwert \bar{x} gleich Null und die Standardabwei-

chung $\sigma = \left(\sqrt{\frac{2}{n}}\right)$ ist, wie in Gleichung 3.10 dargestellt [6, S. 4]. Dies ist gezeigt in Zeile 26 von Listing 9.

$$W_{i,j} = U \left[-\left(\frac{1}{\sqrt{n}}\right) \quad , \quad \left(\frac{1}{\sqrt{n}}\right) \right] \quad (3.9)$$

$$W_{i,j} = G \left[\bar{x} = 0 \quad , \quad \sigma = \left(\sqrt{\frac{2}{n}}\right) \right] \quad (3.10)$$

```
1 """
2 CustomDenseNetReconstruction()
3 Description:
4     Custom DenseNet class for reconstructiong the time domain pulse from SHG-matrix
5 """
6 class CustomDenseNetReconstruction(nn.Module):
7     def __init__(self, num_outputs=512):
8         """
9         Inputs:
10            num_outputs -> [int] Number of outputs of the DenseNet
11            """
12            super(CustomDenseNetReconstruction, self).__init__()
13            # Load pretrained DenseNet
14            self.densenet =
15            ↪ models.densenet121(weights=models.DenseNet121_Weights.DEFAULT)
16            # self.densenet =
17            ↪ models.densenet161(weights=models.DenseNet161_Weights.DEFAULT)
18            # Get the number of features before the last layer
19            num_features = self.densenet.classifier.in_features
20            # Create a Layer with the number of features before the last layer and 256
21            ↪ outputs (2 arrays of 128 Elements)
22            self.densenet.classifier = nn.Linear(num_features, 2048)
23            self.fc1 = nn.Linear(2048, 1024)
24            self.fc2 = nn.Linear(1024, num_outputs)
25            # self.densenet.classifier = nn.Linear(num_features, num_outputs)
26            self.num_outputs = num_outputs
```

```
25     # initialize weights
26     nn.init.kaiming_normal_(self.fc1.weight, mode='fan_out', nonlinearity='relu')
    ↪ # useful before relu
27     nn.init.zeros_(self.fc1.bias) # Initialize biases to zero
28     nn.init.xavier_normal_(self.fc2.weight) # useful before tanh
29     nn.init.zeros_(self.fc2.bias) # Initialize biases to zero
30
31     def forward(self, shg_matrix):
32         '''
33         Forward pass through the DenseNet
34         Input:
35             shg_matrix    -> [tensor] SHG-matrix
36         Output:
37             x             -> [tensor] predicted output
38         '''
39         # get the output of the densenet
40         x = self.densenet(shg_matrix)
41         x = torch.relu(x)
42         x = self.fc1(x)
43         x = torch.relu(x)
44         x = self.fc2(x)
45
46         # use tanh activation function to scale the output to [-1, 1] and then scale it
    ↪ (intensity)
47         x = torch.tanh(x)
48         return x
```

Listing 9: Für die Vorhersage von Zeitprofilen aus Spektrogrammen modifizierte DenseNet-Klasse

Quelle: Eigene Darstellung, Auszug aus `./modules/models.py`

3.2 Genutzter Datensatz

3.2.1 Struktur des Datensatzes

Für das Trainieren des Datensatzes wird ein Datensatz mit künstlich erstellten Spektrogrammen und Zielsignalen genutzt, wobei Pulse im Frequenzbereich simuliert und mittels DTFT in den Zeitbereich transformiert werden. Aus dem nun vorliegenden analytischen Signal werden gemäß Kapitel 2.3 Spektrogramme erzeugt, die so transformiert werden, dass sich eine Wellenlängenachse ergibt.

Der genutzte Datensatz umfasst 100 000 Datenpunkte, die in nummerierten Verzeichnissen organisiert sind. Zwei für das Trainieren des Netzes relevante Dateien sind enthalten: *'Es.dat'* enthält fünf Spalten, wobei die erste die äquidistante Verzögerungsachse darstellt. Diese ist zwischen allen Datenpunkten identisch. Die darauffolgenden zwei Spalten repräsentieren die Intensitäts- und Phasenwerte des Signals. Im Rahmen des Trainings werden ausschließlich die letzten beiden Spalten genutzt, da diese den Real- und Imaginärteil des Signals enthalten. Die zweite relevante Datei *'as_gn00.dat'* enthält einen Header in der ersten ein oder den ersten beiden Zeilen mit insgesamt fünf Elementen:

- Anzahl an Verzögerungswerten im Spektrogramm
- Anzahl an Wellenlängenwerten im Spektrogramm
- Abstand zwischen den Verzögerungswerten $\Delta\tau$ in fs
- Abstand zwischen den Wellenlängenwerten $\Delta\lambda$ in nm
- Wellenlängenwert in der Mitte der Wellenlängenachse λ_c in nm

In den nachfolgenden Zeilen und Spalten werden die Intensitätswerte des Spektrogramms angegeben.

3.2.2 Vorverarbeitung

Vorverarbeitung der Zielsignale

Um die Resultate des Trainings zu verbessern, müssen auch die Zielsignale aufbereitet werden. Wie in Kapitel 2.5 dargelegt, können verschiedene Zeitsignale zum gleichen FROG-Trace führen. Um ein eindeutiges Verhältnis zwischen Spektrogramm und Zeitsignal zu etablieren, müssen vor dem Beginn des Trainings triviale Ambiguitäten entfernt werden. Dies erfolgt mit den in Listing 5 bis 7 dargestellten Funktionen. Im Anschluss

werden die Signale auf eine Amplitude von Eins normalisiert, um eine möglichst eindeutige und einheitliche Struktur der Zielsignale zu gewährleisten. Da lediglich bei künstlich erstellten Datenpunkten ein Zeitsignal bekannt ist, kann die Auflösung der Verzögerungsachse einheitlich gewählt werden, wodurch eine erneute Abtastung entfällt.

Vorverarbeitung der Spektrogramme

Da die vorliegenden Spektrogramme einen unterschiedlichen Wellenlängenbereich abdecken, ist es erforderlich, diese vor Beginn des Trainings auf einen einheitlichen Bereich zu resampeln. Zu diesem Zweck wird zunächst das Skript `'datasetInformation.py'` aufgerufen, welches u. a. die geringsten und höchsten Wellenlängenwerte aus dem Trainingsdatensatz bestimmt. Darüber hinaus wird der höchste Wert für den Abstand zwischen den Verzögerungswerten $\Delta\tau$ ausgegeben.

Über die Datei `./modules/config.py` lässt sich definieren, auf welche Achsenbereiche die Spektrogramme neu abgetastet werden. Hierzu werden zunächst die Spektrogramme und ihre zugehörigen Header eingelesen und anschließend mithilfe linearer Interpolation in Verzögerungs- und Wellenlängenrichtung auf eine einheitliche Größe mit gleichen Achsen gebracht. Die hierfür verwendete Funktion ist in Listing 10 gezeigt. Der Funktion werden eine Achse x , ein Signal y und eine neue Achse x_{new} übergeben. In Zeile 10 werden mit der Funktion `torch.searchsorted` die Indizes der alten x -Achse ermittelt, zwischen denen die neuen x -Achsenwerte liegen. Diese werden in Zeile 12 so beschnitten, dass keine Werte außerhalb des gültigen Bereichs liegen. In den Zeilen 15 und 16 werden jeweils die oberen und unteren Interpolationspunkte extrahiert. Die zu den Indizes gehörigen x -Achsen- und Signalwerte werden in den Zeilen 19 bis 22 bestimmt. In den Zeilen 25 und 26 werden schließlich die Steigungen m zwischen den oberen und unteren Interpolationspunkten gebildet. Hierzu wird die Gleichung 3.11 genutzt. Abschließend wird in Zeile 30 die in Gleichung 3.12 dargestellte Gradengleichung zur Bestimmung der neuen y -Werte genutzt.

$$m = \frac{y_{upper} - y_{lower}}{x_{upper} - x_{lower}} \quad (3.11)$$

$$y_{new} = y_{lower} + m \cdot (x_{new} - x_{lower}) \quad (3.12)$$

```
1 def piecewiseLinearInterpolation(x, y, x_new):
2     # make sure all tensors are on the same device
3     device = y.device
4     x = x.to(device)
5     x_new = x_new.to(device)
6     y_new = torch.zeros_like(x_new).to(device)
7
8     # find the indices in x where the values of x_new
9     # would fit to maintain sorting
10    indices = torch.searchsorted(x, x_new).to(device)
11    # clamp all indices outside of bounds
12    indices = torch.clamp(indices, 1, len(x) - 1)
13
14    # get the indices of upper and lower bounding points of each x_new for
15    ↪ interpolation
16    lower_indices = indices - 1
17    upper_indices = indices
18
19    # get corresponding x and y values for upper and lower bounding points
20    x_lower = x[lower_indices]
21    x_upper = x[upper_indices]
22    y_lower = y[lower_indices]
23    y_upper = y[upper_indices]
24
25    # calculate the slope (m) for the linear segment between each bounding point
26    delta_y = y_upper - y_lower
27    delta_x = x_upper - x_lower
28    m = delta_y / delta_x
29
30    # perform linear interpolation:  $y = y_1 + m \cdot (x_{\text{new}} - x_{\text{lower}})$ 
31    y_new = y_lower + m * (x_new - x_lower)
32
33    return y_new
```

Quelle: Eigene Darstellung, Auszug aus `./modules/helper.py`

Nach dem erneuten Abtasten der Spektrogramme erfolgt ihre Transformation in das für das DenseNet korrekte Format. Zu diesem Zweck werden mit der Klasse *Create3ChannelSHGmatrix* zwei zusätzliche Kanäle zu den Spektrogrammen hinzugefügt, die identisch zu dem ersten Kanal sind. Im Anschluss erfolgt die Normalisierung der Kanäle. Die zur Normalisierung genutzten Werte sind in Tabelle 3.2 angegeben. [15]

Tabelle 3.2: Standardabweichung und Mittelwert für die Normalisierung von jedem Kanal der Spektrogramme für das verwendete DenseNet

	Standardabweichung	Mittelwert
Kanal 1	0,229	0,485
Kanal 2	0,224	0,456
Kanal 3	0,225	0,406

3.2.3 Einheitliche Vorverarbeitung für simulierte und experimentelle Daten

Zusätzlich zu den in dieser Arbeit verwendeten künstlich produzierten Daten soll das neuronale Netz in Zukunft zur Vorhersage von Zeitprofilen aus experimentell aufgenommenen Daten genutzt werden. Experimentelle Daten unterscheiden sich jedoch von den künstlich erstellten Daten, weshalb eine zusätzliche Vorverarbeitung erforderlich ist. So kann es z. B. durch nicht richtig ausgerichtete Laser oder Optiken im Versuchsaufbau zu einem nicht symmetrischen Spektrogramm mit Verzerrungen kommen. Des Weiteren ist ein signifikantes Rauschen in experimentellen Daten festzustellen, welches beim Extrapolieren der Spektrogramme zu harten Rändern im Zielspektrogramm führt. Ein exemplarisches experimentell ermitteltes Spektrogramm ist in Abbildung 3.5 zu sehen.

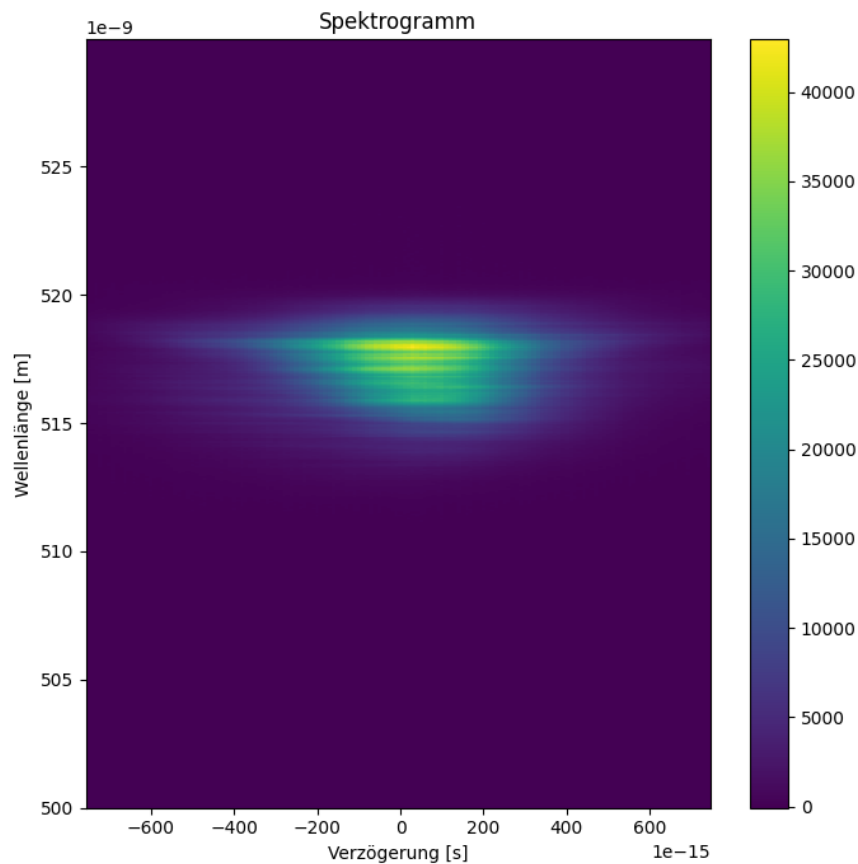


Abbildung 3.5: Experimentell ermitteltes Spektrogramm
Quelle: Eigene Darstellung

Die Nutzung des neuronalen Netzes für beide Typen der Trainingsdaten erfordert einen hohen Grad an einheitlicher Vorverarbeitung. Ein zusätzlicher Vorteil der einheitlichen Vorverarbeitung vor dem Start des Trainierens liegt in der signifikanten Verkürzung der Trainingszeit, die hierdurch erzielt wird.

Zunächst wird das Spektrogramm so beschnitten, dass der Schwerpunkt des Pulses zentriert auf der Verzögerungsachse liegt. Um den Index des Schwerpunkts zu ermitteln, wird über alle Zeilen des Spektrogramms iteriert. Anschließend werden alle Werte der aktuellen Reihe aufsummiert und diese Summe mit dem aktuellen Index multipliziert. Dieses Produkt wird nun über alle Reihen aufaddiert und durch die Summe aller Werte im Spektrogramm dividiert, um den Index mit dem höchsten Gewicht zu bestimmen. Dieser Vorgang ist in Listing 11 dargestellt.

```
1 # get the sum of all spectrogram values
2 total_int = float(torch.sum(shg_matrix))
3
4 com_delay_index = 0    # index of center of mass in delay direction
5 # enumerate over rows
6 for index, row in enumerate(shg_matrix):
7     # get sum of row
8     sum_row = torch.sum(row)
9     # add previous center of mass index and the current index
10    # then multiply by the sum of the values in the current row
11    com_delay_index += float((index + 1) * sum_row )
12
13 # devide the com_delay_index by the sum of total SHG-matrix values and round it
14 com_delay_index = round(com_delay_index / total_int)
```

Quelle: Eigene Darstellung, Auszug aus `./modules/preprocessing.py`

Das vorliegende Spektrogramm wird nun mithilfe des Index beschnitten, wie in Listing 12 zu sehen. In Zeile 5 wird ermittelt, ob der Index des Schwerpunktes der Verzögerung oder seine Differenz von der Gesamtmenge an Verzögerungswerten kleiner ist. Der kleinere Wert wird anschließend genutzt, um vom Schwerpunktsindex ausgehend eine nach oben und unten gleich große Menge an Indizes zu bestimmen. In Zeile 14 wird das Spektrogramm mithilfe dieses Wertes in Verzögerungsrichtung so beschnitten, dass der Schwerpunkt des Pulses sich im Zentrum befindet. Des Weiteren wird eine Verzögerungsachse erstellt, wie in Zeile 23 gezeigt.

```
1 # get the minimum of either:
2 # - the number of delays - the center of mass index
3 # - the center of mass index
4 # This will get the index which has the smallest distance to the end of the matrix
5 distance_to_end = int(min(num_delays - com_delay_index - 1, com_delay_index))
6 # Create a tensor with the index range to be in the new SHG-matrix
7 # The smallest value will be the Center of Mass index - the smallest distance to the
   ↪ end of the original matrix
8 # The highest value will be the Center of Mass index + the smallest distance to the
   ↪ end of the original matrix
9 # This means com_delay_index is exactly in the middle of the index range
10 index_range = torch.arange(com_delay_index - distance_to_end, com_delay_index
   ↪ + distance_to_end + 1)
11
12 # Cut a matrix that is equally as large in both directions of com_delay_index
13 # This means com_delay_index is exactly in the middle
14 symmetric_shg_matrix = shg_matrix[index_range, :]
15 num_symmetric_delays = symmetric_shg_matrix.shape[0] # size of matrix in
   ↪ delay direction (uneven by design)
16
17 # Construct Delay Axis for symmetric_spectrogram_matrix
18 middle_index = (num_symmetric_delays + 1) // 2 # since num_symmetric_delays
   ↪ is uneven (see above)
19 # create the delay axis
20 # get a range of values as long as the number of delays
21 # shift it with around the middle index
22 # scale it with delta_tau
23 symmetric_delay_axis = (torch.arange(num_symmetric_delays) - (middle_index -
   ↪ 1)) * delta_tau
```

