FernUniversität in Hagen
Fakultät für Mathematik und Informatik

# Bachelor-Arbeit

**Klassifikation von klassischer Musik mit Hilfe von neuronalen Netzen**

Silke Schäfer
Matrikelnummer 8718040

Wintersemester 2018/19

ausgeführt am
Lehrgebiet Theoretische Informatik
Leitung Prof. Dr. André Schulz

**Kurzfassung**

Thema dieser Bachelor-Thesis ist die Klassifikation von klassischer Musik mit Hilfe von neuronalen Netzen. Aus Gründen der Vergleichbarkeit werden ausschließlich Klavierstücke verwendet. Die Kategorien entsprechen den Epochen oder den Komponisten der Stücke. Alle Dateien liegen im *MIDI*-Format vor und müssen zunächst zur Eingabe in ein neuronales Netz aufbereitet werden. Da es sich um sequenzielle Daten handelt, bietet sich zur Klassifikation ein *LSTM*-Netz (*Long Short-Term Memory*) an.

Nach einer kurzen Einführung in die Merkmale klassischer Musik verschiedener Epochen und den Aufbau von *MIDI*-Dateien werden die wichtigsten Konzepte des maschinellen Lernens vorgestellt. Die *lineare* und *logistische Regression* sowie die *neuronalen Netze* finden besondere Beachtung, da die hier vorgestellten Methoden in die Implementierung des Netzes einfließen.

Die Beschreibung der Implementierung beinhaltet zunächst die *Datenaufbereitung*. Es werden drei Möglichkeiten vorgestellt, die Stücke zu quantisieren, um jeweils gleich lange Merkmalsvektoren zu erhalten. Der zweite Teil der Implementierung betrifft die *Erstellung* und *Konfiguration* des neuronalen Netzes. Er gliedert sich in drei Phasen. In Phase eins werden drei unterschiedliche Netztypen jeweils in Kombination mit den drei Merkmalscodierungen getestet. Das *vollverbundene Feedforward-Netz* schneidet hier am schlechtesten ab, das *LSTM*-Netz am besten. Bei fünf Kategorien wird eine Korrektklassifizierungsrate von mehr als 95% erzielt. Der dritte getestete Netztyp schaltet dem *LSTM*-Netz eine oder mehrere *konvolutionale* Schichten voran, um die Merkmale zunächst zu größeren Einheiten zusammenzufassen. Dies verkürzt die Laufzeit um mehr als die Hälfte, die Leistungen liegen jedoch knapp unter denen eines reinen *LSTM*-Netzes. In Phase zwei wird das erfolgreichste Netz durch Optimierung verschiedener Parameter feinabgestimmt. Das fertige Netz wird schließlich in Phase drei vor einige schwierigere Aufgaben gestellt. Im Falle der Erkennung dreier Komponisten des Barockzeitalters liegt die Korrektklassifizierung zumindest noch oberhalb der 80%-Marke. Das Training auf einer Datenmenge, die Stücke von mehr als 20 Komponisten enthält, liefert einen Klassifizierer, der bei sechs Kategorien 79.4% der Stücke einer Testmenge korrekt ihrer Epoche zuordnet. Eine weitere Datenmenge dient zur Klassifikation nach Komponist bei 15 Kategorien. Hier werden 61.3% der Stücke ihrem tatsächlichen Komponisten zugeordnet. Dies ist ein vergleichsweise guter Wert, denn die Menge enthält deutlich weniger Beispiele pro Kategorie als die übrigen. Es werden mögliche Ursachen für falsche Klassifizierungen aufgezeigt und anhand der Ergebnisse untersucht, welche Epochen bzw. Komponisten sich so ähnlich sind, dass sie sich nur schwer voneinander unterscheiden lassen. Mit Hilfe der gewonnenen Erkenntnisse wird schließlich eine letzte Datenmenge erstellt, deren fünf Kategorien Stücke jeweils mehrerer Komponisten enthalten. Das Training auf dieser Menge liefert einen Klassifizierer, der fast 93% der Stücke einer Testmenge korrekt einordnet. Zuletzt werden Möglichkeiten aufgezeigt, wie sich die erzielten Ergebnisse mit mehr Speicher- und Rechenkapazität noch steigern lassen könnten.

# Inhaltsverzeichnis

# 1. Einleitung

## 1.1. Kurze Einführung in maschinelles Lernen und künstliche neuronale Netze

Die Idee des *maschinellen Lernens* geht auf einen zukunftsweisenden Artikel von Alan Turing aus dem Jahr 1950 zurück [26]. Es geht dort um die Frage, ob es möglich sei, einem Computer das „Denken“ beizubringen, in dem Sinne, dass er selbstständig lernt, wie eine bestimmte Aufgabe zu lösen ist. Bis zu diesem Zeitpunkt war man davon ausgegangen, dass eine Maschine niemals etwas Neues erschaffen könne. Turing war anderer Ansicht. Damit legte er den Grundstein für ein völlig neues Programmierparadigma. Es wurden selbstlernende Algorithmen entwickelt, die dazu dienen sollten, aus „*gekennzeichneten*“ Daten neues, regelhaftes Wissen zu extrahieren. Das *Kennzeichen* (oder auch *Label*) bezeichnet das korrekte oder beobachtete Ergebnis eines Datensatzes. Dies kann eine Kategorie oder ein numerischer Wert sein. Mit Hilfe des erlernten Wissens können Vorhersagen für neue Datensätze getroffen werden, deren zugehöriges Label unbekannt ist. Dieses Vorgehen beschreibt eines der drei großen Gebiete des maschinellen Lernens, nämlich das *überwachte Lernen*. Daneben gibt es das *unüberwachte* und das *verstärkende Lernen*, auf die in der vorliegenden Arbeit nur am Rande eingegangen wird.

Das *überwachte Lernen* kann wiederum in zwei große Teilbereiche gegliedert werden, nämlich die *Regression* und die *Klassifikation*. In der *Regression* entstammen die Ausgabewerte einem kontinuierlichen Wertebereich, in der *Klassifikation* einem diskreten. Für beide Varianten gibt es zahlreiche Verfahren, von denen einige in Kapitel 3 vorgestellt werden. *Neuronale Netze* eignen sich prinzipiell sowohl zur Regression als auch zur Klassifikation. Im Rahmen dieser Arbeit wird das Augenmerk auf der Klassifikation liegen.

Ein *künstliches neuronales Netz* ist zunächst allgemein ein Netzwerk aus künstlichen Neuronen. Biologisches Vorbild ist die Vernetzung natürlicher Neuronen im Gehirn. Ein künstliches Neuron besitzt eine Übertragungsfunktion, die eine gewichtete Summe der Eingangssignale bildet, und eine im Falle der Klassifikation nichtlineare Aktivierungsfunktion, die aus dieser Summe einen Ausgabewert berechnet. Eine Schwellenwertfunktion, die nur den Wert 0 oder 1 annehmen kann, würde dem natürlichen Vorbild am nächsten kommen, ist aber eher selten in Gebrauch. Häufiger kommen sogenannte *Sigmoid-Funktionen* zum Einsatz, die die Ausgabe in einen Wertebereich zwischen 0 und 1 oder zwischen $-1$ und $+1$ zwängen. Solche künstlichen Neuronen werden nun in mehreren Schichten organisiert und miteinander vernetzt. In der Regel gibt es eine Eingabe- und eine Ausgabeschicht und mindestens eine *verdeckte* Schicht. Bekommt jedes Neuron einer Schicht Informationen von jedem Neuron der Vorgängerschicht und leitet seine Ausgabe

an jedes Neuron der nachfolgenden Schicht weiter, so spricht man von einem *vollverbundenen Feedforward-Netz*. Daneben gibt es viele andere Möglichkeiten der Vernetzung. In *rekurrenten Netzen* kann ein Neuron auch ausgehende Verbindungen zu seiner eigenen oder einer vorhergehenden Schicht besitzen. Dadurch wird es möglich, Informationen über einen kurzen Zeitraum zu speichern. Ein Spezialfall rekurrenter Netze ist das *LSTM*-Netz. Das Akronym steht für *Long Short-Term Memory* („*langes Kurzzeitgedächtnis*“) und drückt aus, dass Informationen hier auch über längere Zeiträume gespeichert werden können. Solche Netze sind für sequenzielle Daten wie Musikstücke oder Sprache prädestiniert und werden in dieser Arbeit eingehend untersucht.