Quelle: Eigene Darstellung, Auszug aus `./modules/preprocessing.py`

SHG-Matrizen sind in Verzögerungsrichtung symmetrisch [20, S. 111]. Allerdings ist dies bei experimentell ermittelten Spektrogrammen nicht immer der Fall, da z.B. Fehler bei der Ausrichtung von Optiken auftreten können. Daher ist es erforderlich, diese zu symmetrisieren, wie Listing 13 zeigt. In den Zeilen 2 und 3 werden zunächst die Teile

des Spektrogramms, welche links bzw. rechts vom Mittenindex liegen, in eigene Matrizen kopiert. In Zeile 5 wird nun das arithmetische Mittel zwischen den Werten des linken Teils und dem gespiegelten rechten Teil gebildet. Anschließend wird die symmetrische SHG-Matrix aus dem arithmetischen Mittel sowie seiner Spiegelung gebildet. Dieser Schritt ist in den Zeilen 7 und 8 abgebildet.

```
1 # 2: Symmetrization of matrix around the center of mass by getting the mean left and
   ↪ right halves
2 left_matrix = symmetric_shg_matrix[:middle_index-1, :]
3 right_matrix = symmetric_shg_matrix[middle_index:, :]
4 # get the mean of left and mirrored right half of the matrix
5 left_symmetric = 0.5 * (left_matrix + torch.flip(right_matrix, dims=[0]))
6 # replace left and right half by their mean and the flipped mean
7 symmetric_shg_matrix[:middle_index-1, :] = left_symmetric
8 symmetric_shg_matrix[middle_index:, :] = torch.flip(left_symmetric, dims=[0])
```

Quelle: Eigene Darstellung, Auszug aus `./modules/preprocessing.py`

Die Auswahl der Verzögerungs- und Wellenlängenschritte sollte unter der Prämisse erfolgen, dass die Abtastrate in beide Richtungen in etwa gleich ist. Dies wird ermöglicht, indem das Verhältnis zwischen Verzögerungsschritt und FWHM-Wert in Verzögerungsrichtung $\Delta\tau_{FWHM}$ gleich dem Verhältnis aus Wellenlängenschritt und FWHM in Wellenlängensrichtung $\Delta\lambda_{FWHM}$ ist. Die korrekten Werte für $\Delta\tau$ und $\Delta\lambda$ können mithilfe eines Quotienten M bestimmt werden, dessen Bestimmung in den Gleichungen 3.13, 3.14 und 3.15 dargestellt ist. In Listing 14 wird gezeigt, wie das Spektrogramm neu abgetastet wird. Zunächst wird, wie in Kapitel 2.1.3, der FWHM-Wert in beide Richtungen des Spektrogramms bestimmt, wie den Zeilen 5 und 10 entnommen werden kann. Aus diesen wird in Zeile 13 der Quotient M bestimmt. Nun werden in den Zeilen 14 und 15 die optimalen Werte für $\Delta\tau$ und $\Delta\lambda$ berechnet. Anschließend werden die Achsen des Spektrogramms in den Zeilen 30 und 41 so neu abgetastet, dass die optimalen Werte genutzt werden. Das Spektrogramm selbst wird in Zeile 53 neu abgetastet.

$$\Delta\tau = \frac{\Delta\tau_{FWHM}}{M} \quad (3.13)$$

$$\Delta\lambda = \frac{\Delta\lambda_{FWHM}}{M} \quad (3.14)$$

$$M = \sqrt{\Delta\tau_{FWHM}\Delta\lambda_{FWHM}N} = \sqrt{\frac{\Delta\tau_{FWHM}\Delta\lambda_{FWHM}Nc_0}{\lambda_c^2}} \quad (3.15)$$

```
1 # estimate full width half mean in both directions (not perfectly exact, but close
  ↪ enough)
2 # get the mean the delay
3 mean_delay_profile = torch.mean(symmetric_shg_matrix, dim=1)
4 # get the full width half mean of the delay
5 fwhm_delay = calcFWHM(mean_delay_profile, symmetric_delay_axis)
6
7 # get the mean the wavelengths
8 mean_shg_profile = torch.mean(symmetric_shg_matrix, dim=0)
9 # get the full width half mean of the wavelengths
10 fwhm_wavelength = calcFWHM(mean_shg_profile, wavelength_axis)
11
12 # Trebino Formula 10.8
13 M = np.sqrt(fwhm_delay * fwhm_wavelength * nTarget * c.c0 /
  ↪ center_wavelength**2)
14 opt_delay = fwhm_delay / M
15 opt_wavelength = fwhm_wavelength / M
16
17 # construction of axes for resampling
18 index_vector = torch.arange(-nTarget // 2, nTarget // 2)
19 resampled_delay_axis = index_vector * opt_delay
20 resampled_wavelength_axis = index_vector * opt_wavelength + center_wavelength
21
22 # get the index range, where the values of the resampled delay axis are within the
23 # limits of the symmetric delay axis
24 # This is done to prevent extrapolating the SHG-matrix, since this fails with noise
25 index_range_new_delay = indexRangeWithinLimits(
26     resampled_delay_axis,
```

```
27     (symmetric_delay_axis.min(), symmetric_delay_axis.max())
28     )
29     # get the subset of 'resampled_delay_axis', that is within the
    ↪ 'index_range_new_delay'
30     resampled_delay_axis_subset = resampled_delay_axis[
31         index_range_new_delay[0]:index_range_new_delay[1]
32     ]
33
34     # get the index range, where the values of the resampled wavelength axis are within
    ↪ the
35     # limits of the wavelength axis
36     index_range_new_wavelength = indexRangeWithinLimits(
37         resampled_wavelength_axis,
38         (wavelength_axis.min(), wavelength_axis.max())
39     )
40     # get the subset of 'resampled_wavelength_axis', that is within the
    ↪ 'index_range_new_wavelength'
41     resampled_wavelength_axis_subset = resampled_wavelength_axis[
42         index_range_new_wavelength[0]:index_range_new_wavelength[1]
43     ]
44
45     # 2D interpolate
46     # initialize the interpolator
47     interpolator = RectBivariateSpline(
48         symmetric_delay_axis.numpy(),
49         wavelength_axis.numpy(),
50         symmetric_shg_matrix.numpy()
51     )
52     # perform interpolation
53     shg_interpolated = interpolator(
54         resampled_delay_axis_subset.numpy(),
55         resampled_wavelength_axis_subset.numpy()
56     ).T
57     index_delay_range_min = index_range_new_delay[0]
58     index_delay_range_max = index_range_new_delay[1]
```

```
59 index_wavelength_range_min = index_range_new_wavelength[0]
60 index_wavelength_range_max = index_range_new_wavelength[1]
```

Listing 14: Erneutes Abtasten, um die optimalen $\Delta\tau$ - und $\Delta\lambda$ -Werte zu nutzen
Quelle: Eigene Darstellung, Auszug aus `./modules/preprocessing.py`

Da die Spektrogramme an diesem Punkt unterschiedliche Größen aufweisen können, sollen sie in ein neue Spektrogramme einheitlicher Größe kopiert werden. Dabei ist zu berücksichtigen, dass bis zum Rand des Spektrogramms Intensitätswerte über $1 \cdot 10^{-4}$ auftreten können, also das Kriterium der FSR nicht eingehalten wird. Dies führt zusätzlich zu harten Sprüngen zwischen dem bisherigen Spektrogramm und dem übrigen Tensor. Daher ist es erforderlich, eine Fensterfunktion über das Spektrogramm zu legen, damit dieses an den Rändern Werte nahe Null besitzt. Die dazu genutzte Funktion ist in Listing 15 dargestellt. In Zeile 5 wird die Standardabweichung des Fensters berechnet, wobei die Länge der Matrix und die ausgewählte Standardabweichung berücksichtigt wird. In Zeile 7 wird die so ermittelte Fensterfunktion in einen Tensor geladen. Je nach ausgewählter Dimension wird dieses mit dem Spektrogramm multipliziert, wie in den Zeilen 11 bis 14 zu sehen ist.

```
1 def windowSHGmatrix(shg_matrix, dimension = 1, standard_deviation_factor = 0.1):
2     # determine the window width (equals the number of samples across the selected
3     ↪ dimension)
4     window_width = int(shg_matrix.size(dimension))
5     # calculate the standard deviation used for the gaussian window
6     window_standard_deviation = window_width * standard_deviation_factor
7     # define the window
8     window_delay = torch.from_numpy( windows.gaussian(window_width,
9     ↪ std=window_standard_deviation)).float()
10
11     # apply the window to the SHG-matrix
12     # 'None' adds a new axis for broadcasting
13     if dimension == 1:
14         shg_matrix_windowed = shg_matrix * window_delay[None, :]
```

```
13     elif dimension == 0:  
14         shg_matrix_windowed = shg_matrix * window_delay[:, None]  
15  
16     return shg_matrix_windowed
```

Listing 15: Funktion, um ein Gauss-Fenster über eine Richtung des Spektrogramms zu legen

Quelle: Eigene Darstellung, Auszug aus `./modules/preprocessing.py`

Um Sprünge beim Einfügen des Spektrogramms in einen Tensor definierter Größe zu verhindern, soll der Tensor mit Rauschen gefüllt werden. Dieses entspricht der Standardabweichung der Werte am oberen Teil des linken und rechten Randes. Dieser Vorgang ist in Listing 16 dargestellt. In den Zeilen 2 bis 6 werden die oberen 10% der Ränder in einem Tensor zusammengefügt. Die Standardabweichung dieses Tensors wird mit einer Zufallsmatrix der definierten Größe multipliziert. Anschließend wird das arithmetische Mittel des Tensors aufaddiert, wie in Zeile 8 zu erkennen ist. In Zeile 14 wird das Spektrogramm in den mit Rauschen gefüllten Tensor eingefügt.

Im finalen Schritt werden sämtliche Spektrogramme einer erneuten Abtastung unterzogen, mit dem Ziel, eine Vereinheitlichung der Achsenabschnitte zu erreichen. Zu diesem Zweck werden vor der Vorverarbeitung die Minima und Maxima der Achsen aller Spektrogramme im Datensatz bestimmt. Das erneute Abtasten wird, wie in Kapitel 3.2.2 beschrieben, durchgeführt. Ein Vergleich zwischen einem originalen und einem vorverarbeiteten Spektrogramm ist in Abbildung 3.6 abgebildet.

```
1 # get the upper 5% of the edge
2 width = shg_interpolated.size(0)
3 top_percent_width = int(width * 0.1)
4 left_edge = shg_interpolated[:top_percent_width, 0]
5 right_edge = shg_interpolated[:top_percent_width, -1]
6 noise = torch.cat([left_edge, right_edge], dim=0)
7 # fill the background of the new SHG-matrix with the noise
8 resampled_shg_matrix = torch.std(noise) * torch.abs(torch.randn(nTarget, nTarget))
9     ↪ + torch.mean(noise)
10
11 # Embed the interpolated matrix into the larger matrix
12 resampled_shg_matrix = resampled_shg_matrix.numpy()
13 resampled_shg_matrix[index_wavelength_range_min:index_wavelength_range_max,
14     ↪ x, \
15     index_delay_range_min:index_delay_range_max] = shg_interpolated
16 resampled_shg_matrix = torch.from_numpy(resampled_shg_matrix.T)
```

Listing 16: Einfügen des vorverarbeiteten Spektrogramms in ein mit Rauschen gefülltes Spektrogramm

Quelle: Eigene Darstellung, Auszug aus `./modules/preprocessing.py`

Die in diesem Unterkapitel beschriebene Vorverarbeitung wird für die weiteren Teile der Untersuchung nicht genutzt. Durch das Vorverarbeiten gehen die Mittenwellenlängen λ_c sowie der ursprüngliche Abstand zwischen den Verzögerungswerten $\Delta\tau$ verloren. Diese sind jedoch für das Berechnen des FROG-Errors in der Verlustfunktion erforderlich. Daher ist es für eine Berücksichtigung in zukünftigen Untersuchungen notwendig, die in Kapitel 3.5 beschriebene Verlustfunktion entsprechend zu modifizieren. Darüber hinaus weisen die derzeit vorliegenden experimentellen Daten nicht die geforderte Qualität und Quantität auf. Daher ist es erforderlich, diese Vorverarbeitung zukünftig zu evaluieren und anzupassen.

Vergleich zwischen original und vorverarbeitetem Spektrogramm

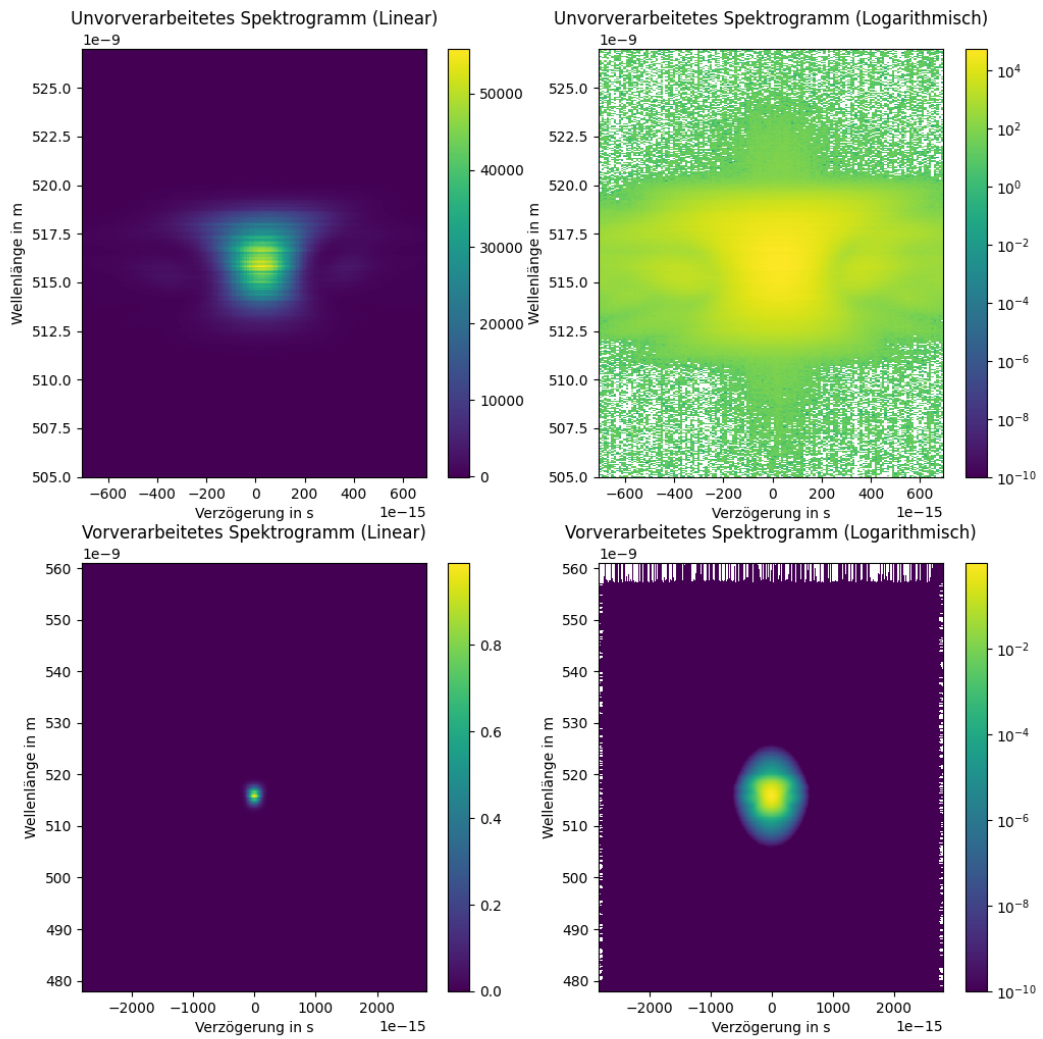


Abbildung 3.6: Vergleich eines experimentell bestimmten Spektrogramms vor und nach seiner Vorverarbeitung. Links in linearer und rechts in logarithmischer Darstellung

Quelle: Eigene Darstellung

3.3 Rekonstruktion der Zeitprofile von Kurzzeitalasern mithilfe eines DenseNets

3.3.1 Verwendete Trainingsschleifen

Überwachtes Lernen mit simulierten Daten

Es gibt zwei Möglichkeiten, das Modell zu trainieren. Zunächst wird das sogenannte überwachte Lernen mit einem simulierten Trainingsdatensatz betrachtet. Überwachtes Lernen bezeichnet hier, dass das neuronale Netz für jedes vorhergesagte Merkmal eine Zielvariable erhält, anhand derer die Vorhersage bewertet wird [4, S. 102 f.]. Im Kontext der hier betrachteten Aufgabe soll ein Zeitprofil des E-Feldes aus einer SHG-Matrix vorhergesagt werden. Dabei stellen die vorhergesagten Abtastwerte jeweils ein Merkmal dar. Jedem dieser Abtastwerte ist eine Zielvariable zugeordnet, die für die Verlustfunktion verwendet wird.

Das überwachte Lernen wird in folgenden Schritten durchgeführt:

1. Laden des Datensatzes und Transformation der darin enthaltenen SHG-Matrizen und Zeilvariablen
2. Laden des Modells
3. Aufteilung des Datensatzes in Trainings-, Validierungs- und Testdaten
4. Training des neuronalen Netzes
5. Validierung des Modells mit Validierungsdaten nach jeder Epoche
6. Testen des Modells auf dem Testdatensatz nach Abschluss des Trainings

Im Folgenden wird auf die einzelnen Schritte näher eingegangen. Zunächst werden die vom Dataloader durchgeführten Transformationen initialisiert. Die Datenpunkte werden zunächst aus einer Datei gelesen und dann in ein einheitliches Format gebracht. Dabei werden die Verzögerungen sowie minimale und maximale Wellenlänge, auf die die SHG-Matrizen neu abgetastet werden sollen, festgelegt. Außerdem wird die Anzahl der Punkte N definiert, die die Größe der resultierenden $N \cdot N$ großen SHG-Matrix bestimmt. Die hier genannten Konfigurationen werden in der Datei `./modules/config.py` definiert. Schließlich wird die SHG-Matrix so kopiert, dass drei Kanäle entstehen. Diese werden dann mit Standardabweichungen und Mittelwerten normalisiert, die auf das DenseNet ausgelegt sind [15]. Die Zielvariablen werden ebenfalls aus einer Datei gelesen und dann

anschließend um ihre trivialen Mehrdeutigkeiten, wie in Kapitel 2.5 beschrieben, bereinigt.

Anschließend wird das in Kapitel 3.1 beschriebene modifizierte DenseNet geladen. Dieses verwendet die Gewichte eines zuvor trainierten Netzes zur Bildklassifikation. Da eine Änderung aller Gewichte des Modells während des Trainings zu keiner signifikanten Verbesserung des Ergebnisses führt, werden i. d. R. alle Gewichte des eigentlichen DenseNets eingefroren. Während des Trainings werden also nur die Gewichte der Schichten verändert, die zur Modifikation des DenseNets genutzt werden. Da sich hierdurch die Anzahl der zu berechnenden Gradienten stark reduziert, wird das Training schneller [4, S. 423 ff.]. Für die folgenden Schritte gilt, es den Datensatz in Trainings-, Validierungs- und Testdaten aufzuteilen. Wenn das Training nur für eine Epoche durchgeführt wird, findet keine Validierung des Modells statt. In diesem Fall werden 90 % der Datenpunkte als Trainingsdaten und 10 % als Testdaten verwendet. Bei mehr als einer Epoche werden 80 % der Datenpunkte als Trainingsdaten und je 10 % als Validierungs- und Testdaten genutzt. Dies ist in Listing 17 abgebildet.

```
1 # get ratios of train, validation and test data
2 if config.NUM_EPOCHS > 1:
3     test_size = int(0.1 * length_dataset)           # amount of test data (10%)
4     validation_size = int (0.1 * length_dataset)    # amount of validation
5     ↪ data (10%)
6     train_size = length_dataset - test_size - validation_size # amount of training
7     ↪ and validation data (80%)
8 else:
9     test_size = int(0.1 * length_dataset)           # amount of test data (10%)
10    validation_size = int (0.0 * length_dataset)    # amount of validation
11    ↪ data (0%)
12    train_size = length_dataset - test_size - validation_size # amount of training
13    ↪ and validation data (90%)
14 logger.info(f"Size of training data: {train_size}")
15 logger.info(f"Size of validation data: {validation_size}")
16 logger.info(f"Size of test data: {test_size}")
```

```
14 # split the dataset accordingly
15 train_data, validation_data, test_data = random_split(data_loader, [train_size,
    ↪ validation_size, test_size]) # split data
```

Listing 17: Ausschnitt des Trainingskriptes zum Aufteilen des Datensatzes
Quelle: Eigene Darstellung, Auszug aus `./training.py`

Für das Training des Netzes müssen einige Funktionen initialisiert werden. Zunächst wird das Verlustkriterium definiert. Dazu wird die in Kapitel 3.5 beschriebene Verlustfunktion verwendet. Da es sich um überwachtes Lernen handelt, kann sowohl die von den Zielgrößen abhängige MSE-Komponente als auch der davon unabhängige FROG-Fehler verwendet werden. Darüber hinaus wird der Optimierer ausgewählt und initialisiert. Dieser wird in Kapitel 3.4 genauer beschrieben. Zuletzt wird ein Learning-Rate-Scheduler gewählt. Er verändert die verwendete Lernrate anhand unterschiedlicher Kriterien und wird in Kapitel 3.3.2 erklärt.

Die Trainingsschleife ist in Listing 18 abgebildet. Das eigentliche Training iteriert über die Trainingsdaten in Batches von 10 Elementen. Das bedeutet, dass in jedem Schritt 10 Spektrogramme gleichzeitig verarbeitet werden, bevor sich die Modellgewichte ändern. Für jeden Datenpunkt werden eine SHG-Matrix, Zielvariablen und ein Header verwendet. Zuerst sagt das Modell einen Merkmalsvektor aus der SHG-Matrix voraus. Die SHG-Matrix, die Zielvariablen, der Merkmalsvektor und der Header werden an die Verlustfunktion übergeben. Diese berechnet, wie in Kapitel 3.5 beschrieben, einen Verlustwert. Nun erfolgt der Rückwärtsdurchlauf, in dem der Optimierer die Gradienten berechnet und die Modellgewichte anpasst.

```
1 # itterate over epochs
2 for epoch in range(config.NUM_EPOCHS):
3     # place model into training mode
4     model.train()
5     # After the defined unfreeze_epoch, unfreeze the earlier layers and train the whole
    ↪ Model
6     if (epoch == config.UNFREEZE_EPOCH):
7         logger.info("Unfreezing earlier layers")
8
9         # Unfreeze all layers
10        for param in model.densenet.parameters():
```

```
11     param.requires_grad = True
12
13     # iterate over train data
14     for i, (shg_matrix, label, header) in enumerate(train_loader):
15         ## Load Data ##
16         # send shg_matrix and label data to selected device
17         shg_matrix = shg_matrix.float().to(device)
18         label = label.float().to(device)
19
20         ## Forward pass ##
21         # get the predicted output from the model
22         outputs = model(shg_matrix)
23         # calculate the loss
24         loss = criterion(
25             prediction=outputs,
26             label=label,
27             shg_matrix=shg_matrix,
28             header=header
29         )
30
31         ## Backward pass ##
32         # calculate gradients
33         loss.backward()
34         # Gradient clipping
35         torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
36         # step the optimizer
37         optimizer.step()
38         optimizer.zero_grad()
39
40         # log information for current batch
41         logger.info(f"Epoch {epoch+1} / {config.NUM_EPOCHS}, Step {i+1} /
42             ↪ {NUM_STEPS_PER_EPOCH}, Loss = {loss.item():.10e}, LR =
43             ↪ {scheduler.get_last_lr()[0]:.4e}")
44         # Write loss into array
45         training_losses.append(loss.item())
```

```
45     # Step the learning rate
46     scheduler.step()
47     # write new learning rate in variable and save it to list
48     new_lr = scheduler.get_last_lr()[0]
49     learning_rates.append(new_lr)
```

Listing 18: Trainingsloop bei unüberwachtem Training
Quelle: Eigene Darstellung, Auszug aus `./training.py`

Nachdem alle Trainingsdaten einmal durchlaufen wurden, erfolgt ein Validierungsschritt. Dieser ist in Listing 19 gezeigt. Hier werden die Validierungsdaten iteriert und die Verlustwerte für diese bestimmt. Im Gegensatz zum Trainingsschritt werden die Validierungsfehler nur gespeichert, um einen Indikator für die Vorhersagegüte zu liefern. Sind alle Epochen durchlaufen, wird ein Testschritt durchgeführt, der identisch zum Validierungsschritt aufgebaut ist.

```
1 logger.info(f"Starting Validation for epoch {epoch+1} / {config.NUM_EPOCHS}")
2 model.eval()  # put model into evaluation mode
3 if validation_size!= 0:
4     with torch.no_grad(): # disable gradient computation for evaluation
5         # itterateover validation data
6         for shg_matrix, label, header in validation_loader:
7             #####
8             ## Load Data ##
9             #####
10            # convert shg_matrix and label to float and send them to the device
11            shg_matrix = shg_matrix.float().to(device)
12            label = label.float().to(device)
13
14            #####
15            ## Forward pass ##
16            #####
17            # calculate prediction
18            outputs = model(shg_matrix)
19            # calculatate validation loss
20            validation_loss = criterion(
```

```
21         prediction=outputs,
22         label=label,
23         shg_matrix=shg_matrix,
24         header=header
25     )
26     # place validation loss into list
27     validation_losses.append(validation_loss.item())
28
29     # calculate the mean validation loss
30     avg_val_loss = np.mean(validation_losses) # calculate validation loss for this
        ↪ epoch
31     logger.info(f"Validation Loss: {avg_val_loss:.10e}")
```

Listing 19: Validierungsloop bei unüberwachtem Training
Quelle: Eigene Darstellung, Auszug aus `./training.py`

Unüberwachtes Lernen

Die verwendete Verlustfunktion ist in der Lage, auch ohne Zielvariable mithilfe des FROG-Fehlers ein Maß für die Güte der Vorhersage zu bestimmen. Der FROG-Fehler kann verwendet werden, um ein sogenanntes unüberwachtes Lernen auf Spektrogrammen zu betreiben, deren zugehöriges Zeitprofil nicht bekannt ist. Hierbei wird auf die in Kapitel 2.4 diskutierte Variante zur Rückgewinnung eines Zeitprofils aus einem Spektrogramm zurückgegriffen. Das neuronale Netz sagt ein Zeitprofil voraus. Aus diesem wird eine neue SHG-Matrix bestimmt und durch Vergleich mit dem ursprünglichen Spektrogramm der FROG-Fehler berechnet. Dieser wird vom Optimierer verwendet, um die Gewichte des Netzes anzupassen. In diesem Fall werden die MSE-Gewichte der Verlustfunktion, welche in Kapitel 3.5 beschrieben ist, mit Null initialisiert und der Funktion außerdem keine Zielvariablen mehr übergeben. [23, S. 668]

Das überwachte Training ist der zuverlässigere Weg zu guten Vorhersagen. Daher soll das unüberwachte Training dazu dienen, ein mit künstlich erzeugten Daten vortrainiertes neuronales Netz auf die Vorhersage experimentell ermittelter Daten vorzubereiten. Auch kann hiermit ein Modell ergänzt werden, welches bereits überwacht gelernt hat. Dazu wird zunächst ein neuronales Netz mit künstlichen Daten trainiert. Anschließend werden die so ermittelten Gewichte geladen und das Netz wird mit einer geringeren Anzahl an

experimentell ermittelten Datenpunkten unüberwacht trainiert. Da sowohl die Qualität als auch die Quantität der experimentellen Daten hierfür noch nicht ausreicht, konnte dieser Ansatz bisher nicht verifiziert werden.

3.3.2 Learning Rate Scheduling

Learning-Rate-Scheduling ändert die gewählte Lernrate während des Trainings. Für das Training des neuronalen Netzes wird ein 1Cycle Lernratenscheduler verwendet. Dabei wird die Lernrate von einem initialen Wert langsam auf einen Maximalwert erhöht. Anschließend sinkt die Lernrate wieder bis auf einen Minimalwert. Die Veränderung der Lernrate folgt dabei einem cosinusförmigen Verlauf. Dieser ist in Abbildung 3.7 gezeigt.[14]

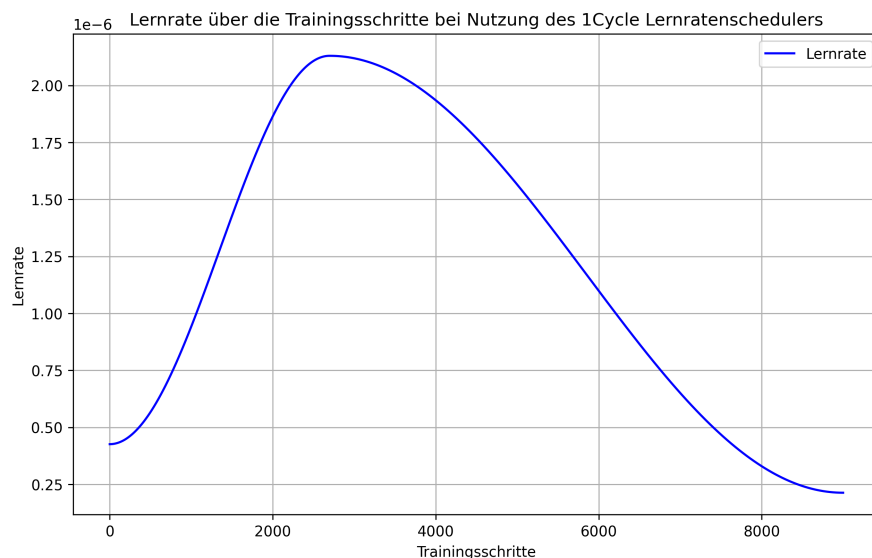


Abbildung 3.7: Beispielverlauf der Lernrate bei Verwendung eines 1Cycle Lernratenschedulers

Quelle: Eigene Darstellung

Es hat sich gezeigt, dass dies für DenseNet-Architekturen zu einer schnelleren Konvergenz führt als eine lineare Veränderung der Lernrate [17, S. 7]. Zur Bestimmung einer passenden Lernrate wird ein Skript verwendet, das die Verlustfunktion für eine kleine Lernrate evaluiert. Mit jedem Batch wird die Lernrate erhöht und dann der Verlust über die Lernrate aufgetragen. Anschließend wird die Maximallernrate `start_lr` eine Zehnerpotenz größer gewählt als die Lernrate, die zum minimalen Fehler geführt hat. [18]

Die Wahl der Batch-Größe kann große Auswirkungen auf die Trainingsgeschwindigkeit haben. In der Publikation von Tanksale, die den 1Cycle Lernratenscheduler vorstellt, werden Batch-Größen von 256 bis 1536 verwendet [18, S. 17]. Aufgrund des geringen Grafikspeichers, der auf der zum Training verwendeten Grafikkarte zur Verfügung steht, wird in der vorliegenden Arbeit für alle beschriebenen Trainingsvorgänge die maximal nutzbare Batch-Größe von 10 genutzt. Da diese im Vergleich sehr klein ist, sollten in Zukunft größere Batch-Größen auf anderer Hardware getestet werden.

3.4 Optimierer und Regularisierung

Als Optimierer wird der *AdamW*-Optimierer gewählt. Dieser basiert auf der Funktionsweise des *Adam*-Algorithmus und erweitert diesen um eine von der Gradientenberechnung entkoppelte Regularisierung. Diese Gewichtszurücknahme (engl. weight decay) ist ein häufig mit der L2-Regularisierung verglichenes Regularisierungsverfahren, das eine Überanpassung beim Trainieren des Netzes verhindern soll. Diese Überanpassung ist ein häufiges Problem beim Trainieren großer Netze mit *Adam*. *AdamW* ist heute einer der am häufigsten verwendeten Optimierer, da er zu robuster und stabiler Konvergenz führt. So wurde ermittelt, dass *AdamW* zu einem besseren Validierungsfehler, einer schnelleren Konvergenz und einer besseren Generalisierung führt als *Adam* oder Statistischer Gradientenabstieg (SGD). [5]

Der *AdamW*-Optimierer benötigt Parameter θ , eine Lernrate α und einen Wert für die Gewichtszurücknahme λ_w . Zunächst werden einige Werte initialisiert. Die ersten Parameter, das erste Moment m_0 , welches den Erwartungswert der Gradienten repräsentiert, und das zweite Moment v_0 , welches wiederum die Varianz der Gradienten abbildet, werden auf Null gesetzt. Anschließend werden einige Hyperparameter definiert. Zum einen die Lernrate α , aber auch ein exponentieller Abfall $\beta_{1,2}$ der ersten und zweiten Momente und eine kleine Konstante ϵ , die eine Division durch Null verhindert. Für jeden Trainingsschritt t werden nun die Gradienten nach Gleichung 3.16 berechnet. Aus diesen können wiederum das erste und zweite Moment berechnet werden. Dies ist in Gleichung 3.17 und 3.18 abgebildet. [5]

$$g_t = \nabla_{\theta_t} f(\theta) \quad (3.16)$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad (3.17)$$

$$v_t = \beta_2 m_{t-1} + (1 - \beta_2)(g_t \odot g_t) \quad (3.18)$$

mit: $g_t \odot g_t =$ elementweise Multiplikation mit sich selbst

Nun kann die in Gleichung 3.19 und 3.20 dargestellte Bias-Korrektur durchgeführt werden. Abschließend werden die Parameter θ_{t+1} für den nächsten Schritt berechnet. Hierbei wird die Gewichtszurücknahme in einem separaten additiven Term $\lambda_w \theta_t$ durchgeführt, wie in Gleichung 3.21 zu sehen ist. [5]

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad (3.19)$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad (3.20)$$

$$\theta_{t+1} = \theta_t - \alpha \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} + \lambda_w \theta_t \right) \quad (3.21)$$

3.5 Verlustfunktion

3.5.1 Allgemeines

Zentraler Teil des Trainingsprozesses ist die verwendete Verlustfunktion. Die hier genutzte Funktion soll sowohl für das überwachte als auch für das unüberwachte Lernen verwendet werden. Gesteuert wird dies, indem bei der Initialisierung der Funktion die Flag `use_label` auf `False` gesetzt wird. Zusätzlich kann beim überwachten Lernen der Fehler unterdrückt werden, der durch hohe Werte der Zielvariablen außerhalb des Pulses entsteht.