## 1.2. Aufgabenstellung

Ziel der vorliegenden Thesis ist die Entwicklung eines neuronalen Netzes zur Klassifikation von klassischer Musik nach Epoche oder Komponist. Die Wahl fiel hier speziell auf klassische Klaviermusik, da eine relativ große Anzahl solcher Werke im *MIDI*-Format zum kostenlosen Download angeboten wird und die Stücke dadurch auch besser vergleichbar sind. Die Herausforderung besteht darin, die Daten zunächst zur Eingabe in ein neuronales Netz aufzubereiten, im Anschluss den am besten geeigneten Netztyp zu bestimmen und diesen zu konfigurieren. Das Netz soll nach dem Training einen Klassifizierer liefern, der auch auf bis zu diesem Zeitpunkt unbekannten Datensätzen eine möglichst hohe Korrektklassifizierungsrate erzielt.

## 1.3. Stand der Forschung

Die Idee, künstliche Neuronen miteinander zu vernetzen, geht auf Warren McCulloch und Walter Pitts zurück, die 1943 mit der nach ihnen benannten *McCulloch-Pitts-Zelle* das erste künstliche Neuron entwarfen [15]. Erste praktische Anwendungen wurden Ende der 1950er Jahre entwickelt, jedoch wurden die Forschungen bereits 10 Jahre später vorläufig wieder eingestellt, da aufgrund von Zweifeln an der Leistungsfähigkeit solcher Netze die Gelder gestrichen worden waren. In den 1970er Jahren wurden wichtige mathematische Modelle entwickelt und der *Backpropagation Algorithmus* erstmals auf neuronale Netze angewandt [28]. Große Erfolge in der Praxis feiern neuronale Netze jedoch erst seit etwa 2009, da zu dieser Zeit die Rechnerleistung ausreichte, um mit riesigen Datenmengen zu trainieren. *LSTM*-Netze wurden 1997 von Sepp Hochreiter und Jürgen Schmidhuber vorgestellt [12], aber erst seit 2015 kontinuierlich erforscht und weiterentwickelt. Google verwendet sie beispielsweise zur Spracherkennung [24]. Zur Anwendung eines *LSTM*-Netzes auf Musikdateien unterschiedlicher Formate gibt es bereits Veröffentlichungen. Diese betreffen jedoch im Wesentlichen die automatische *Improvisation* [5] und die *Komposition* von Stücken im Stile Johann Sebastian Bachs [14]. In Artikeln zur automatischen *Klassifikation* geht es in den meisten Fällen um Erkennung des *Genres* (z.B. Klassik, Jazz, Chanson, Pop, Volksmusik) mit unterschiedlichen Methoden des maschinellen Lernens [27][6]. Eine Veröffentlichung über die automatische Klassifikation klassischer Musik ist nicht bekannt.

## 1.4. Organisation der Arbeit

In Kapitel 2 werden einige wichtige Grundbegriffe zu drei großen Themenbereichen eingeführt. Zunächst betrifft dies die Kategorien des maschinellen Lernens. Anschließend geht es um die Epochen der klassischen Musik und die Frage, anhand welcher Kriterien menschliche Experten die Epoche oder den Komponisten eines klassischen Musikstückes erkennen. Da die Dateien im *MIDI*-Format vorliegen, wird zudem eine kurze Einführung in den Aufbau solch einer Datei zur Verfügung gestellt. Auch die verwendete *Python*-Bibliothek zum Umgang mit *MIDI*-Dateien wird vorgestellt.

In Kapitel 3 werden die wichtigsten Konzepte des maschinellen Lernens beschrieben. Grob können zwei große Kategorien unterschieden werden: die *überwachten* Systeme, in denen die Trainingsdaten die gewünschten Lösungen enthalten, und die *unüberwachten* Systeme, in denen dies nicht der Fall ist. Da neuronale Netze zu den überwachten Lernalgorithmen zählen, wird diese Kategorie ausführlicher behandelt. Die beiden grundlegenden *überwachten* Algorithmen sind die *lineare* und die *logistische Regression*. *Neuronale Netze* bauen auf deren Ideen auf. *Support Vector Machines* und *Entscheidungsbäume* gehören ebenfalls zu den überwachten Algorithmen, jedoch gibt es hier nur wenige Gemeinsamkeiten zu neuronalen Netzen, weshalb lediglich das grobe Vorgehen skizziert werden soll. Die bedeutendste Gruppe von *unüberwachten* Lernverfahren sind die Algorithmen zum *Clustering*. Auch das *Lernen von Assoziationsregeln* gehört zu den unüberwachten Algorithmen. Es gibt noch eine dritte Kategorie, die weder den überwachten noch den unüberwachten Systemen zugeordnet werden kann, nämlich das *Reinforcement Learning*. Hier lernt ein Agent selbstständig in einer Umgebung, in der er für falsche Entscheidungen bestraft und/oder für gute belohnt wird.

Der Schwerpunkt der Thesis liegt auf den Kapiteln 4 bis 6. Hier wird die Implementierung, Auswertung und Feinabstimmung des neuronalen Netzes zur Klassifikation der Musikstücke in der Programmiersprache *Python* vorgestellt. In Kapitel 4 werden nach einer kurzen Beschreibung der Datenerhebung die Daten aufbereitet, um sie zur Eingabe in ein neuronales Netz verwenden zu können. Hierzu werden drei verschiedene Möglichkeiten der Merkmalscodierung aufgezeigt und miteinander verglichen. Zudem werden drei Netz-Typen vorgestellt: ein *vollverbundenes Feedforward-Netz*, ein *LSTM*-Netz *mit* und eines *ohne* vorgeschaltete *konvolutionale Schichten*. In Kapitel 5 werden die neun Kombinationen aus Netztyp und Merkmalscodierung konfiguriert und diejenige Kombination mit der höchsten Korrektklassifizierungsrate auf den Validierungsdaten ausgewählt. Das ausgewählte Netz wird in Kapitel 6 noch einmal optimiert.

Aus musiktheoretischer Sicht ist es zudem interessant, welche Epochen und Komponisten sich gut oder weniger gut klassifizieren lassen, und welche Komponisten sich möglicherweise so ähnlich sind, dass eine Unterscheidung kaum oder gar nicht möglich ist. Aus diesem Grund wird das System am Ende auch auf Datensätzen trainiert, die Stücke weiterer Komponisten enthalten. Die Ergebnisse dieser Trainingsphasen und die Schlussfolgerungen daraus werden in Kapitel 7 präsentiert.

# 2. Vorbereitende Begriffsbildung

## 2.1. Maschinelles Lernen - ein Überblick

Eine allgemein gültige Definition für maschinelles Lernen gibt es nicht. Der Übergang zu herkömmlichen Programmiertechniken ist fließend, denn auch ein neuronales Netz muss selbstverständlich zunächst programmiert werden. Eine oft zitierte Definition ist die von Tom Mitchell [16]:

**Definition 1.** *„A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.“*

(Man sagt, dass ein Computerprogramm dann aus Erfahrungen $E$ bezüglich einer Aufgabe $T$ und eines Maßes für die Leistung $P$ lernt, wenn seine durch $P$ gemessene Leistung bei $T$ mit der Erfahrung $E$ anwächst.)

Wendet man diese Definition direkt auf die Klassifikation von Musik an, so ist die Aufgabe $T$ die Zuordnung von Stücken zu ihren Komponisten. Die Erfahrungen $E$ entsprechen den Trainingsbeispielen, d.h. einer Menge von Stücken, deren Komponisten bereits bekannt sind. Als Maß für die Leistung $P$ dient der Anteil der richtig klassifizierten Stücke.

Ein Pionier auf dem Gebiet des maschinellen Lernens war Arthur Samuel, der insbesondere 1959 für seine Arbeit an einer Dame-spielenden künstlichen Intelligenz Bekanntheit erlangte [21]. Er selbst war kein guter Dame-Spieler, und außer den wenigen Spielregeln in Form der erlaubten Züge wurde nichts vorgegeben, insbesondere keine Strategien. Die Idee war es, den Computer immer wieder gegen sich selbst spielen und daraus lernen zu lassen. Nach mehreren Zehntausend Spielen konnte der Computer Anfang der 1960er Jahre selbst den damaligen Landesmeister von Conneticut schlagen.

Mittlerweile gibt es viele unterschiedliche Arten von Machine-Learning-Systemen. Einige repräsentative Arten werden in Kapitel 3 vorgestellt. Grob kann man sie in folgende Kategorien einteilen [4]:

- *Überwachtes Lernen* vs. *unüberwachtes Lernen*: Beim *überwachten Lernen* ist als Eingabe eine Menge von *gekennzeichneten* („*gelabelten*“) Trainingsbeispielen notwendig, d.h. das korrekte oder beobachtete/gemessene Ergebnis (z.B. in Form eines numerischen Wertes, einer Kategorie oder einer Entscheidung) steht zur Verfügung. Aus diesen Daten lernt das System, mit dem Ziel, im Anschluss neue Beispiele mit

möglichst hoher Wahrscheinlichkeit korrekt zuzuordnen bzw. neue Aufgaben möglichst gut zu lösen. Um diese Fähigkeit zur Verallgemeinerung überprüfen zu können, müssen zusätzlich zu den Trainingsbeispielen Validierungs- und Testdaten zur Verfügung stehen, die ebenfalls mit der Lösung gekennzeichnet sind. Typische Systeme für überwachtes Lernen verwenden *lineare Regression* (zur Vorhersage eines Wertes aus einem kontinuierlichen Wertebereich), *logistische Regression* (zur Klassifikation), *Support Vector Machines*, *Entscheidungsbäume* oder *neuronale Netze*.

Beim *unüberwachten Lernen* sind die Daten nicht gekennzeichnet. Es gibt keine Vorgaben in Form von Kategorien. Aufgabe des Systems ist es, selbstständig Muster in den Daten zu erkennen und sie so zu kategorisieren. Verwendung finden hier insbesondere *Clustering*-Algorithmen sowie die *Hauptkomponentenanalyse* (*PCA*, *Principal Component Analysis*). Auch das *Lernen von Assoziationsregeln* gehört zum unüberwachten Lernen, obwohl der Algorithmus selbst wenig mit den anderen Algorithmen gemeinsam hat.

Zu keiner der beiden Kategorien gehört das *Reinforcement Learning* (*verstärkendes Lernen*). Hier beobachtet ein *Agent* eine Umgebung und wählt, anfangs zufällig, Aktionen aus. Für deren Ausführung erhält er Belohnungen oder Strafen. Dadurch entwickelt er nach und nach eine Strategie, die definiert, welche Aktion in einer gegebenen Situation auszuführen ist, um möglichst hoch belohnt zu werden.

- *Batch-Lernen* vs. *Online-Lernen*: Beim *Batch-Lernen* wird das System mit allen verfügbaren Daten trainiert, die gleichzeitig (sozusagen auf einem *Stapel*) bereitgestellt werden. Dies kann lange dauern und wird in der Regel *offline* durchgeführt. Danach wird das System in einer *Produktivumgebung* eingesetzt und läuft ohne weiteres Lernen. Stehen neue Trainingsdaten zur Verfügung, so werden diese den alten zugefügt, und das System muss mit der gesamten Menge eine erneute Lernphase durchlaufen. Da in der Praxis in den meisten Fällen einige Tausend bis Millionen Datensätze zur Verfügung stehen, werden anstelle des einzelnen Stapels kleine Pakete („*Mini-Batches*“) geeigneter Größe verwendet, die ebenfalls offline durchlaufen werden.

  Die meisten Algorithmen lassen sich so modifizieren, dass sie im *Online-Lernen* eingesetzt werden können. Hier werden dem System nach und nach einzelne Datensätze zur Verfügung gestellt, die nach deren Verarbeitung wieder verworfen werden können. Somit eignet sich diese Art des Lernens für Systeme mit kontinuierlich eintreffenden Daten und begrenzter Rechen- und Speicherkapazität. Sehr große Systeme wie Suchmaschinen oder Spam-Filter können nur so funktionieren. Von großer Bedeutung ist beim Online-Lernen die genaue Justierung der Lernrate. Diese entscheidet, ob sich das System schnell an neue Daten anpasst und ältere Daten entsprechend schnell wieder vergisst, oder ob die Anpassung langsamer erfolgt und das System dadurch weniger fehleranfällig ist.

- *Instanzbasiertes Lernen* vs. *modellbasiertes Lernen*: Hier geht es darum, wie ein System verallgemeinert. Beim *instanzbasierten Lernen* wird ein Ähnlichkeitsmaß

benötigt. Das System wählt das zum aktuellen Problem ähnlichste Trainingsbeispiel aus und gibt dessen Lösung aus.

Beim *modellbasierten Lernen* wird aus allen zur Verfügung stehenden Trainingsdaten ein Modell entwickelt, anhand dessen Vorhersagen für neue Probleme getroffen werden.

## 2.2. Klassifikation klassischer Musik nach Epoche und Komponist

Der Begriff *klassische Musik* umfasst hier nicht nur die spezielle Epoche der *Wiener Klassik*, sondern sehr allgemein die sogenannte *E-Musik* („*ernste*“ Musik im Gegensatz zur *Unterhaltungsmusik*) vom Mittelalter bis ins 20. Jahrhundert. Jede Epoche (grobe Einteilung: Mittelalter/Renaissance, Barock, Klassik, Romantik, Impressionismus und Moderne) hat ihre eigenen stilistischen Besonderheiten, ebenso jeder einzelne Komponist. Für Personen, die sich in klassischer Musik ein wenig auskennen, sollte es kein größeres Problem sein, zumindest die Epoche eines gehörten Musikstückes mit großer Sicherheit zu bestimmen. „*Experten*“ im Sinne von Menschen, die schon sehr viele Stücke gehört haben, bestimmen häufig auch den Komponisten eines ihnen unbekannten Werkes korrekt. Es gibt demnach Kriterien, anhand derer sich die Komponisten voneinander unterscheiden lassen, auch wenn die genannten Personengruppen diese Kriterien oftmals nicht benennen können. Um sie tatsächlich zu benennen, ist viel musikgeschichtliches und musiktheoretisches Fachwissen notwendig. Der Entwurf eines Expertensystems ohne maschinelles Lernen, beispielsweise mit Hilfe logischer Programmierung, würde die Codierung sehr vieler (möglicherweise unübersichtlich vieler) Regeln voraussetzen. Auch die Verwendung eines Systems zum instanzbasierten Lernen wäre sehr aufwändig in der Entwicklung, weil das Ähnlichkeitsmaß sehr viele Merkmale benötigen würde, die alle extrahiert und dem Entwickler im Voraus bekannt sein müssten. Einige wenige dieser Merkmale sollen hier kurz aufgelistet werden. Da die Klassifikation mittels neuronaler Netze kein tiefgehendes Expertenwissen voraussetzt, werden nicht alle Begriffe bis ins Detail erklärt. Dem interessierten Leser sei hierzu die *Allgemeine Musiklehre* von Wieland Ziegenrücker vorgeschlagen [32].

- **Tonumfang**: Im Laufe der Zeit erweiterte sich der Tonumfang aller Instrumente. Ein barockes Cembalo umfasst höchstens die Töne von $C$ („*großes C*“) bis $d^3$, ein heutiges Klavier reicht von $A_2$ („*subkontra A*“) bis $c^5$. Andere Instrumente der entsprechenden Epochen umfassten stets einen Teil dieses Bereiches, gingen aber (bis auf tiefe Register der Kirchenorgel) nicht darüber hinaus.

- **Polyphonie vs. Homophonie**: Viele barocke Werke sind *polyphon* (mehrstimmig) gesetzt. Dies bezeichnet die Gleichberechtigung aller Stimmen im Gegensatz zur *homophonen* Aufteilung in *Melodie* und *Begleitung*. In polyphonen Werken treten oft dieselben Motive zu unterschiedlichen Zeiten in allen Stimmen auf. Jedoch gibt es auch polyphone Werke aus späteren Epochen und homophone Werke im

Barock. Polyphonie erhöht lediglich die Wahrscheinlichkeit, dass es sich um ein Werk der Barockzeit handelt.

- **Art der Akkorde**: Der Begriff „*Akkord*“ bezeichnet gleichzeitig erklingende Töne, ein *Intervall* den Abstand zwischen zwei Tonstufen. Die Namen der Intervalle (*Prime*, *Sekunde*, *Terz*, *Quarte*, *Quinte*, *Sexte*, *Septime*, *Oktave*, *None*, ...) leiten sich von den lateinischen Ordinalzahlen ab. So bezeichnet eine *Terz* (der „dritte Ton“) beispielsweise den Abstand eines Tons zum übernächsten einer Tonleiter. Im Mittelalter entwickelte sich die Mehrstimmigkeit in Form von Quinten- und Quartenharmonik. In der Renaissance traten Terzen und Sexten hinzu, wodurch die ab dann vorherrschende Dreiklangsharmonik und später die Dur-/Moll-Tonalität vorbereitet wurde. Reine Dur- und Molldreiklänge sowie der Dominantseptakkord (der gebräuchlichste Vierklang) treten ab der Barockzeit in allen hier betrachteten Epochen auf. Je später die Epoche, desto mehr zulässige Akkorde mit insgesamt mehr unterschiedlichen Tönen kommen hinzu. Dur- und Molldreiklänge bestehen aus jeweils zwei übereinander liegenden Terzen, Septakkorde aus dreien. Bis zur Spätromantik wird die Anzahl der Terzen auf bis zu sechs erhöht, wodurch alle sieben Stammtöne jeweils einmal enthalten sind. Ab dem 20. Jahrhundert werden die Klänge zunehmend „*dissonanter*“ (*Dissonanz* = Missklang). Einige Komponisten wie z.B. Paul Hindemith (eigene Akkord-Definitionen) oder Arnold Schönberg („*Zwölftontechnik*“) führen völlig neue Regelsysteme ein, andere verzichten ganz auf Regeln und komponieren „*atonal*“, also frei von jeder Tonalität.