Vereinfacht lässt sich der Aufbau der Verlustfunktion wie folgt beschreiben. Zunächst wird, wie in Zeile 4 von Listing 20 gezeigt, die in Kapitel 2.6 erläuterte Hilbert-Transformation durchgeführt, um aus dem vorhergesagten Realteil des Signal dessen Imaginärteil zu bestimmen. Aus diesem werden nun in den Zeilen 23 und 27 die Phase und die Intensität des Zielsignals bestimmt, sollte das neuronale Netz überwacht lernen. In den Zeilen 30 bis 31 werden die Variablen zur Speicherung der unterschiedlichen Verluste initiiert.

```
1 prediction_analytical = torch.zeros(batch_size, half_size, dtype=torch.complex64)
2
3 for i in range(batch_size):
4     prediction_analytical[i] = hilbert(prediction[i], plot=False).to(device)
5
6     # get real and imaginary parts of predictions
7     prediction_real = prediction_analytical.real.to(device)
8     prediction_imag = prediction_analytical.imag.to(device)
9     # remove the ambiguities and get corrected prediction real and imaginary parts
10    prediction_tensor = torch.cat([prediction_real, prediction_imag], dim=1)
11    prediction_tensor = self.remove_ambiguities(prediction_tensor)
12    prediction_real = prediction_tensor[:, :half_size].to(device)
13    prediction_imag = prediction_tensor[:, half_size:].to(device)
14
15
16    if self.use_label:
17        # get real and imaginary parts of labels
18        label_real = label[:, :half_size].to(device)
19        label_imag = label[:, half_size:].to(device)
20
21        # calculate intensities
22        label_intensity = label_real**2 + label_imag**2
23        prediction_intensity = prediction_real**2 + prediction_imag**2
24
25        # calculate phases
26        label_phase = torch.atan2(label_imag, label_real)
27        prediction_phase = torch.atan2(prediction_imag, prediction_real)
28
29        # initialize losses
30        loss = 0.0
31        mse_loss = 0.0
32        frog_error = 0.0
```

Listing 20: Ausschnitt der Verlustfunktion

Quelle: Eigene Darstellung, Auszug aus `./modules/loss.py`

3.5.2 Gewichtete MSE Funktion

Im Falle des überwachten Trainierens des Netzes werden nun die MSE-Fehler der verschiedenen Parameter berechnet. Der hierfür relevante Teil der Verlustfunktion ist in Listing 21 gezeigt. Zunächst wird, wie in den Zeilen 4 bis 23 gezeigt, eine Maske erstellt, die nur Eins wird, wenn Real- oder Imaginärteil über einem definierbaren Grenzwert liegen. Dann wird jeweils der erste und der letzte Index in dieser Maske bestimmt, der gleich Eins ist. Dies ist in den Zeilen 26 und 27 zu sehen. Aufgrund geringer Signalamplituden außerhalb des Pulses, kommt es in diesem Bereich schnell zu Fehlern bei der Phasenvorhersage. Daher wird, wie in den Zeilen 29 und 30 zu sehen, der Fehler der Phase außerhalb des Pulses auf Null gesetzt. In den Zeilen 36 bis 44 werden die MSE-Fehler sowohl für die Intensität und Phase als auch für den Real- und Imaginärteil bestimmt. Dazu wird Gleichung 3.22 genutzt. Die in Zeile 32 und 33 definierte Maske wird nun genutzt, um die MSE-Fehler außerhalb des Pulses mit einem Bestrafungsfaktor zu bestrafen. Unter Berücksichtigung der definierten Gewichte wird nun ein Gesamtfehler aus den MSE-Fehlern berechnet. Dies ist in Zeile 54 gezeigt. In einem letzten Schritt wird nun eine Summe aus FROG-Fehler und MSE-Fehlern gebildet, wie in Zeile 57 zu sehen ist.

$$MSE = \frac{1}{N} \sum_{i=1}^N (Y_i - \hat{Y}_i)^2 \quad (3.22)$$

```
1 if self.use_label:
2
3     # Create masks for all absolute values higher than the threshold
4     mask_real_threshold = abs(label_real[i]) > self.pulse_threshold
5     mask_imag_threshold = abs(label_imag[i]) > self.pulse_threshold
6
7     # if any real value is greater than the threshold
8     if torch.any(mask_real_threshold):
9         # get the first and last index, where a value is greater than the threshold
10        first_significant_idx_real = torch.nonzero(mask_real_threshold).min().item()
11        last_significant_idx_real = torch.nonzero(mask_real_threshold).max().item()
12    else:
13        first_significant_idx_real = 0
```

```
14     last_significant_idx_real = half_size - 1
15
16     # if any imaginary value is greater than the threshold
17     if torch.any(mask_imag_threshold):
18         # get the first and last index, where a value is greater than the threshold
19         first_significant_idx_imag = torch.nonzero(mask_imag_threshold).min().item()
20         last_significant_idx_imag = torch.nonzero(mask_imag_threshold).max().item()
21     else:
22         first_significant_idx_imag = 0
23         last_significant_idx_imag = half_size - 1
24
25     # determine the lower first significant index
26     first_significant_idx = min(first_significant_idx_real,
27     ↪ first_significant_idx_imag) # determine the higher last significant index
28     last_significant_idx = max(last_significant_idx_real, last_significant_idx_imag)
29     # create the phase mask
30     phase_mask = torch.zeros(half_size).to(device)
31     phase_mask[first_significant_idx:last_significant_idx] = 1
32     # create the intensity_mse mask
33     pulse_mask = torch.ones(half_size).to(device)
34     pulse_mask[first_significant_idx:last_significant_idx] = self.penalty
35
36     # Calculate MSE for the real and imaginary part
37     mse_real = (prediction_real[i] - label_real[i]) ** 2
38     mse_real = mse_real * pulse_mask
39     mse_imag = (prediction_imag[i] - label_imag[i]) ** 2
40     mse_imag = mse_imag * pulse_mask
41
42     # Calculate MSE for the intensity and phase
43     mse_intensity = (prediction_intensity[i] - label_intensity[i]) ** 2
44     mse_intensity = mse_intensity * pulse_mask
45     mse_phase = (prediction_phase[i] - label_phase[i]) ** 2
46     # Use the phase mask for phase blanking
47     mse_phase = mse_phase * phase_mask
48
49     # calculate weighted means for MSE
```

```
49     mse_real_mean = mse_real.mean()*self.real_weight
50     mse_imag_mean = mse_imag.mean()*self.imag_weight
51     mse_intensity_mean = mse_intensity.mean()*self.intensity_weight
52     mse_phase_mean = mse_phase.mean()*self.phase_weight
53     # calculate weighted mse loss
54     mse_loss += (mse_real_mean + mse_imag_mean + mse_intensity_mean +
55                 ↪ mse_phase_mean) / self.mse_weight_sum
56
57 # calculate weighted mean loss of mse loss and frog error
58 loss += mse_loss + frog_error*self.frog_error_weight
```

Listing 21: Ausschnitt der Verlustfunktion zum Berechnen des MSE-Fehlers
Quelle: Eigene Darstellung, Auszug aus `./modules/loss.py`

3.5.3 FROG-Fehler

Zur Bestimmung des FROG-Fehlers wird zunächst, wie in Zeile 8 von Listing 22 gezeigt, aus dem analytischen Signal eine SHG-Matrix erzeugt. Diese wird in ein Spektrogramm transformiert, welches Intensitätswerte abbildet und eine Wellenlängenachse besitzt. Dies geschieht wie in Kapitel 2.3 beschrieben. Das Spektrogramm wird auf die Achsenabschnitte der Eingangsmatrix neu abgetastet und entsprechend normalisiert. Dies ist in Zeile 14 bis 22 zu sehen. Abschließend wird in Zeile 25 der FROG-Fehler bestimmt und dann zum restlichen Fehler addiert.

```
1 if self.frog_error_weight != 0.0:
2     # just get current index from batch
3     original_shg = shg_matrix[i]
4     original_header = header[i]
5     # get original SHG-matrix (without 3 identical channels)
6     original_shg = original_shg[0]
7     # get a wavelength SHG-Matrix from the analytical signal and the header
8     predicted_shg, new_header = createSHGmatFromAnalytical(
9         analytical_signal= prediction_analytical[i],
10        header=original_header
11    )
```

```
12     # resample to correct size
13     predicted_shg_data = [predicted_shg, new_header]
14     resample_outputs = self.shg_resample(predicted_shg_data)
15     resample_outputs = self.shg_create3channels(resample_outputs)
16     __, prediction_header, predicted_shg, __, __ = resample_outputs
17
18     # get only one channel
19     predicted_shg = predicted_shg[0,:,:]
20     # normalize SHG-matrix
21     predicted_shg = helper.normalizeSHGmatrix(predicted_shg)
22     original_shg = helper.normalizeSHGmatrix(original_shg)
23
24     # calculate_frog_error
25     frog_error = calcFrogError(original_shg, predicted_shg)
```

Listing 22: Ausschnitt der Verlustfunktion zum Berechnen des FROG-Fehlers
Quelle: Eigene Darstellung, Auszug aus `./modules/loss.py`

4 Auswertung der Trainingsergebnisse

4.1 Allgemeines

Um die Trainingsergebnisse auszuwerten, soll die Auswirkung verschiedener Parameter auf das Vorhersageergebnis untersucht werden. Dazu wird der Einfluss des Lernratenschedulers, des TBDs, des Rauschens, der Hyperparameter und der Anzahl an Trainingsepochen erläutert werden. Abschließend werden die Ergebnisse mit den Vorhersagen der Rekonstruktionssoftware von Swamp-Optics verglichen.

4.2 Einfluss des Schedulers auf die Vorhersage

Zur Bewertung des Einflusses des Lernratenschedulers auf das Vorhersageergebnis sollen drei Ansätze miteinander verglichen werden. Um eine Vergleichbarkeit der Trainings- und Testverluste zu gewährleisten, wurden für alle drei Ansätze identische Hyperparameter gewählt. Diese sind in Tabelle A.2 im Anhang gezeigt. Das neuronale Netz wurde zunächst mit einer konstanten Lernrate von $2,13 \cdot 10^{-6}$ trainiert. In einem zweiten Ansatz wird das Netz so trainiert, dass die Lernrate nach jedem Trainingsschritt linear von $2,13 \cdot 10^{-6}$ auf $2,13 \cdot 10^{-5}$ ansteigt. Als letzter Ansatz wird beim Trainieren der in Kapitel 3.3.2 beschriebene 1Cycle Lernratenscheduler verwendet, wobei die Lernrate hier bei $2,13 \cdot 10^{-6}$ startet, dann auf $2,13 \cdot 10^{-5}$ ansteigt, um anschließend wieder auf $2,13 \cdot 10^{-6}$ abzusinken.

Um einen Vergleich zwischen diesen Ansätzen durchzuführen, wird zunächst der Verlauf der Trainingsverluste gegenübergestellt. Hierzu ist in Abbildung 4.1 der Verlauf des Trainings- und des Testfehlers dargestellt. Es lässt sich erkennen, dass sich der Verlauf bei konstanter und linear ansteigender Lernrate kaum voneinander unterscheidet. Bei Nutzung des 1Cycle Lernratenschedulers hingegen ist ein deutlich schnelleres Absinken des Trainingsfehlers zu erkennen. Das arithmetische Mittel des Testfehlers unterstreicht dies. So liegt dieses beim Training mit konstanter Lernrate bei 1,538 und bei einer linear ansteigenden Lernrate bei 1,623. Das arithmetische Mittel des Testfehlers bei Nutzung des 1Cycle Lernratenschedulers hingegen liegt mit 1,205 um 0,333 unter dem

Testfehler bei konstanter Lernrate und sogar um 0,418 unter dem Testfehler bei einer linear ansteigender Lernrate.

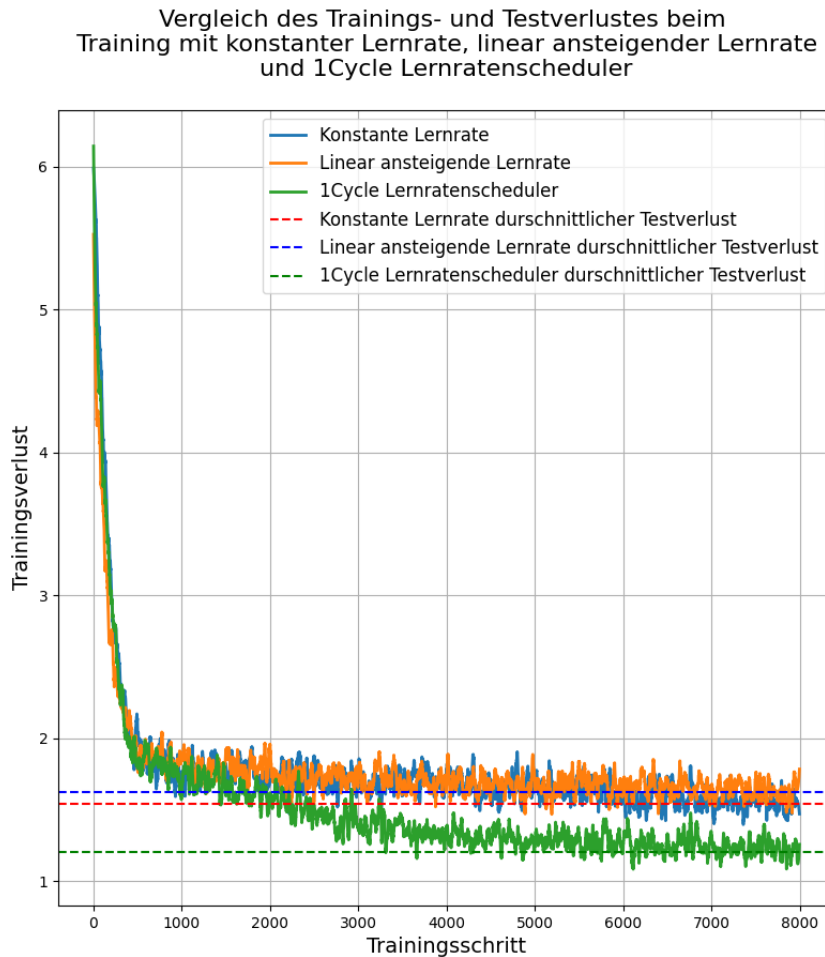


Abbildung 4.1: Vergleich des Trainings- und Testfehlers beim Training mit konstanter Lernrate, linear ansteigender Lernrate und 1Cycle Lernratenscheduler
Quelle: Eigene Darstellung

In Abbildung 4.2 sind Boxplots der Testverluste abgebildet. Anhand dieser sollen statistische Kenngrößen miteinander verglichen werden. Es ist zu erkennen, dass Median und arithmetisches Mittel den höchsten Wert bei einer linear ansteigenden Lernrate annehmen. Ebenfalls fallen hier das erste und dritte Quartil sowie der minimale bzw. maximale Verlust am größten aus. Ähnliche, aber etwas niedrigere Werte erreicht das Training mit konstanter Lernrate. Die niedrigsten und damit besten Werte liefert das Training mit

einem 1Cycle Lernratenscheduler. Hier liegt der minimale Wert um etwa 0,25 unter dem minimalen Wert bei konstanter Lernrate. Außerdem liegt auch der maximale mit etwa 0,2 unter dem maximalen Wert bei konstanter Lernrate. Auch die Quartile sowie Median und arithmetisches Mittel liegen signifikant unter den Werten der anderen Verfahren. Hieraus lässt sich folgern, dass von den getesteten Verfahren zur Festlegung der Lernraten der 1Cycle Lernratenscheduler am besten abschneidet. Er wird im Folgenden für alle weiteren Tests genutzt.

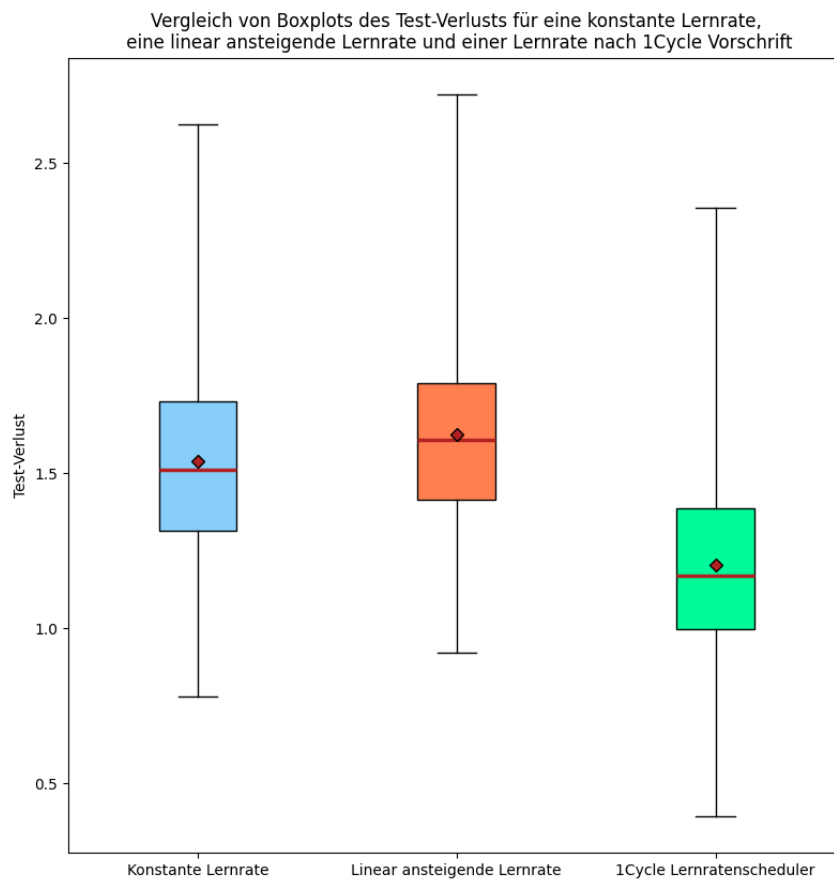


Abbildung 4.2: Boxplots des Testfehlers beim Training mit konstanter Lernrate, linear ansteigender Lernrate und 1Cycle Lernratenscheduler

Quelle: Eigene Darstellung

4.3 Einfluss des TBD auf die Vorhersage

In Abbildung 4.3 ist ein Histogramm des TBD aller Spektrogramme dargestellt. Der minimale TBD-Wert liegt bei 0,25 und der maximale bei 5,414. Das 90 %-Perzentil liegt bei 1,063 und das 50 %-Perzentil bei 0,5514. Es liegen also 90 % bzw. 50 % der Datenpunkte unter diesen Werten.