- **Akkordverbindungen**: Nicht nur die Akkorde selbst, sondern auch deren Abfolge unterliegt bestimmten Regeln, die umso strenger sind, je weiter die Epoche zurückliegt. Diese Regeln genauer zu erläutern, würde hier zu weit führen. Ein deutlicher Hinweis auf eine Entstehungszeit nach 1880 (oder einen Kompositionsfehler) wäre ein Auftreten von *Quintparallelen*, d.h. von zwei Stimmen im Abstand einer Quinte, die denselben Schritt in dieselbe Richtung ausführen. Ein gehäuftes Auftreten dieser Quintparallelen weist auf ein impressionistisches Werk hin.

- **Taktarten**: Auch die Anzahl möglicher Taktarten nimmt im Laufe der Zeit zu. Übliche Taktarten des Barock sind 2/4, 3/4, 4/4, 6/4, 3/8, 6/8, 9/8 und 12/8. Vor dem 20. Jahrhundert völlig unüblich sind zusammengesetzte Taktarten wie 5/8, 7/8, 5/16 oder ähnliche. Erkennbar sind sie an einem Zähler, der sich weder durch zwei noch durch drei teilen lässt.

- **Rhythmik**: Üblicherweise werden Notenwerte jeweils durch zwei geteilt, um den nächstkleineren Wert zu erhalten. Musiktheoretisch möglich ist jedoch auch jede andere Art der Teilung. Ternäre Teilung („*Triolen*“) gibt es bereits in barocken Kompositionen. Quintolen, Septolen usw. weisen auf die Epoche der Romantik oder später hin. Einige Komponisten wie z.B. Frédéric Chopin (romantische Klaviermusik) notieren in verschiedene Stimmen jeweils eine Anzahl von Tönen, die keinen gemeinsamen Teiler besitzen, z.B. sechs Töne in der linken Hand gegen 19

Töne in der rechten. Auch die Anzahl der *Synkopen* (Verschiebung der Betonung von einem Taktschwerpunkt nach vorne) nimmt im Laufe der Zeit zu.

Ergänzend sei erwähnt, dass ein menschlicher Zuhörer weitere Merkmale zur Klassifikation verwendet. Dazu gehört der *Pedalgebrauch*, der im Mittelalter ohnehin wegfällt, in barocken Stücken sehr sparsam ist, in der Klassik zunimmt und in der Romantik seinen Höhepunkt erreicht. Dasselbe gilt für die *Agogik* und ebenso für die *Dynamik*. Der Begriff *Agogik* bezeichnet leichte Veränderungen des Tempos, die dem musikalischen Ausdruck von Gefühlen dienen, die *Dynamik* betrifft die Änderung der Lautstärke. Bei der Klassifikation von Audio-Dateien könnte dies alles berücksichtigt werden. In *MIDI*-Dateien können derartige Informationen zwar codiert werden, jedoch wird in den meisten Fällen darauf verzichtet. Mittlerweile gibt es die Möglichkeit, aus eingescannten Noten *MIDI*-Dateien zu erzeugen. Dabei wird musikalischer Ausdruck nicht erfasst. Auch Audio-Dateien können in *MIDI* umgewandelt werden. Die daraus resultierenden Dateien sind jedoch häufig zur Analyse völlig ungeeignet, da die Notenwerte nicht exakt genug erfasst werden können. Die besten Dateien sind die, die von Hand erstellt oder zumindest bearbeitet werden. Soll hierbei musikalischer Ausdruck erfasst werden, so ist dies mit sehr viel Zeitaufwand verbunden. Da die im Rahmen dieser Arbeit verwendeten Dateien aus vielen unterschiedlichen Quellen stammen, fällt diese Möglichkeit ohnehin weg. Die hier implementierten neuronalen Netze werden nur die Informationen verarbeiten können, die im reinen Notentext enthalten sind. Diese beinhalten die Tonhöhen und die Dauer der Töne.

Die aufgeführten Merkmale betreffen bisher im Wesentlichen die Epochen. Die Eigenarten der einzelnen Komponisten zu unterscheiden, ist noch einmal deutlich komplizierter. Bei Verwendung eines neuronalen Netzes ist es aber gar nicht notwendig, die Merkmale im Voraus zu kennen. Im Nachhinein lassen sie sich auf Wunsch extrahieren. Dies ist aus musiktheoretischer Sicht interessant, aber zur Klassifikation an sich ebenfalls nicht notwendig.

## 2.3. MIDI

### 2.3.1. Was ist MIDI?

*MIDI* steht für „*Musical Instrument Digital Interface*“. Es handelt sich um eine Sprache, mit deren Hilfe Computer, elektronische Musikinstrumente und andere Musik-Hardware (Sequenzer, Sampler, Drumkits, Bühnen-Effektgeräte, ...) egal welcher Hersteller miteinander kommunizieren können [31][30]. Da sich der Standard [1] seit seiner Einführung Anfang der 1980er Jahre nicht geändert hat, funktioniert dies sogar unabhängig vom Alter der Geräte oder der Software. Eine *MIDI*-Datei enthält keine Audio-Signale, sondern Steuerungsdaten. Um sie abzuspielen, ist demnach ein Klangerzeuger (*Sampler*) notwendig, z.B. in Form eines Synthesizers oder einer Soundkarte. Das zentrale Steuerungsgerät zur Aufnahme, Wiedergabe und Bearbeitung von *MIDI*-Dateien ist in der Regel ein *Sequenzer*. Sequenzer und Sampler sind sowohl als Hard- als auch als Software verfügbar. Heutzutage sind sie in der Regel in einem einzigen Gerät vereint. Da es hier

lediglich um die *Analyse* von Musikstücken geht und nicht um deren Bearbeitung, sind diese Geräte nicht notwendig. Die Aufbereitung der Daten zur Eingabe in ein neuronales Netz wird direkt in *Python* vorgenommen. Auch zum Abspielen der Musikbeispiele ist kein externes Gerät notwendig. Zum Ansehen der Dateien als Notentext wurde die frei zugängliche Software *MuseScore* verwendet, die hier heruntergeladen werden kann:

`https://musescore.com/`

### 2.3.2. Aufbau einer MIDI-Datei

Eine *MIDI*-Datei ist aus sogenannten „*Chunks*“ (*Blöcken*) zusammengesetzt. Der erste Chunk ist der *Header*. Er beginnt immer mit dem String „`MThd`“ (hexadezimal: `0x4D546864`), gefolgt von vier Bytes, die die Anzahl der noch folgenden Bytes dieses Chunks angeben (immer 6 Bytes, also `0x00000006`), zwei Bytes, die das Dateiformat bezeichnen (0 für eine Spur, 1 für mehrere synchron abgespielte Spuren, 2 für mehrere Spuren, die asynchron abgespielt werden können), zwei Bytes für die Anzahl der noch folgenden Tracks der Datei und zwei Bytes für die „*Auflösung*“, d.h. die „`ticks per quarter note`“ oder auch `PPQ` (`parts per quarter`).

Alle weiteren Chunks sind *Track-Chunks* und beginnen dementsprechend jeweils mit dem String „`MTrk`“ (hexadezimal: `0x4D54726B`). Die folgenden vier Bytes geben wieder die Anzahl der noch folgenden Bytes des Tracks an. All diese Bytes codieren die eigentlichen `Track-Events`. Jedes Track-Event enthält den zeitlichen Abstand zum vorherigen Event (in *Ticks*, s.u.) und entweder

- ein *MIDI-Event* in Form einer beliebigen `Channel-Message`, oder
- ein *Meta-Event* in Form einer `System Common`- oder `System Realtime-Message`, oder
- eine `System Exclusive-Message`.

Jede „*Message*“ (Anweisung) beginnt mit einem *Statusbyte* (d.h. das höchste Bit hat den Wert 1), meist gefolgt von mindestens einem *Datenbyte* (höchstes Bit 0).

`Channel Messages` richten sich an jeweils einen von 16 möglichen Kanälen. Ihr Statusbyte besteht aus zwei „*Nibbles*“: Die vier höherwertigen Bits bezeichnen den Nachrichtentyp, die niederwertigen den adressierten *MIDI*-Kanal. Da das höchste Bit des Statusbytes `1` ist und '`1111xxxx`' den `System Messages` vorbehalten ist, können durch das höherwertige Nibble sieben Nachrichtentypen unterschieden werden. Die zur Klassifikation wichtigsten sind „`note_on`“ (Beginn mit `1000`, hexadezimal `8`) und „`note_off`“ (Beginn mit `1001`, hexadezimal `9`). Dem zweiten Nibble (Kanal) folgen zwei Datenbytes. Das erste Datenbyte bezeichnet die Tonhöhe (`note`), das zweite die Anschlagsstärke (`velocity`) des Keyboards, und damit die Lautstärke. Zu beachten ist, dass eine `note_on`-Anweisung mit `velocity=0` gleichbedeutend mit einer `note_off`-Anweisung ist. Die Codierung der Tonhöhe ist intuitiv. Das „*mittlere c*“ oder $c^1$ hat (dezimal) die

Nummer 60. In *MIDI* wird es uneinheitlich als `C3` oder `C4` bezeichnet. Pro Halbtonschritt nach oben wird nun jeweils 1 addiert bzw. nach unten subtrahiert. Dies macht das Transponieren eines kompletten Musikstückes in andere Tonarten sehr einfach.