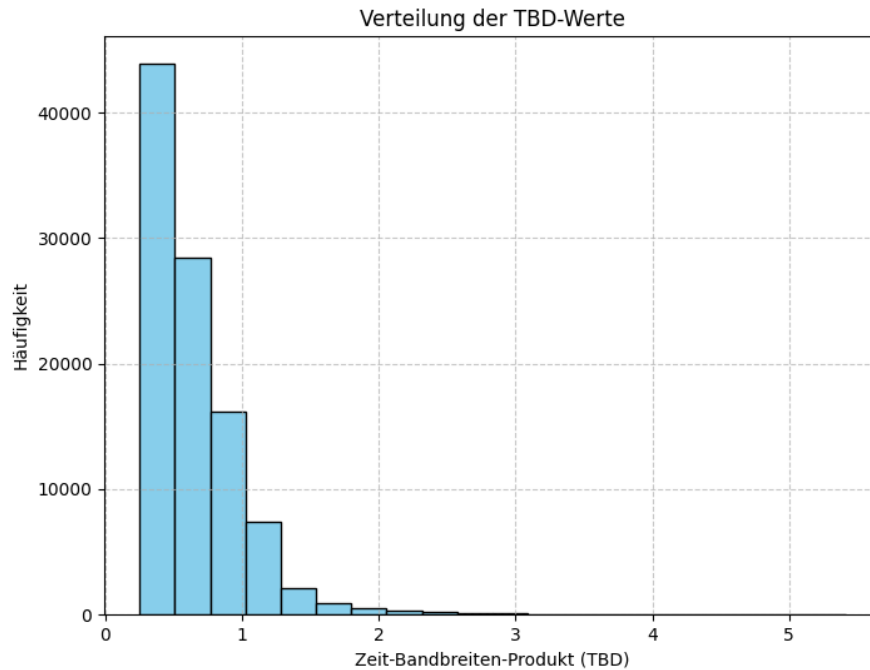


Abbildung 4.3: Histogramm der TBD-Werte im vorliegenden Datensatz
Quelle: Eigene Darstellung

Um den Einfluss des TBD auf die Vorhersage zu bewerten, wird zunächst der Verlust des Testfehlers über den TBD-Wert der Spektrogramme aufgetragen. Hierbei wurden 100 Spektrogramme untersucht. Dies ist in Abbildung 4.4 gezeigt. Es lässt sich erkennen, dass der Fehler mit zunehmendem TBD ansteigt. In den untersuchten Fällen lässt sich dieser mit einer linearen Gleichung approximieren. Sie ist in der Abbildung ebenfalls dargestellt.

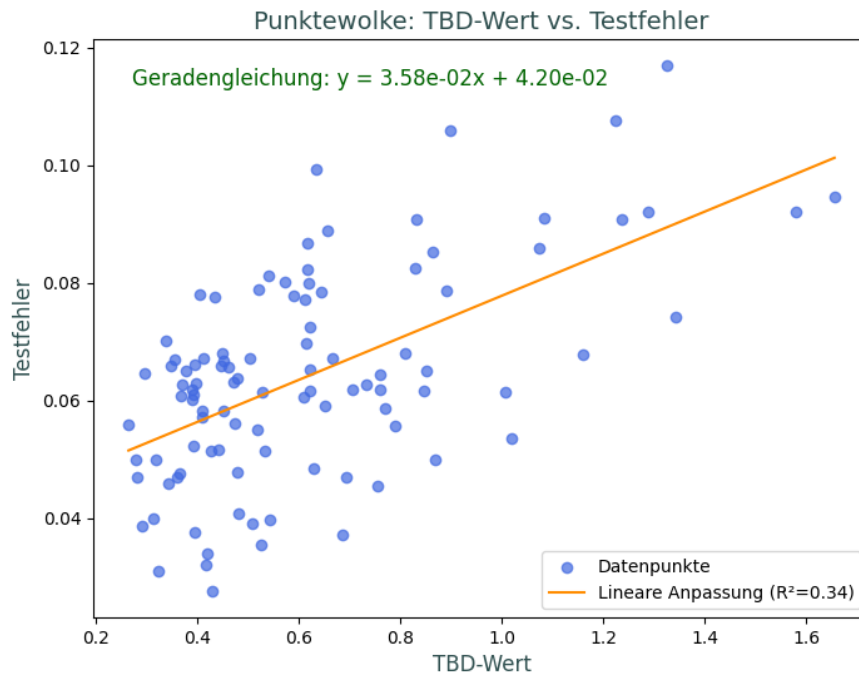


Abbildung 4.4: Testfehler eines neuronalen Netzes über den TBD-Wert von 100 Spektrogrammen sowie Approximation als lineare Gleichung
Quelle: Eigene Darstellung

Da mit zunehmendem TBD der Testfehler ansteigt, wird im Folgenden die Veränderung des Testfehlers betrachtet, wenn nur Spektrogramme mit einem definierbaren maximalen TBD-Wert beim Training berücksichtigt werden. In Abbildung 4.5 wird der Testfehler des gesamten Datensatzes mit dem Testfehler bei Spektrogrammen mit TBD-Werten im 90 %- bzw. 50 %-Perzentil verglichen. Der Trainingsdatensatz umfasst hierbei 49 006 Datenpunkte und der Testdatensatz 1000 Datenpunkte. Datenpunkte liegen bei einem TBD-Wert von unter 1,063 im 90 %-Perzentil und bei einem TBD-Wert von unter 0,5514 im 50 %-Perzentil. Um den gesamten Datensatz zu berücksichtigen, wird ein TBD von 20 gewählt. Die Hyperparameter sind ansonsten identisch. Sie sind in Tabelle A.3 im Anhang zu erkennen.

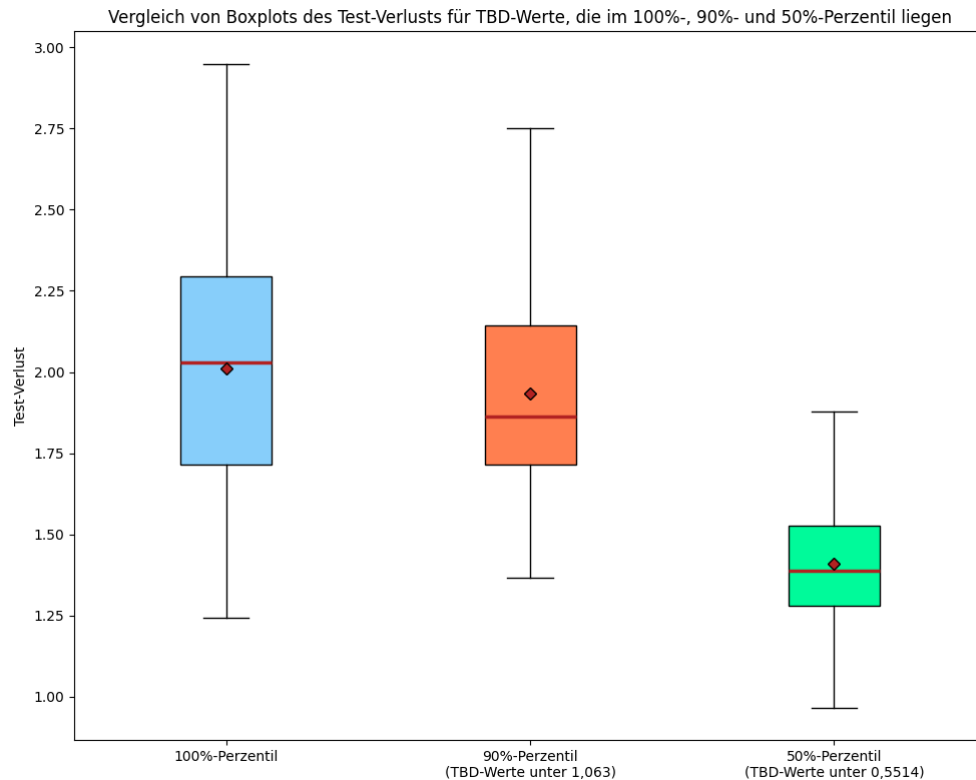


Abbildung 4.5: Boxplots des Testfehlers beim Training mit einem maximalen TBD-Wert von 20, 1,063 und 0,5514

Quelle: Eigene Darstellung

Es lässt sich erkennen, dass der Testfehler sinkt, je geringer der TBD-Wert ist. Dies lässt sich darauf zurückführen, dass der TBD-Wert ein direktes Maß für die Komplexität des Pulses ist. Zum einen müssen komplexere Pulse vorausgesagt werden, was einen höheren Trainingsaufwand zur Folge hat. Des Weiteren enthält der Testdatensatz bei einem größeren maximalen TBD-Wert komplexere Pulse, die zu einem höheren Fehler führen. Hier ist also eine Abwägung anhand des vorliegenden Datensatzes vorzunehmen. Besitzen experimentell erstellte Pulse im Durchschnitt ein hohes TBD, kann es sich lohnen, das neuronale Netz ebenfalls unter Berücksichtigung komplexerer Pulse zu trainieren. Durch das Trainieren auf komplexen und weniger komplexen Pulsen wird außerdem die Generalisierbarkeit des Netzes erhöht.

4.4 Einfluss von Rauschen auf die Vorhersage

Das in dieser Untersuchung erarbeitete neuronale Netz soll zu einem späteren Zeitpunkt verwendet werden, um die Intensität und Phase eines Signals aus experimentell gewonnenen Spektrogrammen vorauszusagen. Da diese Rauschen enthalten, soll die Empfindlichkeit des Netzes auf Rauschen untersucht werden. Hierzu wird das neuronale Netz zunächst mit einem Datensatz ohne Rauschen trainiert. Anschließend wird ein Netz mit identischen Hyperparametern, aber unterschiedlich stark verrauschten Datensätzen trainiert. Diese werden ausgewählt, indem in `./modules/config.py` die Variable `'ShgFilename'` gesetzt wird. `'as_gn00.dat'` enthält Spektrogramme ohne Rauschen und `'as_gn01.dat'`, `'as_gn10.dat'` und `'as_gn30.dat'` enthalten Spektrogramme mit unterschiedlich starkem Rauschen. Die verschiedenen Rauschstufen werden in Abbildung 4.6 dargestellt.

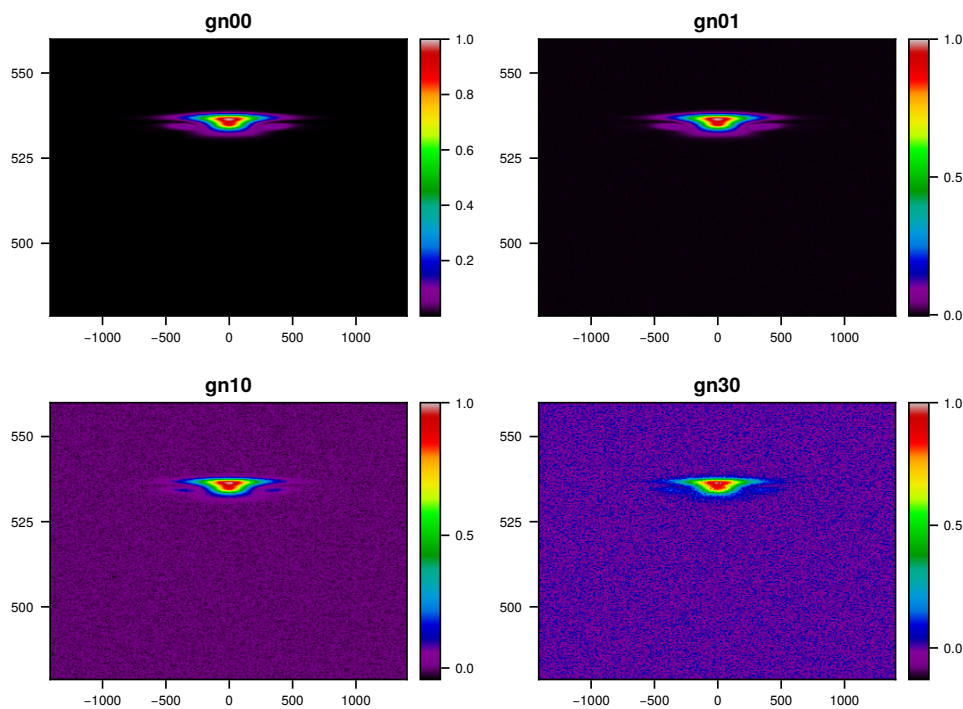


Abbildung 4.6: Unterschiedlich stark verrauschte Spektrogramme
Quelle: Datensatz `'grid_256_v4'`

Es ist hierbei zu erwarten, dass mit jedem Anstieg des Rauschens auch der Testfehler des Netzes steigt. In Abbildung 4.7 sind Boxplots für den Testfehler der verschiedenen

Rauschstufen angegeben. Ihre Hyperparameter sind in Tabelle A.1 im Anhang angegeben. Es ist zu erkennen, dass wie erwartet der Testfehler mit stärkerem Rauschen zunimmt. Die niedrigsten Rauschwerte bleiben zwischen *'as_gn00.dat'*, *'as_gn01.dat'* und *'as_gn10.dat'* nahezu identisch. Auch das arithmetische Mittel, der Median und das erste bzw. dritte Quartil der Testfehler steigen nur leicht bei der Verwendung der verrauschten Spektrogramme. Zwischen *'as_gn01.dat'* und *'as_gn10.dat'* kommt es zu kaum einer Veränderung. Der größte Unterschied tritt beim maximalen Testfehler auf, der mit jeder Rauschstufe steigt. Die am stärksten verrauschten Spektrogramme *'as_gn30.dat'* führen zu einer stärkeren Verschlechterung des Ergebnisses. Die untersuchten statistischen Kennwerte steigen in diesem Fall alle an. Es kommt zu einer Verschlechterung von etwa 0,2 für den Median, das arithmetische Mittel sowie den geringsten Testfehler. Der maximale Testfehler verschlechtert sich sogar um etwa 0,5. Es lässt sich aus diesem Ergebnis schließen, dass ein geringes bzw. moderates Rauschen in den Spektrogrammen nur zu einer geringen Verschlechterung der Vorhersageergebnisse führt. Stärkeres Rauschen hingegen kann die Voraussage stärker stören.

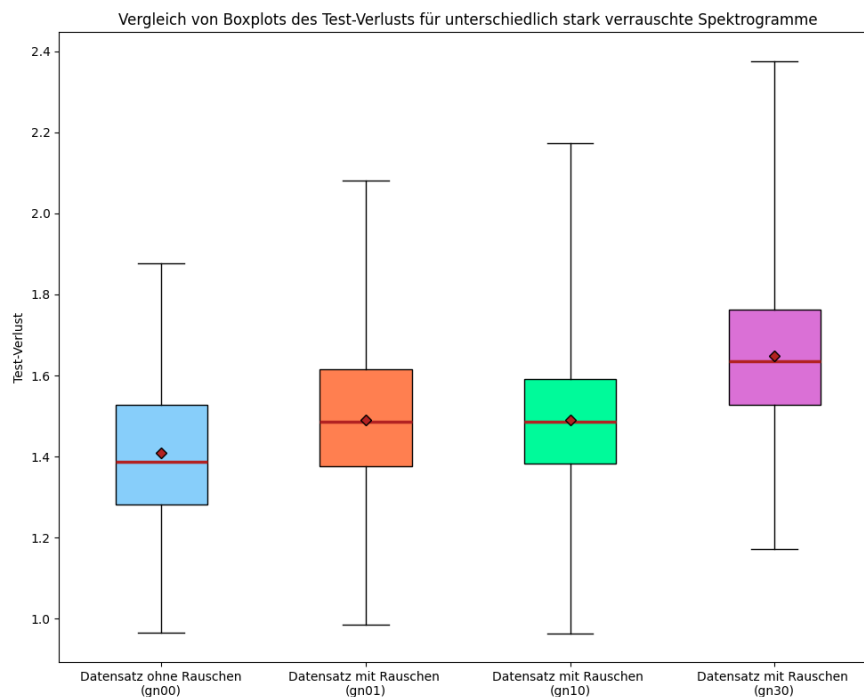


Abbildung 4.7: Boxplots des Testfehlers beim Training mit einem Anstieg des Rauschens der Spektrogramme in 4 Stufen

Quelle: Eigene Darstellung

4.5 Einfluss der Hyperparameter der Verlustfunktion auf die Vorhersage

4.5.1 Einfluss der Hyperparameter der Verlustfunktion beim überwachten Trainieren ohne FROG-Fehler

Um den Einfluss von Hyperparametern auf die Qualität der Vorhersage zu untersuchen, werden zunächst verschiedene Hyperparameter-Konfigurationen überprüft, welche nicht den FROG-Fehler berücksichtigen. Da trotz unterschiedlicher Hyperparameter ein Maß für den Fehler des Trainings benötigt wird, wird nach Beendigung des Trainings der MSE-Fehler des Realteils der Modelle auf identischen Testdaten bestimmt. Werte unter 0,01 werden hierbei geringer gewichtet. Die im Testschritt genutzten Hyperparameter sind in Tabelle A.4 im Anhang abgebildet.

Der Realteil des Signals ist der einzige Teil des Pulses, der direkt mit der Vorhersage verglichen werden kann. Es ist also ein großer Einfluss von dessen Gewicht zu erwarten. Beim überwachten Training des neuronalen Netzes wird er daher immer verwendet. Der Imaginärteil ergibt sich über die Hilbert-Transformation rechnerisch aus dem Realteil. Da sein Informationsgehalt redundant ist, wird erwartet, dass der Einfluss seines Gewichts eher gering ist. Tatsächlich kommt es bei einem Fehler von $4,5699 \cdot 10^{-4}$ hier zum geringsten Testfehler. Die rekonstruierten Signale stellen jedoch keine gute Vorhersage dar, da ihre Amplituden deutlich zu niedrig sind. Der Vergleich eines vorhergesagten Signals mit dessen Zielsignal ist in Abbildung 4.8 dargestellt.