Die `note_on`- und `note_off`-Nachrichten besitzen neben dem `note`- und `velocity`-Attribut auch ein `time`-Attribut. Es bezeichnet den Abstand zum vorigen *MIDI*- oder *Meta-Event* in *Ticks*. Wie lange solch ein Tick dauert, ist abhängig von dem im Header festgelegten `PPQ`-Wert und vom aktuellen Tempo (`BPM`, `beats per minute`, wobei mit einem *beat* unabhängig von der tatsächlichen Taktart immer eine Viertelnote gemeint ist). Die Festlegung oder Änderung des Tempos wird durch *Meta-Events* des Typs „`set_tempo`“ codiert. Diese gehören zu den im Folgenden beschriebenen `System Common`-Messages.

Neben den `note_on`- und `note_off`-Nachrichten gibt es weitere `Channel-Messages`, die jedoch zur Analyse von klassischer Klaviermusik nicht von Bedeutung sind. Hierzu zählen `Controller Change`, `Program Change` (zum Wechsel des Instrumentes), `Pitch Bend` u.v.a.

`System Common`- und `System Realtime`-Messages entsprechen den `Meta-Events`. Ihr Statusbyte beginnt immer mit `0x1111` (hexadezimal `F`). Sie richten sich an das ganze System, das alle Kanäle umfasst. Dazu zählen Befehle zur Synchronisation verschiedener Geräte wie Start- und Stop-Signale, der `MTC` (`MIDI Time Code`) und die `MIDI-Clock`. All diese spielen bei der Analyse von Musikstücken keine Rolle. Von Bedeutung sind allerdings die Meta-Messages

- „`set_tempo`“ (Beginn mit `0xFF51`) zur Bestimmung des Tempos in `BPM` (`beats per minute`),
- „`time_signature`“ (Beginn mit `0xFF58`) zur Festlegung der Taktart und
- „`key_signature`“ (Beginn mit `0xFF59`) zur Festlegung der Tonart.

`System Exclusive-Messages` richten sich nur an einen bestimmten Gerätetyp einer bestimmten Firma und können bei der Analyse der Stücke ignoriert werden.

Zur reinen Einführung in den Aufbau von *MIDI*-Dateien sollen diese Informationen zunächst genügen. Details sowie eine Auflistung aller zur Verfügung stehenden Nachrichtentypen sind in der offiziellen *MIDI*-Spezifikation von 1996 zu finden, die auf folgender Seite kostenlos heruntergeladen werden kann:

`https://www.midi.org/specifications-old/item/the-midi-1-0-specification`

### 2.3.3. Mido - die verwendete Bibliothek zur Arbeit mit MIDI-Dateien

*Mido* steht für „*Midi Objects for Python*“ und bezeichnet eine *Python*-Bibliothek zum Umgang mit *MIDI*-Dateien. Die Messages werden in intuitiv lesbarer Form dargestellt, so dass eine detaillierte Kenntnis des eigentlichen Byte- oder Hexadezimal-Codes nicht

notwendig ist. Es werden nicht alle Message-Typen unterstützt, jedoch alle zum Zwecke der Klassifikation notwendigen. Von Bedeutung sind insbesondere

- die Message-Typen „`note_on`“ und „`note_off`“ mit den Attributen `channel` (Wertebereich 0 bis 15), `note` und `velocity`, beide mit Wertebereich 0 bis 127
- die Meta-Message-Typen
  - „`set_tempo`“ mit dem einzigen Attribut `tempo`, dessen Wert in Mikrosekunden pro Viertelnote angegeben ist. Der Default-Wert von 500 000 entspricht demnach 120 Viertelnoten pro Minute.
  - „`time_signature`“ mit den Attributen `numerator` (Anzahl Schläge pro Takt), `denominator` (Wert eines Grundschlages), `clocks_per_click` (nur zum Abspielen relevant) und `notated_32nd_notes_per_beat` (hier genügt der Default-Wert von 8). Ein 3/8-Takt würde demnach codiert als

    ```
    MetaMessage('time_signature', numerator=3, denominator=8)
    ```

Der Message-Typ „`key_signature`“ mit dem einzigen Attribut `key` dient zur Angabe der Tonart. Zulässig sind hier intuitiv die Werte `C`, `Cm`, `C#`, `C#m`, ..., `Bb`, `Bbm`, `B`, `Bm`. Zu Beginn der Implementierung wurde er verwendet, um alle Stücke zur besseren Vergleichbarkeit nach *C-Dur* bzw. *a-Moll* zu transponieren. Dies erwies sich später als nicht zielführend, da die angegebene Tonart in den verwendeten Dateien häufig falsch ist. Die tatsächliche Tonart hätte anderweitig bestimmt werden müssen, was die Laufzeit der Merkmalserstellung noch einmal wesentlich verlängert hätte. Statt dessen werden die Stücke im Training in alle 12 Tonarten transponiert. Daraus resultiert eine Verzwölffachung der Trainingsbeispiele, was die in Abschnitt 3.1.3.1 erläuterte Gefahr des *Overfitting* zumindest reduziert.

Zur Merkmalserstellung sind zudem die *Notenwerte* von großer Bedeutung. Das dazu notwendige `time`-Attribut ist laut *MIDI*-Spezifikation nicht Bestandteil der Messages, sondern der Events. Es wird dennoch in *Mido* innerhalb der Messages ausgegeben. Zu beachten ist allerdings, dass die Einheit unterschiedlich ist, je nachdem ob die Messages eines einzelnen Tracks oder die der gesamten Datei aufgelistet werden. In den Messages eines Tracks ist der Wert in *Ticks* angegeben. Wird über die Messages der Datei iteriert, so findet (abhängig vom aktuellen Tempo) eine Konvertierung in Sekunden statt. Da Tonart-, Tempo- und Taktwechsel in der Regel in Track 0 codiert sind, `note_on`- und `note_off`-Messages jedoch in den folgenden Tracks, ist eine Iteration über die Tracks nicht möglich. Daher werden zur Merkmalserstellung die Messages der gesamten Datei in der Reihenfolge durchlaufen, in der sie auch beim Abspielen der Datei ausgeführt werden. Das `time`-Attribut bezeichnet dann die vergangenen Sekunden relativ zur vorigen Message.

Eine vollständige Dokumentation von *Mido* kann hier eingesehen werden:

`https://media.readthedocs.org/pdf/mido/latest/mido.pdf`

# 3. Wichtige Konzepte des maschinellen Lernens

## 3.1. Überwachtes Lernen

Wie in Abschnitt 2.1 bereits beschrieben, enthalten die Trainingsdaten beim überwachten Lernen die gewünschten Lösungen. Je nach Problem kann dies entweder ein numerischer Zielwert sein oder eine Klasse (Kategorie), eine Entscheidung oder eine Handlung. Zunächst findet eine Lernphase statt, in der die Trainingsbeispiele verwendet werden, um ein möglichst gutes Modell zu finden, anhand dessen anschließend Lösungen für neue Probleme vorgeschlagen werden können. Die beiden grundlegenden Algorithmen, auf denen viele andere Konzepte aufbauen, sind die *lineare Regression* zur Vorhersage eines Zielwertes aus einem kontinuierlichen Wertebereich und die *logistische Regression* zur Klassifikation, d.h. zur Vorhersage eines diskreten Wertes. Ein *neuronales Netz* kann prinzipiell zu beidem verwendet werden, hier wird jedoch der Schwerpunkt auf der Klassifikation liegen. Dasselbe gilt für die *Support Vector Machines*. *Entscheidungsbäume* dienen in ihrer hier erläuterten Grundform ausschließlich zur Klassifikation bzw. zur Entscheidungsfindung.

Die Abschnitte zur linearen und logistischen Regression sowie zu den Grundlagen der neuronalen Netze bis einschließlich Abschnitt 3.1.5.5 sind an den Online-Kurs *Machine Learning* von Prof. Andrew Ng [17] angelehnt, der unter folgender URL abrufbar ist:

`https://www.coursera.org/learn/machine-learning`

Da alle Algorithmen eine Menge von Trainingsbeispielen benötigen und jeweils eine oder mehrere Eingabevariablen und eine Ausgabevariable besitzen, soll hier bereits einheitlich eine Notationsweise festgelegt werden, die in den Abschnitten 3.1.1 bis 3.1.5 beibehalten wird. Es seien also

- $m$ die Anzahl der Trainingsbeispiele,
- $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$ ein Merkmalsvektor, wobei $n$ die Anzahl der Merkmale ist und $x_i$ der $i$-te Wert des Merkmals. Im späteren Verlauf wird diesem Vektor der Wert $x_0$ zugefügt, der stets 1 beträgt. Dies dient der Vektorisierung und damit der effizienten Berechnung.

- $(\mathbf{x}^{(i)}, y^{(i)})$ bezeichne das $i$-te Trainingsbeispiel, wobei $y^{(i)}$ das zu $\mathbf{x}^{(i)}$ gehörende Label ist.
- $\mathbf{X}$ sei dann die Matrix mit den Werten aller Merkmale, ohne die Labels. Jede Zeile entspricht einem Datenpunkt, d.h. die Merkmalsvektoren erscheinen in der Matrix in ihrer transponierten Form:

$$\mathbf{X} = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(n)})^T \end{bmatrix}$$

Ein überwachter Lern-Algorithmus bekommt nun eine Menge von Trainingsbeispielen und soll daraus eine Modellfunktion lernen, die mit $h_\theta$ ($h$ für *Hypothese*) bezeichnet wird, und die jeden $\mathbf{x}$-Wert auf einen $\hat{y}$-Wert abbildet. $\hat{y}$ ist also der vorhergesagte Wert, während $y$ dem tatsächlichen (gemessenen, beobachteten) Zielwert des Trainingsbeispiels entspricht. Wie die Hypothese konkret aussieht, hängt vom gewählten Algorithmus ab und wird in den entsprechenden Abschnitten angegeben. In jedem Fall aber verwendet sie *Modellparameter*, die hier je nach System mit $\theta$ bzw. $\Theta^{(i)}$ bezeichnet werden. Bei $\theta$ handelt es sich um den Parametervektor des Modells der linearen oder logistischen Regression mit dem Bias-Term $\theta_0$ und den Gewichten $\theta_1$ bis $\theta_n$ der Merkmale. Im Falle von neuronalen Netzen sind die Gewichte in mehreren Matrizen $\Theta^{(i)}$ enthalten. Ziel ist es, diese Parameter im Laufe des Trainings immer besser anzupassen, so dass die Ausgabewerte $\hat{y}^{(i)}$ möglichst nahe an die tatsächlichen Werte $y^{(i)}$ herankommen. Daraus folgt die Notwendigkeit einer *Kostenfunktion* (auch: *Verlustfunktion*, engl. *loss function*) $L(\theta)$, die große Abweichungen vom Zielwert bestraft. Wie diese aussieht, ist nicht nur abhängig vom Algorithmus. Es gibt viele Möglichkeiten für Verlustfunktionen. Einige werden in den folgenden Abschnitten eingeführt. In jedem Falle müssen die Werte des Parametervektors bzw. der Parametermatrizen des Modells so angepasst werden, dass die Verlustfunktion minimiert wird. Dies geschieht in der Praxis fast immer schrittweise durch Anwendung einer Form des *Gradientenverfahrens*. Auch hier stehen mehrere zur Auswahl. In Abschnitt 3.1.1.2 wird die Grundform erläutert.

## 3.1.1. Lineare Regression

### 3.1.1.1. Modell und Verlustfunktion

Das Modell der linearen Regression ist eine lineare Gleichung:

$$\hat{y} = h_\theta(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n$$

Enthält der Merkmalsvektor als ersten Wert $x_0 = 1$, so kann dies vektorisiert werden:

$$\hat{y} = h_\theta(\mathbf{x}) = \theta^T \mathbf{x} \tag{3.1}$$

Als Verlustfunktion bietet sich anschaulich zunächst die Wurzel der mittleren quadratischen Abweichung der jeweils vorhergesagten Werte vom tatsächlichen Wert an. In der Praxis wird aus Effizienzgründen in der Regel auf das Wurzelziehen verzichtet und nur der *mittlere quadratische Fehler* (*MSE*, *mean sqared error*) verwendet, da dies an den berechneten Parametern nichts ändert. Zudem wird der Wert am Ende durch zwei geteilt, weil sich dadurch später die partiellen Ableitungen vereinfachen und auch dies nichts am Ergebnis ändert. Die Verlustfunktion $L(\theta)$, die es zu minimieren gilt, ist demnach folgende:

$$L(\theta_0, \theta_1, \theta_2, ..., \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2 = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(\mathbf{x}^{(i)}) - y^{(i)})^2$$

Ersetzt man nun noch $h_\theta(\mathbf{x}^{(i)})$ durch die vektorisierte Form der linearen Gleichung, so erhält man folgende kompakte Schreibweise der Verlustfunktion $L(\theta)$:

$$L(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (\theta^T \mathbf{x}^{(i)} - y^{(i)})^2 \tag{3.2}$$

Theoretisch können über die *Normalengleichung* die gesuchten Werte für $\theta$ direkt bestimmt werden. In der Praxis funktioniert dies aber nur bei einer sehr begrenzten Anzahl von Merkmalen. Da das Verfahren zudem im Zusammenhang mit neuronalen Netzen ungeeignet ist, soll hier nicht weiter darauf eingegangen werden. Statt dessen wird das *Gradientenverfahren* eingesetzt, bei dem die Parameter iterativ so verändert werden, dass die Verlustfunktion minimiert wird.

#### 3.1.1.2. Das Batch-Gradientenverfahren

Speziell in der linearen Regression ist die Verlustfunktion stets *konvex*. Sie besitzt demnach nur ein *globales* Minimum und keine weiteren lokalen Extremwerte. Hier lässt sich die Idee des Gradientenverfahrens besonders gut veranschaulichen: Man initialisiere den Parametervektor $\theta$ mit zufälligen Werten und bewege sich stets in Richtung des steilsten Abstiegs. Wählt man hierbei eine geeignete Schrittweite, die umso kleiner wird, je näher man dem Minimum kommt, so konvergiert der Algorithmus bei eben diesem Minimum. Die Schrittweite wird durch die sogenannte *Lernrate* $\alpha$ bestimmt. Ist $\alpha$ zu klein, so müssen extrem viele Iterationen durchlaufen werden, und der Algorithmus konvergiert nur sehr langsam. Wird $\alpha$ dagegen zu groß gewählt, so kann der tiefste Punkt übersprungen werden, und möglicherweise landet man auf der gegenüberliegenden Seite sogar an einem höheren Punkt als zuvor. Der Algorithmus divergiert. Zu klären wären folgende Fragen:

1. Wie wird die Richtung des steilsten Abstiegs berechnet?
2. Wie wird die Lernrate $\alpha$ festgelegt, und wann gilt das Verfahren als erfolgreich beendet?
3. Mit welchen Werten wird der Parametervektor $\theta$ initialisiert?

Die **Richtung des steilsten Abstiegs** wird über die *partiellen Ableitungen der Verlustfunktion* ermittelt. Zu beachten ist hier, dass die Verlustfunktion die Summe der quadratischen Fehler *aller* Trainingsbeispiele enthält. Es werden also alle Beispiele *auf einem Stapel* verarbeitet, daher der Name *Batch*-Gradientenverfahren. Zwei Alternativen hierzu kommen in Abschnitt 3.1.1.3 zur Sprache. Aus den Gradienten der Verlustfunktion ergibt sich bereits der eigentliche Ablauf des Verfahrens, der in Pseudocode sehr knapp formuliert werden kann [17]:

**Algorithm 1** Gradientenverfahren

**repeat**
  $temp_j := \theta_j - \alpha \cdot \frac{\partial}{\partial \theta_j} L(\theta_0, \theta_1, \theta_2, ..., \theta_n)$ (for $j = 0, ..., n$)
  $\theta_j = temp_j$ (for $j = 0, ..., n$)
**until** convergence

Dadurch, dass für alle $j$ der neue Parameterwert $\theta_j$ zunächst in der Hilfsvariablen $temp_j$ gespeichert wird, ist sichergestellt, dass alle Updates *gleichzeitig* ausgeführt werden. Der Term $\frac{\partial}{\partial \theta_j} L(\theta_0, \theta_1, \theta_2, ..., \theta_n)$ bezeichnet die partielle Ableitung der Verlustfunktion nach $\theta_j$. Sie wird für alle $j$ wie folgt berechnet:

$$\frac{\partial}{\partial \theta_j} L(\theta_0, \theta_1, \theta_2, ..., \theta_n) = \frac{\partial}{\partial \theta_j} \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(\mathbf{x}^{(i)}) - y^{(i)})^2 = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(\mathbf{x}^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \quad (3.3)$$

Da $x_0^{(i)} = 1$ für alle $i$, fällt dieser Faktor bei der Ableitung nach $\theta_0$ jeweils weg. Die partiellen Ableitungen lassen sich in einem Vektor anordnen, der als *Gradient* der Funktion bezeichnet wird. Mit seiner Hilfe lässt sich für jeden beliebigen Datenpunkt die Steigung in jeder Achsenrichtung bestimmen. Der negative Gradient entspricht der gesuchten Richtung des steilsten Abstiegs. In diese Richtung kann nun ein Schritt ausgeführt werden, dessen Größe noch von der Lernrate $\alpha$ abhängt.