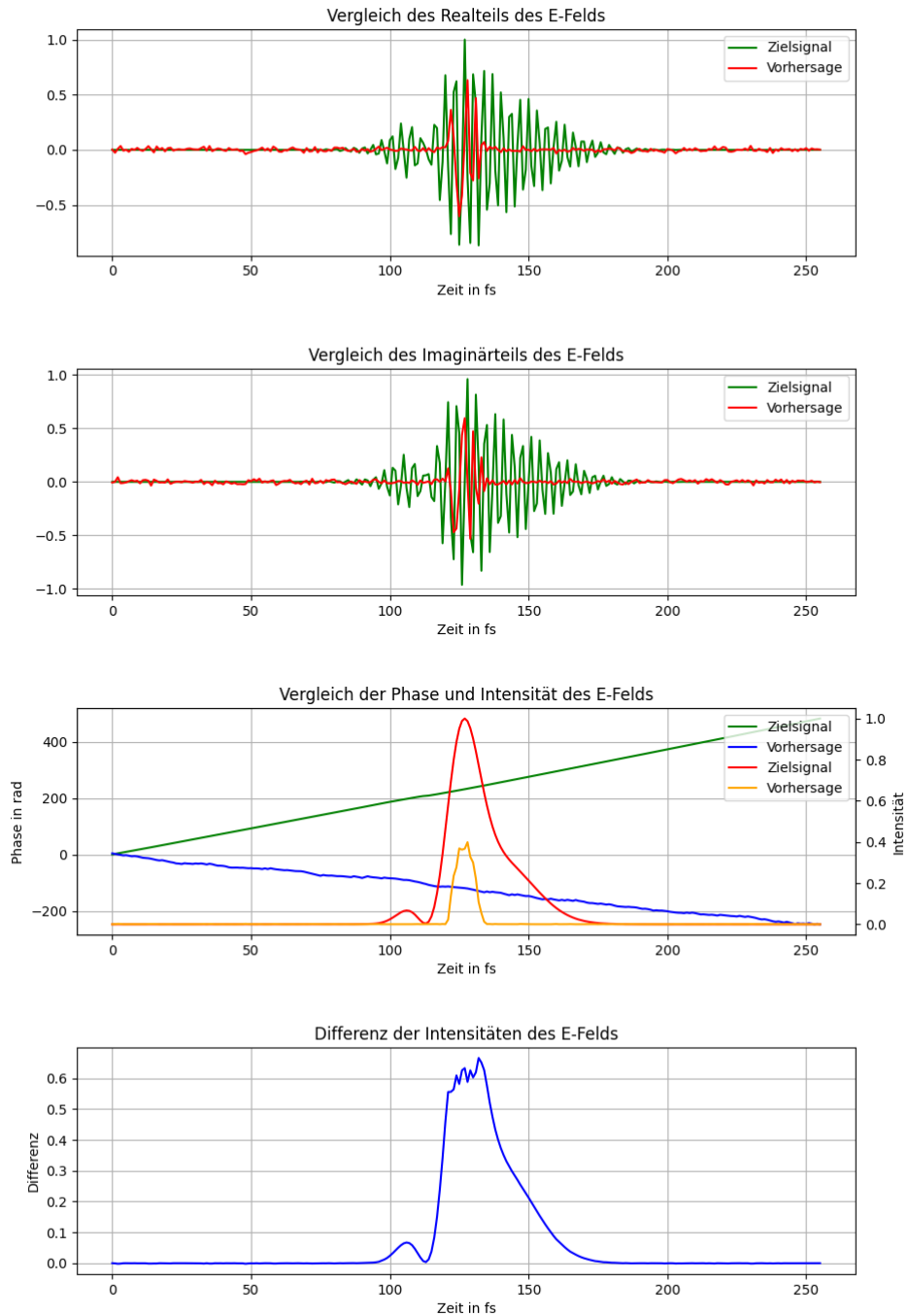


Abbildung 4.8: Vergleich eines Zielsignals mit einer Voraussage bei gleicher Gewichtung von Real- und Imaginärteil

Quelle: Eigene Darstellung

Die Intensität stellt einen weiteren Zusammenhang zwischen Real- und Imaginärteil her. Es ist zu erwarten, dass sich die Voraussage des Realteils vor allem im Zentrum des Signals deutlich verbessert, wenn der MSE der Intensität des Signals berücksichtigt wird. Es wird ein Testfehler von $6,8420 \cdot 10^{-4}$ erreicht. Dieser ist zwar höher als bei Verwendung des Imaginärteils, jedoch kommen die Signalverläufe des Realteils dem des Zielsignals näher als bei der Nutzung des Imaginärteils. Dies ist in Abbildung 4.9 zu sehen.

Das Vorzeichen der Phase kann auf Grund der Ambiguitäten nicht sicher bestimmt werden. Die Funktionen zur Entfernung der Ambiguitäten hängt vom Verlauf des vorhergesagten Pulses ab. Bei größeren Fehlern bei der Vorhersage werden die Ambiguitäten demnach nicht immer entfernt. Dies führt zu starken Sprüngen im Fehler der Phase. Es ist also zu erwarten, dass der MSE der Phase nur bei bereits sehr guten Vorhersagen ein sinnvoller Ansatz ist. In Abbildung 4.10 ist der Testfehler bei starker Gewichtung der Phase in der Verlustfunktion aufgetragen. In der Abbildung lässt sich erkennen, dass der Trainingsfehler nach einem initialen Fall wieder beginnt anzusteigen. Dies weist darauf hin, dass die Phase als Hyperparameter in der vorliegenden Implementierung der Verlustfunktion nicht geeignet ist, um das Vorhersageergebnis zu verbessern. Die zum Trainieren des Netzes genutzten Hyperparameter sind in Tabelle A.5 im Anhang gezeigt.

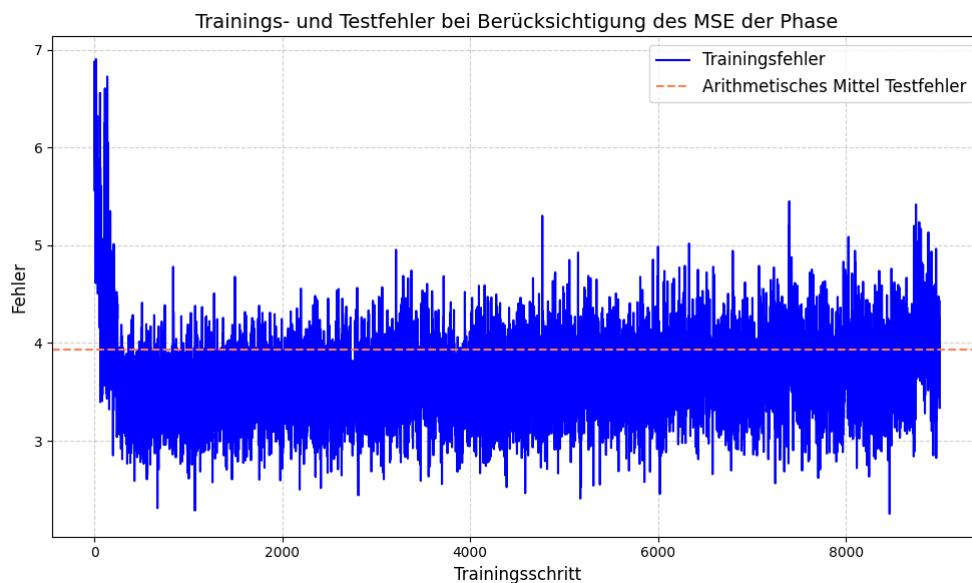


Abbildung 4.10: Trainingsfehler bei Nutzung des MSE der Phase
Quelle: Eigene Darstellung

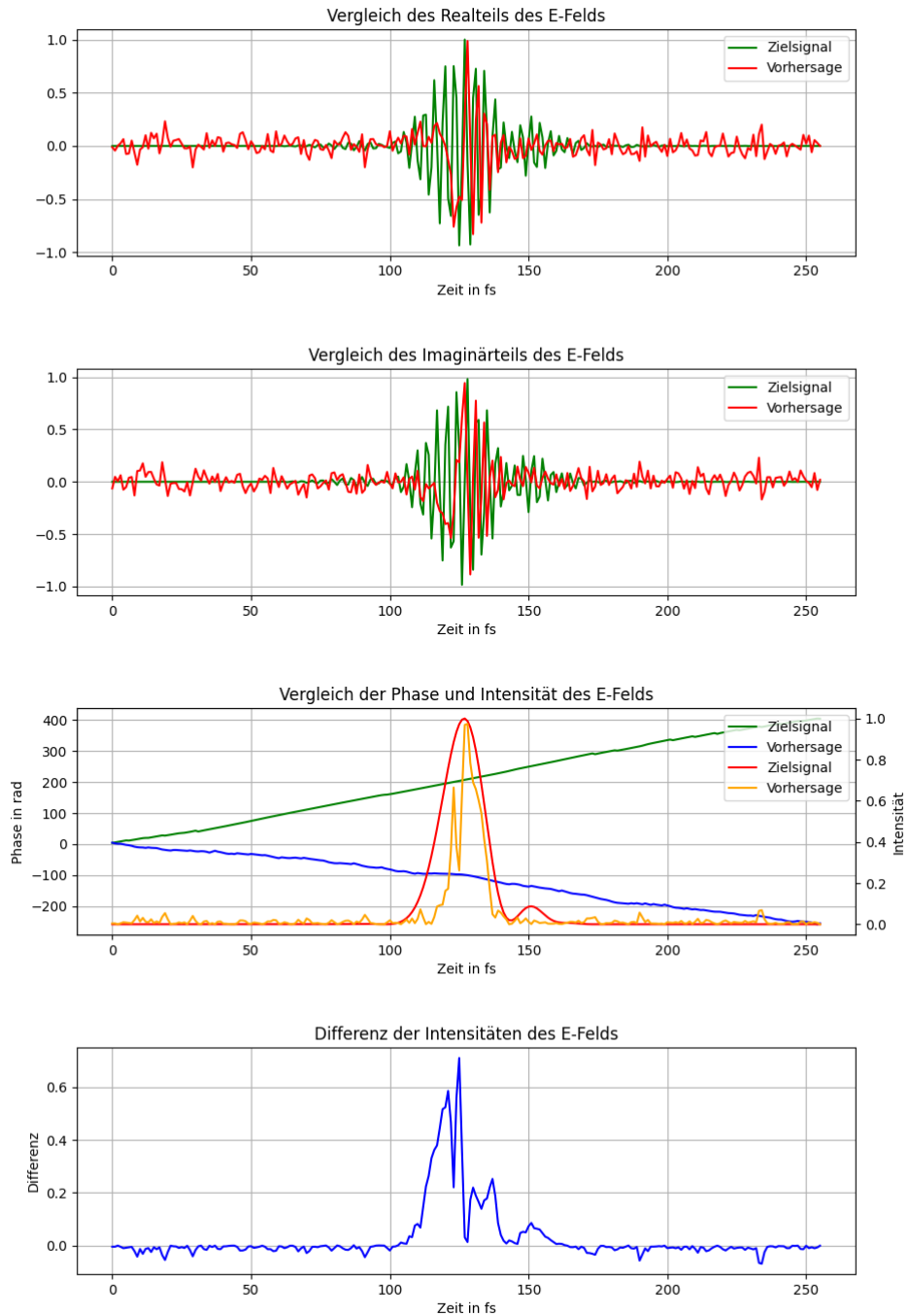


Abbildung 4.9: Vergleich eines Zielsignals mit einer Voraussage bei gleicher Gewichtung von Intensität und Realteil

Quelle: Eigene Darstellung

4.5.2 Einfluss der Hyperparameter der Verlustfunktion beim überwachten Trainieren mit FROG-Fehler

Nun soll zusätzlich der FROG-Fehler in der Verlustfunktion berücksichtigt werden. Hierzu werden die drei verschiedenen Gewichte 10, 5 und 1 für diesen untersucht. Die übrigen Hyperparameter bleiben identisch. Sie sind in Tabelle A.6 im Anhang aufgelistet. In Abbildung 4.11 sind Boxplots des Testfehlers dargestellt. Wie zu erkennen ist, liegt der Testfehler bei einem Gewicht von 5 mit $4,8021 \cdot 10^{-4}$ am niedrigsten. Des Weiteren kommt es bei jedem getesteten Gewicht zu einer Verbesserung gegenüber dem Fehler ohne Berücksichtigung des FROG-Fehlers. Dieser sollte also, wenn möglich, immer berücksichtigt werden.

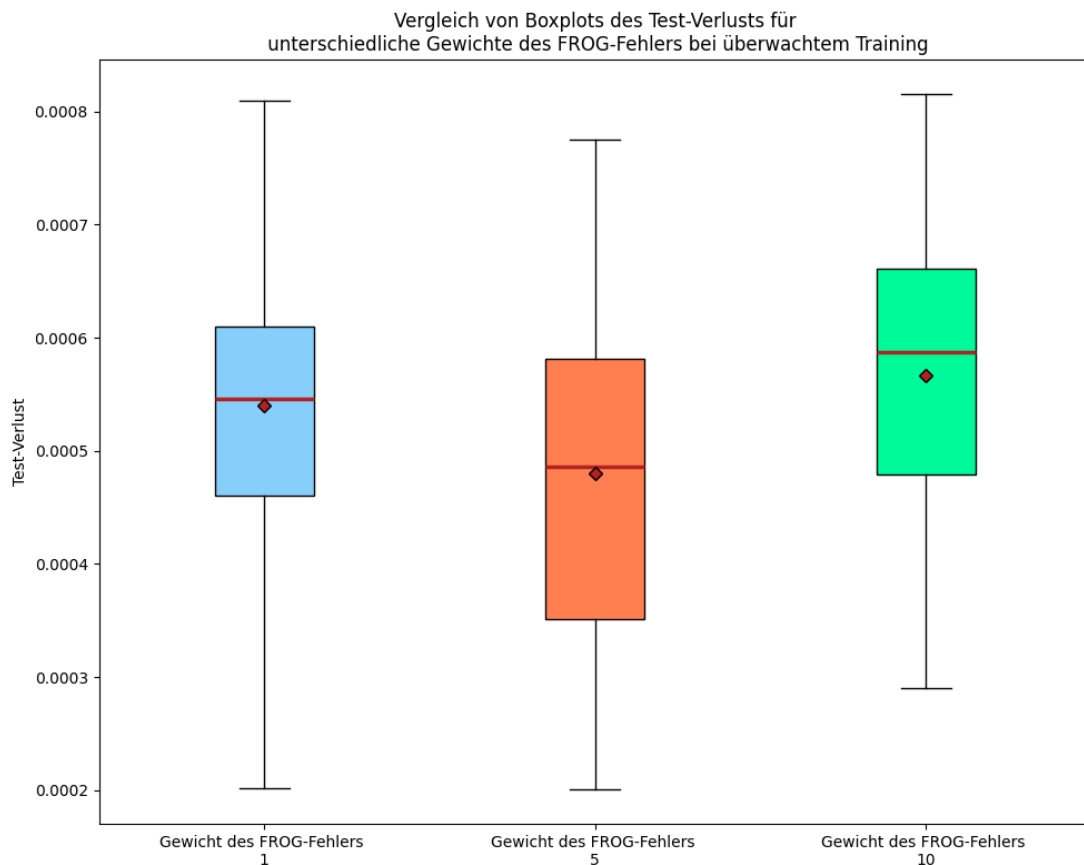


Abbildung 4.11: Boxplots des Testfehlers beim überwachten Training mit Gewicht des FROG-Fehlers von 1,5 und 10

Quelle: Eigene Darstellung

4.5.3 Vorhersage bei unüberwachtem Trainieren mit FROG-Fehler

Um das unüberwachte Trainieren des neuronalen Netzes zu bewerten, soll zunächst ein nicht vortrainiertes DenseNet trainiert werden. Ein resultierender Puls ist in Abbildung A.1 im Anhang gezeigt. Hier ist ein sehr großer Fehler über den gesamten Puls zu erkennen. Ein großes Problem ist außerdem, dass hohe Amplituden nicht nur in der Mitte des Pulses auftreten und der Puls somit seine charakteristische Form verliert. Als alleiniger Hyperparameter in der Verlustfunktion eignet sich der FROG-Fehler folglich nicht.

Anschließend wird überprüft, ob das Training mit ausschließlich dem FROG-Fehler auf einem vortrainierten Netz zu einer Verbesserung der Vorhersage führt. Hierfür wird ein deutlich kleinerer Datensatz von 1000 Spektrogrammen genutzt. Ein resultierender Beispieldatensatz ist in Abbildung A.2 im Anhang abgebildet. Es ist erkennbar, dass der Fehler durch das unüberwachte Lernen steigt. Werden mehr Daten verwendet, nimmt der Fehler weiter zu. Es lässt sich nicht abschließend bewerten, ob das unüberwachte Trainieren eines vortrainierten neuronalen Netzes auf experimentellen Daten zu einer besseren Anpassung der Voraussage führt. Wenn genügend experimentelle Daten vorliegen, sollten in der Zukunft weitere Tests hierzu durchgeführt werden.

4.6 Einfluss der Anzahl an Trainingsepochen auf die Vorhersage

Als letzter Hyperparameter soll der Einfluss der Anzahl an Trainingsepochen untersucht werden. Es ist zu erwarten, dass es bei einer Erhöhung der Trainingsepochen zunächst zu einer Verbesserung des Fehlers kommt. Nach einiger Zeit kann es jedoch zum Overfitting des Modells kommen, wodurch der Validierungs- und Testfehler wieder steigen könnte. Hierzu soll das Training auf dem vollen Datensatz für eine Epoche mit dem Training auf $\frac{1}{10}$ des Datensatzes für zehn Epochen und dem Training auf dem vollen Datensatz für zehn Epochen verglichen werden. Die hierzu verwendeten Hyperparameter sind in Tabelle A.7 abgebildet. Der Testfehler wird in Abbildung 4.12 gezeigt. Zunächst wird der Testfehler zwischen dem Training von $\frac{1}{10}$ des Datensatzes für zehn Epochen mit dem Training des gesamten Datensatzes für eine Epoche verglichen. Beide Trainingsdurchläufe haben die gleiche Anzahl an Trainingsschritten durchlaufen. Es lässt sich erkennen, dass der Fehler deutlich geringer ausfällt, wenn für jeden Trainingsschritt ein neuer Datenpunkt verwendet wird. Vergleicht man das Training auf dem gleichen Datensatz für eine bzw. zehn Epochen ist ersichtlich, dass es bei der aktuellen Datenmenge noch zu

keinem Overfitting kommt, wenn mehr als eine Epoche lang trainiert wird. Hieraus lässt sich folgern, dass der Umfang des Datensatzes in Zukunft noch deutlich erhöht werden kann. Dadurch sollte es zu einer deutlichen Verbesserung des Fehlers kommen.