Nun kann Algorithmus 1 wie folgt konkretisiert werden [17]:

**Algorithm 2** Gradientenverfahren, konkreter

**repeat**
  $temp_j := \theta_j - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} (\theta^T \mathbf{x}^{(i)} - y^{(i)}) \cdot x_j^{(i)}$ (for $j = 0, ..., n$)
  $\theta_j := temp_j$ (for $j = 0, ..., n$)
**until** convergence

Um einen geeigneten Wert für die **Lernrate** $\alpha$ zu bestimmen, ist es notwendig, die Verlustfunktion während des Trainings zu beobachten und sicherzustellen, dass sie monoton fällt. Tut sie dies nicht, so muss $\alpha$ verkleinert werden. Ist $\alpha$ dagegen zu klein, so konvergiert die Funktion nur sehr langsam. Es empfiehlt sich, z.B. mit dem Wert 0.001 zu beginnen und bei Bedarf jeweils mit 10 zu multiplizieren oder durch 10 zu

dividieren. Sobald ein geeigneter Wert für $\alpha$ gefunden wurde, ist es nicht mehr notwendig, diesen im Laufe der Trainingsphase zu verkleinern, da die Verlustfunktion umso flacher und die Werte des Gradienten umso kleiner werden, je näher das Minimum liegt. Damit verringert sich die Schrittweite automatisch. Dies gilt jedoch nur bei konvexen Verlustfunktionen, was bei linearer Regression stets der Fall ist.

Die Überwachung der Verlustfunktion eignet sich auch zur Überprüfung des **Abbruchkriteriums**. Dieses ist dann erfüllt, wenn sich die Verlustfunktion mit jeder Iteration kaum noch ändert. Was *kaum* in diesem Fall bedeutet, kann individuell festgelegt werden. Ein Differenzwert $\epsilon$ der Verlustfunktion zweier aufeinander folgender Iterationen von weniger als 0.001 könnte z.B. ein solches Kriterium sein. Auch dieses Verfahren eignet sich in seiner strikten Form nur für konvexe Verlustfunktionen. Im Falle von neuronalen Netzen wäre lediglich darauf zu achten, dass die Verlustfunktion *im Mittel* fällt. Beispielsweise könnte der Mittelwert der jeweils letzten 10 oder sogar 50 Epochen berechnet und überwacht werden.

Zuletzt bleibt noch die Frage, wie der **Parametervektor $\theta$ initialisiert** wird. Empfehlenswert ist zunächst eine *Skalierung der Merkmalsvektoren* $\mathbf{x}$ derart, dass alle Werte $x_i$ ungefähr im Bereich von $-1$ bis $+1$ liegen. Erreichen lässt sich dies z.B. dadurch, dass man von jedem Wert $x_i$ zunächst den Mittelwert $\mu_i$ aller Werte dieses Merkmals subtrahiert und das Ergebnis dann durch das Maximum $\max_i$ der Werte teilt:

$$x_i := \frac{x_i - \mu_i}{\max_i}$$

Grund für die Skalierung ist, dass sich sehr große Unterschiede im Wertebereich ungünstig auf die Konvergenzgeschwindigkeit auswirken. Nach der Skalierung kann der Parametervektor $\theta$ zufällig mit kleinen Werten (z.B. zwischen $-1$ und $+1$) initialisiert werden. Eine Initialisierung aller Werte mit 0 ist bei linearer Regression möglich, in neuronalen Netzen allerdings nicht. Der komplette Algorithmus der linearen Regression sei hier noch einmal zusammengefasst:

**Algorithm 3** Lineare Regression - Lernphase

**Eingabe**: $m$ Trainingsbeispiele, jeweils mit Merkmalsvektor $\mathbf{x} \in \mathbb{R}^{n+1}$ und Label $y$.
Skaliere Merkmale, so dass alle Werte ca. im Bereich $-1$ bis $+1$ liegen.
Initialisiere Parametervektor $\theta$ mit kleinen Zufallszahlen, oder alle Werte mit 0.
Bestimme Lernrate $\alpha$ und gewünschtes $\epsilon$.
**repeat**
  $temp_j := \theta_j - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} (\theta^T \mathbf{x}^{(i)} - y^{(i)}) \cdot x_j^{(i)}$ (for $j = 0, ..., n$)
  $\theta_j := temp_j$ (for $j = 0, ..., n$)
  Berechne $L(\theta)$
  **if** $L(\theta) > L(\theta)_{old}$ **then**
    Abbruch; Neustart mit kleinerem $\alpha$
  **end if**
**until** $|L(\theta)$ - $L(\theta)_{old}| < \epsilon$
**Ausgabe**: der Parametervektor $\theta$

Im Anschluss an die Lernphase dient der ermittelte Parametervektor $\theta$ dazu, mittels der Gleichung $\hat{y} = \theta^T \mathbf{x}$ Vorhersagen für neue Beispiele zu treffen, deren Zielwerte unbekannt sind.

#### 3.1.1.3. Alternativen zum Batch-Gradientenverfahren

Je nach Anzahl der Trainingsbeispiele kann es zu zeitaufwändig sein, in der Verlustfunktion alle Beispiele zu berücksichtigen. Es bieten sich zwei Alternativen:

- Das *Stochastische Gradientenverfahren* verwendet in jedem Schritt nur ein einziges, zufällig ausgewähltes Trainingsbeispiel zur Berechnung der Verlustfunktion. Der Gradient wird also nur für diesen einen Datenpunkt berechnet. Dadurch können zwar sehr große Datenmengen verarbeitet werden, und das Verfahren eignet sich auch zum *Online-Lernen*, doch der Preis dafür sind eine höhere Ungenauigkeit und eine größere Schwierigkeit, den Verlauf der Verlustfunktion während des Trainings zu interpretieren. Denn ihre Werte sinken auch bei guter Lernrate nur im Mittel. Ansonsten können sie hin und her springen und kommen selbst in der Nähe des globalen Minimums nicht zur Ruhe.
- Einen Kompromiss bietet das *Mini-Batch-Gradientenverfahren*. Wie der Name schon vermuten lässt, wird der große Stapel mit Trainingsdaten auf mehrere kleinere Stapel verteilt, um den Gradienten auf jeweils einem dieser Stapel zu berechnen.

#### 3.1.1.4. Polynomielle Regression

Bei Problemstellungen, die sich nicht über lineare Modelle lösen lassen, können dem Modell der linearen Regression Potenzen der Merkmale oder auch Kombinationen verschiedener Merkmale als jeweils neue Merkmale zugefügt werden. Diese Technik wird als *polynomielle Regression* bezeichnet. Im *Python*-Paket *Scikit-Learn* gibt es die Klasse *PolynomialFeatures*, die z.B. bei Vorgabe von „`degree` $= 3$“ automatisch einem übergebenen Array mit Merkmalen $a$ und $b$ die Merkmale $a^3$, $a^2$, $a^2b$, $ab^2$, $b^2$ und $b^3$ hinzufügt. Aus einem Array mit $n$ Merkmalen wird so bei gewünschtem Grad $d$ ein Array mit $\frac{(n+d)!}{d!n!}$ Merkmalen. Offenbar ist dieses Vorgehen nur bei sehr begrenzter Anzahl Merkmale praktikabel. Ansonsten ist ein neuronales Netz der polynomiellen Regression in der Regel vorzuziehen.

### 3.1.2. Logistische Regression

Logistische Regression dient der *Klassifikation*, d.h. der Vorhersage einer Kategorie, oder allgemeiner formuliert, eines Wertes aus einem diskreten Wertebereich. Ein in der Literatur häufig verwendetes Anwendungsbeispiel ist die Klassifikation von E-Mails als *Spam* oder *Nicht-Spam*. Auch bei mehr als zwei Klassen (z.B. zur Erkennung handschriftlicher Ziffern oder auch bei Zuordnung von klassischen Musikstücken zu ihrer jeweiligen Epoche) werden prinzipiell immer *binäre* Klassifizierer eingesetzt, die nur zwei mögliche Ausgaben besitzen: 1 (die *positive Klasse*) oder 0 (*negative Klasse*). Für $n$ Klassen benötigt man demnach $n$ binäre Klassifizierer, von denen jeder eine Klasse erkennt und

alle anderen Klassen als *negativ* einstuft. Zum Einsatz kommen ähnliche Prinzipien wie in der linearen Regression. Zusätzlich wird die sogenannte *logistische Funktion* benötigt, die im folgenden Abschnitt 3.1.2.1 erläutert wird, und mit deren Hilfe die aus der linearen Regression bekannte Hypothesenfunktion so transformiert wird, dass sie nur Werte zwischen 0 und 1 ausgibt.