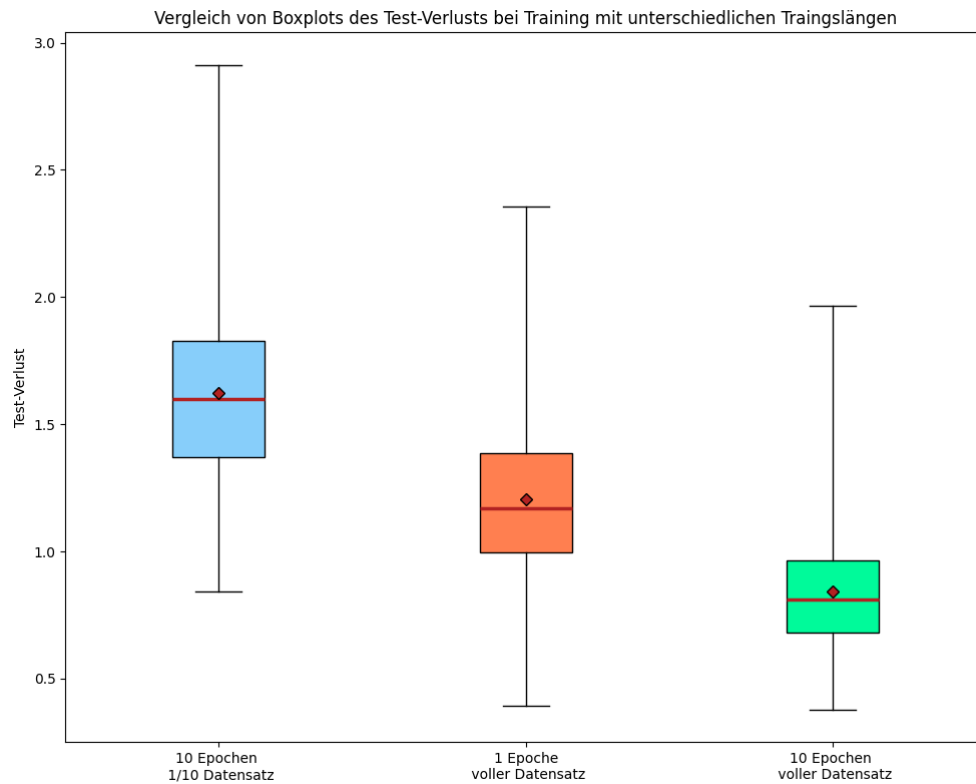


Abbildung 4.12: Boxplots des Testfehlers bei unterschiedlicher Trainingslänge
Quelle: Eigene Darstellung

4.7 Vergleich mit der Software von Swamp-Optics

In der Software von Swamp-Optics muss jeder Puls manuell rekonstruiert werden. Auch muss vom Nutzer entschieden werden, wann die Vorhersage des Pulses dessen Ansprüchen genügt. Zur Untersuchung des Vorhersagefehlers wurden 100 Pulse mithilfe der Swamp-Software rekonstruiert. Das arithmetische Mittel des Fehlers liegt bei $5,221 \cdot 10^{-8}$, der größte Fehler bei etwa $5,528 \cdot 10^{-6}$ und der niedrigste bei $2,801 \cdot 10^{-51}$. Die große Spanne des Fehlers lässt sich dadurch erklären, dass bei jedem Puls manuell entschieden werden muss, wann der Durchlauf der Software beendet wird. Im Allgemeinen liegt der erreichte Fehler deutlich unter den bisher mit dem neuronalen Netz erzielten Fehlern.

In Abbildung 4.13 ist der Boxplot des Testfehlers des bisher besten neuronalen Netzes dargestellt. Die verwendeten Hyperparameter sind Tabelle A.8 im Anhang zu entnehmen. Wie in Abbildung 4.14 zu erkennen ist, lässt sich anhand der aktuellen Vorhersage bereits ein ungefährender Verlauf des Signals erkennen. Auch die Steigung der Phase wird im relevanten Bereich gut vorhergesagt, hier gibt es neben einem kleinen Fehler vor allem einen absoluten Phasensprung. Da dieser eine triviale Ambiguität ist, lässt er sich vernachlässigen. Aufgrund des noch deutlich zu hohen Fehlers stellt das erstellte Modell jedoch noch keinen Ersatz für die Software von Swamp-Optics dar.

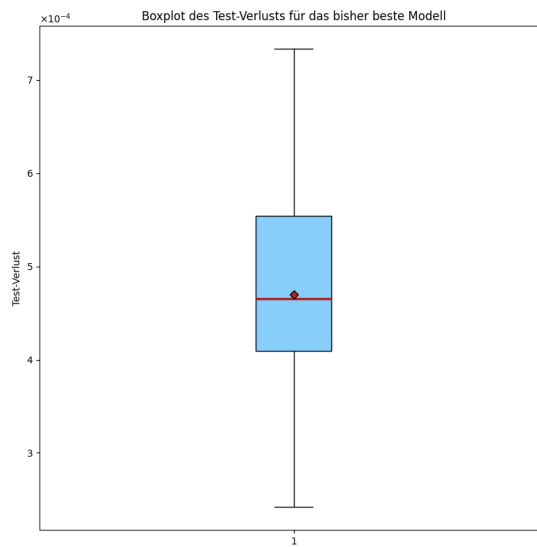


Abbildung 4.13: Boxplot des Testfehlers bei Nutzung des bisher besten Modells
Quelle: Eigene Darstellung

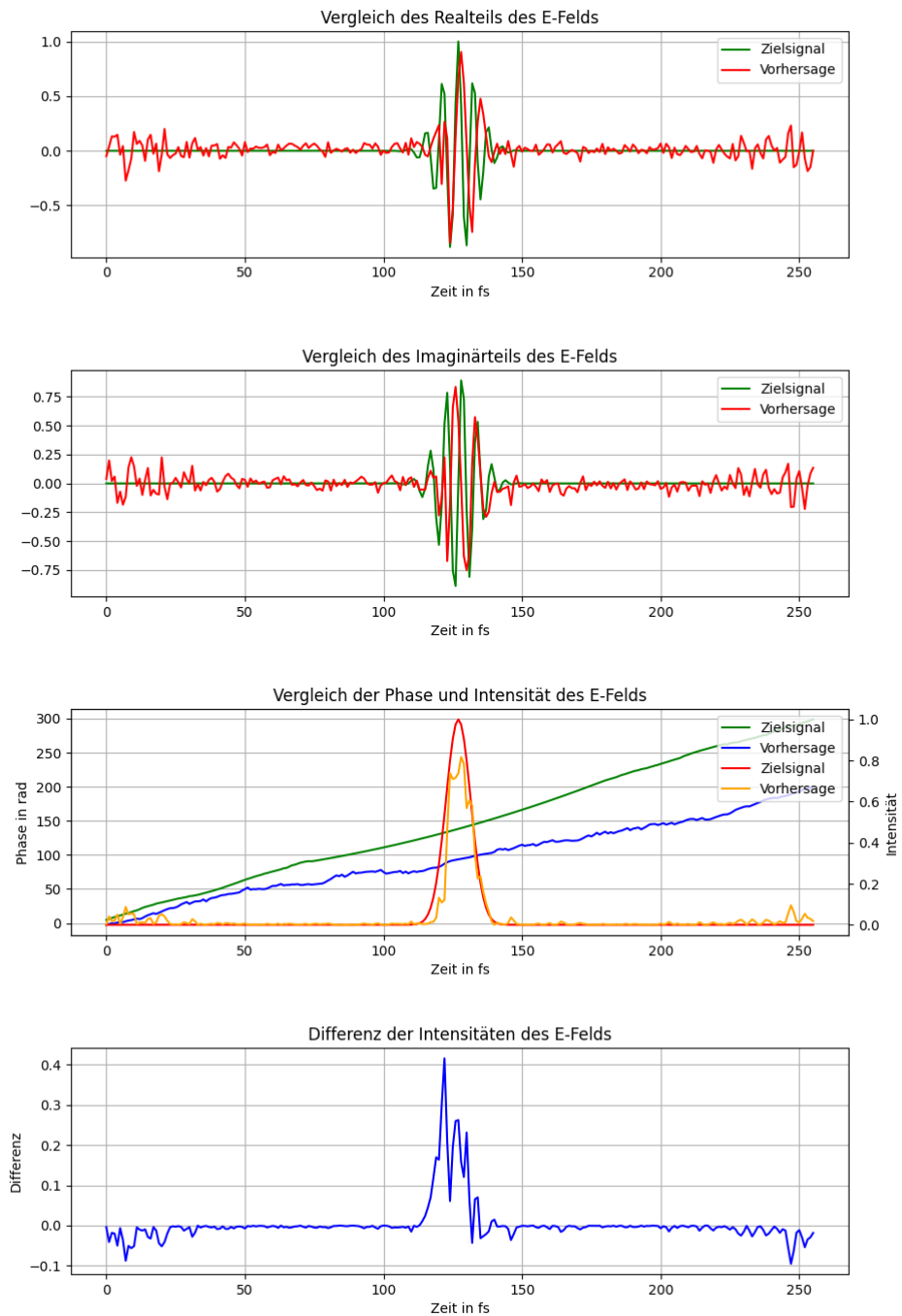


Abbildung 4.14: Vergleich eines Zielsignals mit einer Voraussage bei Nutzung des bisher besten Modells

Quelle: Eigene Darstellung

5 Ausblick

Ziel der vorliegenden Untersuchung ist die möglichst fehlerfreie Rekonstruktion des Zeitprofils eines ultrakurz Laserpulses aus dessen FROG-Trace. Sie ist hierbei noch zu keinem zufriedenstellenden Ergebnis gekommen. Damit die Voraussagen des Modells mit denen aus üblichen Softwarelösungen konkurrieren kann, muss es noch weiter optimiert werden. Im folgenden sollen einige mögliche Ansätze hierzu vorgestellt werden.

Um die Ergebnisse der Voraussage in Zukunft weiter zu verbessern, gibt es einige mögliche Ansätze. So sollte der Umfang und die Variabilität des Trainingsdatensatzes weiter erhöht werden. Wie in Kapitel 4.6 erklärt, könnte der Umfang erhöht werden, bis es zu einem Overfitting des Modells kommt. Da die Verlustfunktion über viele Hyperparameter verfügt, ist die Suche nach den optimalen Werten sehr zeitaufwändig. Hier könnte ein Grid-Search Ansatz verfolgt werden, welcher die Ergebnisse der Voraussagen stark verbessern könnte. Zum Einsparen von Rechenzeit könnte es zudem sinnvoll sein, die Wellenlängenchse der Spektrogramme vor dem Training in eine Frequenzachse zu wandeln. Hierdurch könnte die Umwandlung von Frequenz- zu Wellenlängenchse in der Verlustfunktion vermieden werden. Die Umwandlung sollte hierzu möglichst in das in Kapitel 3.2.3 beschriebene Vorverarbeitungsskript aufgenommen werden. Das Skript spielt außerdem eine große Rolle darin, das neuronale Netz auf die Verwendung experimentell erstellter Daten vorzubereiten. Hierzu sollten die Verlustfunktion und das Vorverarbeitungsskript so aufeinander angepasst werden, dass das Erstellen von Spektrogrammen aus dem vorhergesagten Realteil zuverlässig funktioniert. Ein weiterer Ansatz zur Verbesserung der Voraussagen ist es die Netzwerkarchitektur zu variieren, um so ein für die Aufgabe möglichst ideales Modell zu erzeugen. In der vorliegenden Untersuchung wurde lediglich die DenseNet Architektur betrachtet.

Um aus den jetzt vorliegenden Modellen bereits bessere Voraussagen zu erzeugen, könnten sie als erste Schätzung in einem traditionellen Ansatz zur Rückgewinnung des Signals genutzt werden. Dazu könnte ein Ansatz, wie er in Kapitel 2.4 beschrieben ist, sinnvoll sein. Dies könnte zu einer Verkürzung der Vorhersagezeit solcher Software dienen.

Als Fernziel kann das hier vorgestellte Modell im Rahmen eines Reinforcement-Learning ansatzes genutzt werden. Dieses könnte einen Pulse-Shaper so steuern, dass er aus einem gechirpten Puls ein fourierlimitierten erzeugt.

Literaturverzeichnis

- [1] BROWNLEE, Jason: *Weight Initialization for Deep Learning Neural Networks*. – URL <https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/>. – Zugriffsdatum: 2024-13-17
- [2] GEIB, Nils C. ; ZILK, Matthias ; PERTSCH, Thomas ; EILENBERGER, Falk: Common pulse retrieval algorithm: a fast and universal method to retrieve ultrashort pulses. In: *Optica* 6 (2019), Apr, Nr. 4, S. 495–505. – URL <https://opg.optica.org/optica/abstract.cfm?URI=optica-6-4-495>
- [3] GLOROT, Xavier ; BENGIO, Yoshua: Understanding the difficulty of training deep feedforward neural networks. In: TEH, Yee W. (Hrsg.) ; TITTERINGTON, Mike (Hrsg.): *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* Bd. 9. Chia Laguna Resort, Sardinia, Italy : PMLR, 13–15 May 2010, S. 249–256. – URL <https://proceedings.mlr.press/v9/glorot10a.html>
- [4] GOODFELLOW, Ian ; BENGIO, Yoshua ; COURVILLE, Aaron: *Deep Learning*. MIT Press, 2016. – <http://www.deeplearningbook.org>
- [5] HAO, Yufeng ; TANG, Zhengdao ; TIAN, Yixiao ; ZHANG, Yijie ; ZHOU, Zheng: *Cornell University Computational Optimization Open Textbook - AdamW*. – URL <https://optimization.cbe.cornell.edu/index.php?title=AdamW>. – Zugriffsdatum: 2025-01-14
- [6] HE, Kaiming ; ZHANG, Xiangyu ; REN, Shaoqing ; SUN, Jian: *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. – URL <https://arxiv.org/abs/1502.01852>
- [7] HUANG, Gao ; LIU, Zhuang ; MAATEN, Laurens van der ; WEINBERGER, Kilian Q.: *Densely Connected Convolutional Networks*. 2018. – URL <https://arxiv.org/abs/1608.06993>
- [8] JANSEN, Paul: *TIOBE Index for January 2025*. – URL <https://www.tiobe.com/tiobe-index/>. – Zugriffsdatum: 2025-01-22

- [9] MATHURANATHAN: *Understanding Analytic Signal and Hilbert Transform*. – URL <https://www.gaussianwaves.com/2017/04/analytic-signal-hilbert-transform-and-ft/>. – Zugriffsdatum: 2024-11-21
- [10] NUMPY DEVELOPERS: *What is NumPy?*. – URL <https://numpy.org/doc/stable/user/whatisnumpy.html>. – Zugriffsdatum: 2025-01-14
- [11] PANDAS DEVELOPERS: *About pandas*. – URL <https://pandas.pydata.org/about/>. – Zugriffsdatum: 2025-01-14
- [12] PICKERING, James D.: *Ultrafast Lasers and Optics for Experimentalists*. IOP Publishing Ltd., 2021. – ISBN 9780750336574
- [13] PYTORCH CONTRIBUTORS: *PyTorch documentation*. – URL <https://pytorch.org/docs/stable/index.html>. – Zugriffsdatum: 2025-01-14
- [14] PYTORCH CONTRIBUTORS: *Pytorch Documentation - OneCycleLR*. – URL https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.OneCycleLR.html. – Zugriffsdatum: 2024-12-23
- [15] PYTORCH DEVELOPERS: *PyTorch Dokumentation - Densenet*. – URL https://pytorch.org/hub/pytorch_vision_densenet/. – Zugriffsdatum: 2024-11-13
- [16] SCIPY DEVELOPERS: *SciPy*. – URL <https://scipy.org/>. – Zugriffsdatum: 2025-01-16
- [17] SMITH, Leslie N. ; TOPIN, Nicholay: *Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates*. 2018. – URL <https://arxiv.org/abs/1708.07120>
- [18] TANKSALE, Nachiket: *Finding Good Learning Rate and The One Cycle Policy*. – URL <https://towardsdatascience.com/finding-good-learning-rate-and-the-one-cycle-policy-7159fe1db5d6>. – Zugriffsdatum: 2024-12-23
- [19] THE MATPLOTLIB DEVELOPMENT TEAM: *Matplotlib: Visualization with Python*. – URL <https://matplotlib.org/>. – Zugriffsdatum: 2025-01-15
- [20] TREBINO, Rick: *Frequency-resolved optical gating: the measurement of ultrashort laser pulses*. Kluwer Academic Publishers, 2000. – ISBN 1402070667
- [21] TÓTH, István ; GHERMAN, Ana Maria M. ; KOVÁCS, Katalin ; CHO, Wosik ; YUN, Hyeok ; TOŞA, Valer: *Reconstruction of Femtosecond Laser Pulses from FROG*

- Traces by Convolutional Neural Networks. In: *Photonics* 10 (2023), Nr. 11. – URL <https://www.mdpi.com/2304-6732/10/11/1195>. – ISSN 2304-6732
- [22] WOLLENHAUPT, Matthias ; ASSION, Andreas ; BAUMERT, Thomas: *Femtosecond Laser Pulses: Linear Properties, Manipulation, Generation and Measurement*. Springer Science+Business Media, LLC New York, 2007. – ISBN 978-0-387-95579-7
- [23] ZAHAVY, Tom ; DIKOPOLTSEV, Alex ; MOSS, Daniel ; HAHAM, Gil I. ; COHEN, Oren ; MANNOR, Shie ; SEGEV, Mordechai: Deep learning reconstruction of ultrashort pulses. In: *Optica* 5 (2018), May, Nr. 5, S. 666–673. – URL <https://opg.optica.org/optica/abstract.cfm?URI=optica-5-5-666>

A Anhang

A.1 Listings

```
1 def zeroCrossings(x, y, tolerance=1e-12):
2     N = len(x)
3     x_zero = []
4     # check if y and x have the same length
5     if len(y) != N:
6         raise ValueError("arguments x and y need to have the same length")
7
8     # check if first element of y is close to zero (smaller than +-tolerance)
9     if abs(y[0]) < tolerance:
10        # add value to zero crossings
11        x_zero.append(x[0])
12
13    for n in range(1, N):
14        # check if element is close to zero (smaller than +-tolerance)
15        if abs(y[n]) < tolerance:
16            # add value to zero crossings
17            x_zero.append(x[n])
18            continue # go to next index n
19
20        # check for changing sign between y[n] and y[n-1]
21        # this is the case when y[n] * y[n-1] is negative and y[n-1] isn't inside the
22        # ↪ tolerance considered zero
23        if y[n] * y[n-1] < 0 and abs(y[n-1]) > tolerance:
24            # calculate the difference between x[n] and x[n-1]
25            delta_x = x[n] - x[n-1]
26            if not (delta_x > 0):
```

```
26         raise ValueError("Difference between x-values needs to be > 0")
27
28     # calculate the difference between y[n] and y[n-1]
29     delta_y = y[n] - y[n-1]
30
31     # calculate the slope
32     slope = delta_y / delta_x
33     assert abs(slope) > 0 # check that slope is not 0
34
35     # calculate intercepts
36     intercept_1 = y[n-1] - slope * x[n-1]
37     intercept_2 = y[n] - slope * x[n]
38     # mean intercept
39     mean_intercept = (intercept_1 + intercept_2) / 2
40
41     # calculate zero
42     zero = -mean_intercept / slope
43
44     x_zero.append(zero)
45
46     return x_zero
```

Listing 23: Funktion zum Bestimmen von Nulldurchgängen eines Signals
Quelle: Eigene Darstellung, Auszug aus `./modules/preprocessing.py`

A.2 Abbildungen

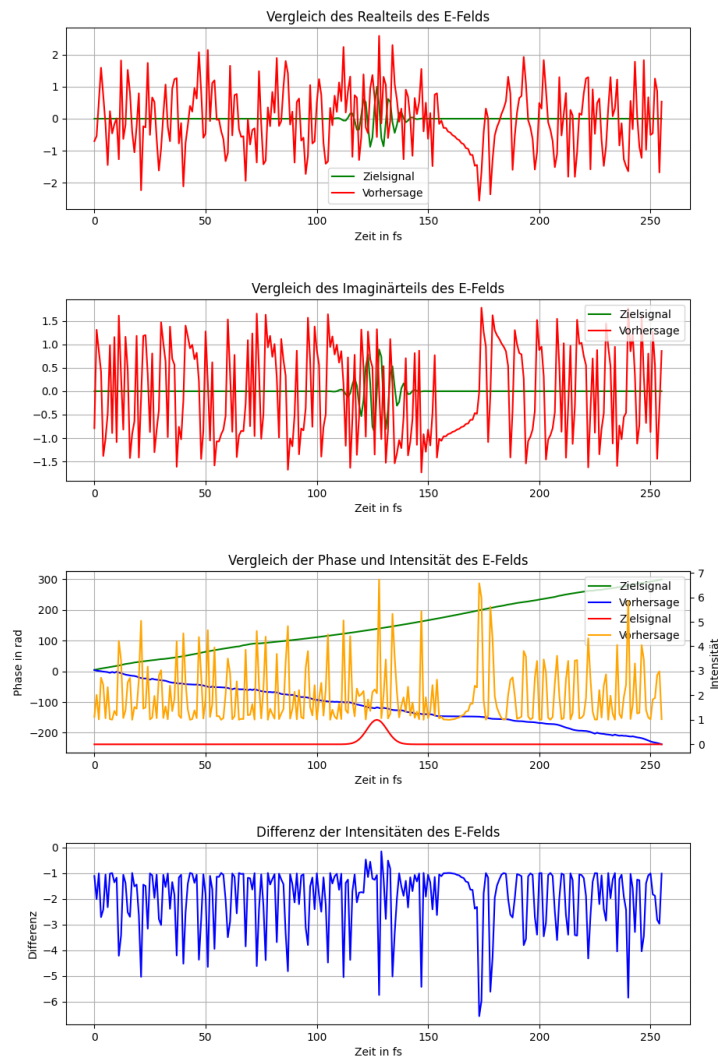


Abbildung A.1: Vergleich eines Zielsignals mit einer Vorraussage bei unüberwachtem Lernen ohne vortrainiertes Netz
Quelle: Eigene Darstellung

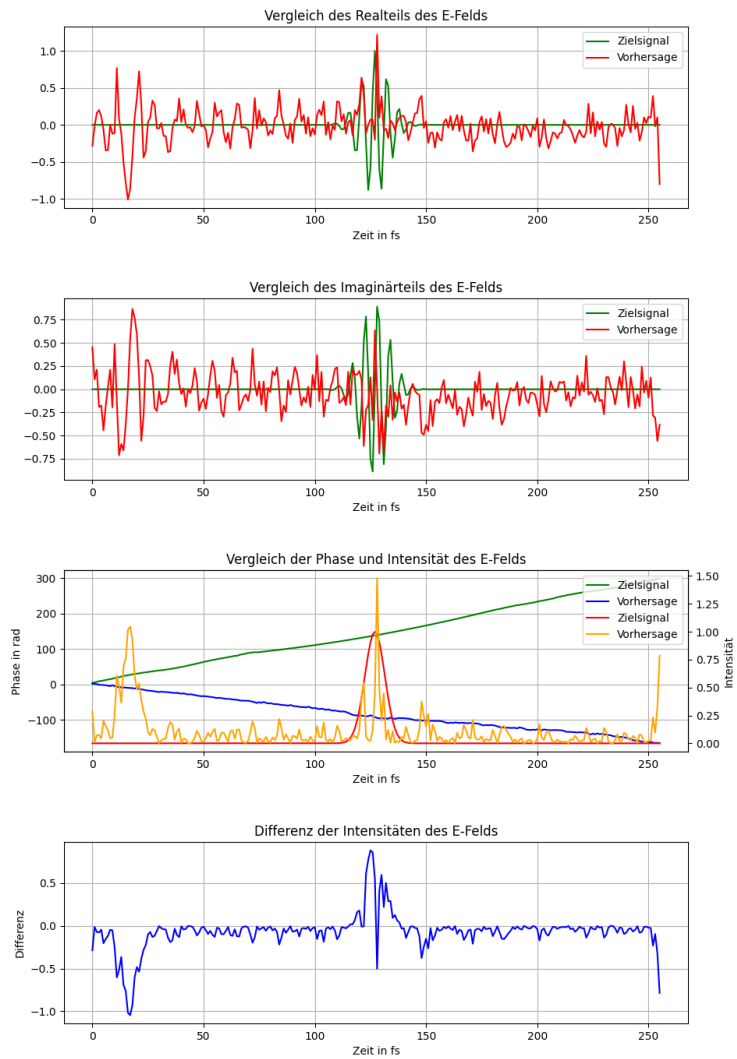


Abbildung A.2: Beispielpuls nach unüberwachtem Training auf einem vortrainierten Modell
 Quelle: Eigene Darstellung

A.3 Hyperparameter

Tabelle A.1: Hyperparameter des Trainings zum Vergleich des Effekts des Rauschens
 Quelle: Eigene Darstellung

	Rauschen
Anzahl der Epochen	1
Batch-Größe	10
Lernrate α	$2, 13 \cdot 10^{-6}$
Maximale Lernrate	$2, 13 \cdot 10^{-5}$
Gewichtszurücknahme λ_w	$1 \cdot 10^{-5}$
Maximaler TBD_{RMS} -Wert	0,5514
Verlustfunktion	Überwacht
Nutzung des Zielsignals?	Ja
Minimale Amplitude des Pulses	0,001
Bestrafungsfaktor für Abtastwerte außerhalb des Pulses	10,0
Gewicht MSE des Realteils	10,0
Gewicht MSE des Imaginärteils	0,0
Gewicht MSE der Intensität	10,0
Gewicht MSE der Phase	0,0
Gewicht MSE des FROG-Fehlers	0,0

Tabelle A.2: Hyperparameter des Trainings zum Vergleich von Lernrattenschedulern
Quelle: Eigene Darstellung

	Konstante Lernrate	Linear ansteigende Lernrate	1Cycle-Lernrattenschedulern
Anzahl der Epochen	1	1	1
Batch-Größe	10	10	10
Lernrate α	$2, 13 \cdot 10^{-6}$	$2, 13 \cdot 10^{-6}$	$2, 13 \cdot 10^{-6}$
Maximale Lernrate (bei Lernrattenschedulern)		$2, 13 \cdot 10^{-5}$	$2, 13 \cdot 10^{-5}$
Gewichtszurücknahme λ_w	$1 \cdot 10^{-5}$	$1 \cdot 10^{-5}$	$1 \cdot 10^{-5}$
Maximaler TBD_{RMS} -Wert	20	20	20
Verlustfunktion	Überwacht	Überwacht	Überwacht
Nutzung des Zielsignals?	Ja	Ja	Ja
Minimale Amplitude des Pulses	0,001	0,001	0,001
Bestrafungsfaktor für Abtastwerte außerhalb des Pulses	10.0	10.0	10.0
Gewicht MSE des Realteils	10.0	10.0	10.0
Gewicht MSE des Imaginärteils	0.0	0.0	0.0
Gewicht MSE der Intensität	10.0	10.0	10.0
Gewicht MSE der Phase	0.0	0.0	0.0
Gewicht MSE des FROG-Fehlers	0.0	0.0	0.0

Tabelle A.3: Hyperparameter des Trainings zum Vergleich des Effekts des TBD-Werts
Quelle: Eigene Darstellung

	Alle Datenpunkte		90%-Perzentil der TBD_{RMS} -Werte	50%-Perzentil der TBD_{RMS} -Werte
Anzahl der Epochen	1	10	1	1
Batch-Größe	10	10	10	10
Lernrate α	$2, 13 \cdot 10^{-6}$	$2, 13 \cdot 10^{-6}$	$2, 13 \cdot 10^{-6}$	$2, 13 \cdot 10^{-6}$
Maximale Lernrate (bei Lernratenscheduler)	$2, 13 \cdot 10^{-5}$	$2, 13 \cdot 10^{-5}$	$2, 13 \cdot 10^{-5}$	$2, 13 \cdot 10^{-5}$
Gewichtszurücknahme λ_w	$1 \cdot 10^{-5}$	$1 \cdot 10^{-5}$	$1 \cdot 10^{-5}$	$1 \cdot 10^{-5}$
Maximaler TBD_{RMS} -Wert	20	20	1,063	0,5514
Verlustfunktion	Überwacht	Überwacht	Überwacht	Überwacht
Nutzung des Zielsignals?	Ja	Ja	Ja	Ja
Minimale Amplitude des Pulses	0,001	0,001	0,001	0,001
Bestrafungsfaktor für Abtastwerte außerhalb des Pulses	10.0	10.0	10.0	10.0
Gewicht MSE des Realteils	10.0	10.0	10.0	10.0
Gewicht MSE des Imaginärteils	0.0	0.0	0.0	0.0
Gewicht MSE der Intensität	10.0	10.0	10.0	10.0
Gewicht MSE der Phase	0.0	0.0	0.0	0.0
Gewicht MSE des FROG-Fehlers	0.0	0.0	0.0	0.0

Tabelle A.4: Hyperparameter zum Ermitteln eines Testfehlers, der unabhängig von den Hyperparametern des Trainings ist
 Quelle: Eigene Darstellung

	Ermittlung des Testfehlers
Anzahl der Epochen	
Batch-Größe	
Lernrate α	
Maximale Lernrate	
Gewichtszurücknahme λ_w	
Maximaler TBD_{RMS}-Wert	Überwacht
Verlustfunktion	Ja
Nutzung des Zielsignals?	0,001
Minimale Amplitude des Pulses	
Bestrafungsfaktor für Abtastwerte außerhalb des Pulses	0.01
Gewicht MSE des Realteils	1.0
Gewicht MSE des Imaginärteils	0.0
Gewicht MSE der Intensität	0.0
Gewicht MSE der Phase	0.0
Gewicht MSE des FROG-Fehlers	0.0

Tabelle A.5: Hyperparameter des Trainings zum Vergleich ihrer Auswirkung auf das überwachte Lernen
Quelle: Eigene Darstellung

	Gleiches Gewicht für Real- und Imaginärteil	Gleiches Gewicht für Realteil und Intensität	Berücksichtigung der Phase
Anzahl der Epochen	1	1	1
Batch-Größe	10	10	10
Lernrate α	$2,13 \cdot 10^{-6}$	$2,13 \cdot 10^{-6}$	$2,13 \cdot 10^{-6}$
Maximale Lernrate	$2,13 \cdot 10^{-5}$	$2,13 \cdot 10^{-5}$	$2,13 \cdot 10^{-5}$
Gewichtszurücknahme λ_w	$1 \cdot 10^{-5}$	$1 \cdot 10^{-5}$	$1 \cdot 10^{-5}$
Maximaler TBD_{RMS} -Wert	20	20	20
Verlustfunktion	Überwacht	Überwacht	Überwacht
Nutzung des Zielsignals?	Ja	Ja	Ja
Minimale Amplitude des Pulses	0,001	0,001	0,001
Bestrafungsfaktor für Abtastwerte außerhalb des Pulses	10,0	10,0	5,0
Gewicht MSE des Realteils	10,0	10,0	5,0
Gewicht MSE des Imaginärteils	10,0	0,0	0,0
Gewicht MSE der Intensität	0,0	10,0	10,0
Gewicht MSE der Phase	0,0	0,0	5,0
Gewicht MSE des FROG-Fehlers	0,0	0,0	0,0

Tabelle A.6: Hyperparameter des Trainings zum Vergleich ihrer Auswirkung des FROG-Fehlers beim überwachten Lernen

	Gewicht des		Gewicht des	
	FROG-Fehlers = 1	FROG-Fehlers = 5	FROG-Fehlers = 1	FROG-Fehlers = 10
Anzahl der Epochen	1	1	1	1
Batch-Größe	10	10	10	10
Lernrate α	$2,13 \cdot 10^{-6}$	$2,13 \cdot 10^{-6}$	$2,13 \cdot 10^{-6}$	$2,13 \cdot 10^{-6}$
Maximale Lernrate	$2,13 \cdot 10^{-5}$	$2,13 \cdot 10^{-5}$	$2,13 \cdot 10^{-5}$	$2,13 \cdot 10^{-5}$
Gewichtszurücknahme λ_w	$1 \cdot 10^{-5}$	$1 \cdot 10^{-5}$	$1 \cdot 10^{-5}$	$1 \cdot 10^{-5}$
Maximaler TBD_{RMS} -Wert	0,5514	0,5514	0,5514	0,5514
Verlustfunktion	Überwacht	Überwacht	Überwacht	Überwacht
Nutzung des Zielsignals?	Ja	Ja	Ja	Ja
Minimale Amplitude des Pulses	0,001	0,001	0,001	0,001
Bestrafungsfaktor für Abtastwerte außerhalb des Pulses	10,0	10,0	10,0	10,0
Gewicht MSE des Realteils	10,0	10,0	10,0	10,0
Gewicht MSE des Imaginärteils	0,0	0,0	0,0	0,0
Gewicht MSE der Intensität	10,0	10,0	10,0	10,0
Gewicht MSE der Phase	0,0	0,0	0,0	0,0
Gewicht MSE des FROG-Fehlers	1,0	5,0	5,0	10,0

Tabelle A.7: Hyperparameter des Trainings zum Vergleich der Auswirkung der Epochenanzahl

	10 Epochen	1 Epoche	10 Epochen
Anzahl der Epochen	10	1	10
Batch-Größe	10	10	10
Lernrate α	$2,13 \cdot 10^{-6}$	$2,13 \cdot 10^{-6}$	$2,13 \cdot 10^{-6}$
Maximale Lernrate	$2,13 \cdot 10^{-5}$	$2,13 \cdot 10^{-5}$	$2,13 \cdot 10^{-5}$
Gewichtszurücknahme λ_w	$1 \cdot 10^{-5}$	$1 \cdot 10^{-5}$	$1 \cdot 10^{-5}$
Maximaler TBD_{RMS} -Wert	20	20	20
Verlustfunktion	Überwacht	Überwacht	Überwacht
Nutzung des Zielsignals?	Ja	Ja	Ja
Minimale Amplitude des Pulses	0,001	0,001	0,001
Bestrafungsfaktor für Abtastwerte außerhalb des Pulses	10,0	10,0	10,0
Gewicht MSE des Realteils	10,0	10,0	10,0
Gewicht MSE des Imaginärteils	0,0	0,0	0,0
Gewicht MSE der Intensität	10,0	10,0	10,0
Gewicht MSE der Phase	0,0	0,0	0,0
Gewicht MSE des FROG-Fehlers	0,0	0,0	0,0

Tabelle A.8: Hyperparameter des bisher besten Modells

	Bester bisheriger Puls
Anzahl der Epochen	3
Batch-Größe	10
Lernrate α	$2,13 \cdot 10^{-6}$
Maximale Lernrate	$2,13 \cdot 10^{-5}$
Gewichtszurücknahme λ_w	$1 \cdot 10^{-5}$
Maximaler TBD_{RMS} -Wert	0,5514
Verlustfunktion	Überwacht
Nutzung des Zielsignals?	Ja
Minimale Amplitude des Pulses	0,001
Bestrafungsfaktor für Abtastwerte außerhalb des Pulses	10,0
Gewicht MSE des Realteils	10,0
Gewicht MSE des Imaginärteils	0,0
Gewicht MSE der Intensität	10,0
Gewicht MSE der Phase	0,0
Gewicht MSE des FROG-Fehlers	5,0

A.4 Verwendete Hilfsmittel

In der Tabelle A.9 sind die im Rahmen der Bearbeitung des Themas der Masterarbeit verwendeten Werkzeuge und Hilfsmittel aufgelistet.

Tabelle A.9: Verwendete Hilfsmittel und Werkzeuge

Tool	Verwendung
L ^A T _E X	Textsatz- und Layout-Werkzeug verwendet zur Erstellung dieses Dokuments
TikZ	Grafik-Werkzeug verwendet zur Erstellung von Abbildungen
Matplotlib	Python-Bibliothek verwendet zur Erstellung von Abbildungen
ChatGPT	Chatbot auf Basis künstlicher Intelligenz verwendet zur Rechtschreibprüfung und Quellenrecherche
DeepL Write	Schreibassistent auf Basis künstlicher Intelligenz verwendet zur Rechtschreibprüfung

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen 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