#### 3.1.2.1. Die Modellfunktion der logistischen Regression

Die Hypothesenfunktion der linearen Regression ist zur Klassifikation ungeeignet. Zum einen reagiert sie sehr empfindlich gegenüber Ausreißern, zum anderen gibt sie auch Werte größer 1 und kleiner 0 aus. Die *logistische Funktion* wird vielfach auch *Sigmoid-Funktion* genannt, ist aber ein Spezialfall von ihr. Folgende Definition wird verwendet:

$$\text{sig}(t) = \frac{1}{1 + e^{-t}} \tag{3.4}$$

Die Funktion berechnet für jede Eingabe einen Wert zwischen 0 und 1. Für Eingabewerte größer 0 konvergiert sie gegen 1, für Werte kleiner 0 gegen 0. Nimmt man als Eingabe den Zielwert der Hypothesenfunktion aus der linearen Regression (der ja immer eine Linearkombination der Merkmalswerte ist), so lässt sich die Ausgabe als Wahrscheinlichkeit interpretieren. Bei einer Wahrscheinlichkeit von z.B. mindestens 0.5 würde ein binärer Klassifizierer dann 1 ausgeben, sonst 0. (Andere Grenzen als 0.5 sind möglich).

Die Hypothesenfunktion der logistischen Regression ist also folgende:

$$h_\theta(\mathbf{x}) = \text{sig}(\theta^T \mathbf{x}) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}} \tag{3.5}$$

Ein Blick auf den Graphen der logistischen Funktion in Abbildung 3.1 zeigt, dass genau dann $h_\theta(\mathbf{x}) \geq 0.5$ gilt, wenn $\theta^T \mathbf{x} \geq 0$, und umgekehrt $h_\theta(\mathbf{x}) < 0.5$, wenn $\theta^T \mathbf{x} < 0$.

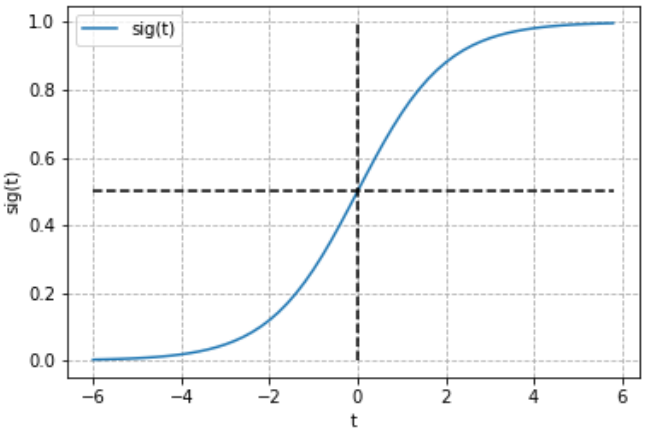



Abbildung 3.1.: Die logistische Funktion

Dadurch wird eine *Entscheidungsgrenze* definiert, die bei einer linearen Hypothese und nur zwei Merkmalen sehr anschaulich als Gerade dargestellt werden kann. Sei die Modellfunktion $h_\theta(\mathbf{x}) = \text{sig}(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$. Mit dem Parametervektor $\theta = \begin{bmatrix} -3 & 1 & 1 \end{bmatrix}^T$ wird bei $-3 + x_1 + x_2 \geq 0$ ein Beispiel als positiv eingestuft, bei $-3 + x_1 + x_2 < 0$ als negativ. Daraus ergibt sich die Entscheidungsgrenze $x_1 + x_2 = 3$. Auf dieser Grenze gilt $h_\theta(\mathbf{x}) = \text{sig}(\theta^T \mathbf{x}) = 0.5$. Die grüne Linie in Abbildung 3.2 veranschaulicht diese Grenze.

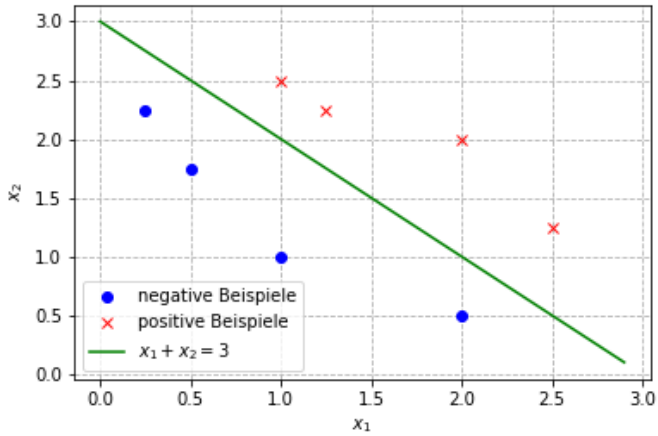



Abbildung 3.2.: Entscheidungsgrenze bei zwei Merkmalen

Die Entscheidungsgrenze ist zunächst nur eine Eigenschaft der Hypothesenfunktion mit ihren aktuellen Parameterwerten und keine der Datenpunkte. Die Parameter müssen allerdings schrittweise so angepasst werden, dass möglichst viele der Datenpunkte auf der richtigen Seite der Entscheidungsgrenze liegen, also korrekt klassifiziert werden. Dies geschieht wieder mit Hilfe einer *Verlustfunktion*.

#### 3.1.2.2. Die Verlustfunktion der logistischen Regression

Die Verlustfunktion der linearen Regression wäre bei logistischer Regression nicht konvex. Daher würde das Gradientenverfahren nicht notwendigerweise das globale Minimum finden. Für logistische Regression wird daher eine andere Verlustfunktion verwendet, die auch bei der Klassifikation durch neuronale Netze eine wichtige Rolle spielt.

Die Hypothesenfunktion $h_\theta(\mathbf{x})$ gibt einen Wert aus dem Intervall $(0, 1)$ zurück. Liegt dieser Wert nahe 1 (z.B. $h_\theta(\mathbf{x}) = 0.99$), so wird das Beispiel mit einer sehr hohen Wahrscheinlichkeit als positiv eingestuft. Ist das Beispiel tatsächlich positiv ($y = 1$), so sind die Kosten sehr niedrig. Ist es jedoch tatsächlich negativ, so soll dies mit sehr hohen Kosten bestraft werden. Umgekehrt gilt dasselbe: Ein Beispiel, das mit sehr hoher Wahrscheinlichkeit als negativ eingestuft wird (z.B. $h_\theta(\mathbf{x}) = 0.001$) und auch negativ ist ($y = 0$), wird mit sehr kleinen Kosten belegt und mit hohen Kosten bestraft, wenn es tatsächlich positiv ist.

Gleichung 3.6 definiert die Verlustfunktion für ein einziges Trainingsbeispiel und besitzt die gewünschten Eigenschaften, Abbildung 3.3 veranschaulicht sie.

$$Cost(h_\theta(\mathbf{x}), y) = \begin{cases} -\log(h_\theta(\mathbf{x})) & \text{falls } y = 1 \\ -\log(1 - h_\theta(\mathbf{x})) & \text{falls } y = 0 \end{cases} \tag{3.6}$$

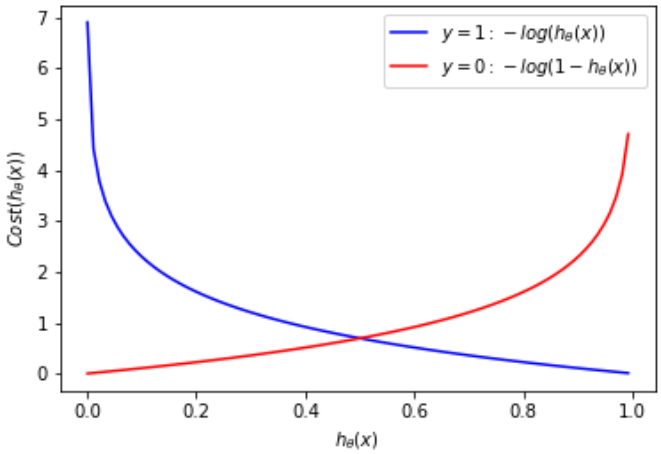



Abbildung 3.3.: Die Verlustfunktion für logistische Regression mit einem einzelnen Trainingsbeispiel

Benötigt wird nun noch eine Verlustfunktion für *alle* Trainingsbeispiele. Hierzu dient der Mittelwert der Kosten aller Beispiele:

$$L(\theta) = \frac{1}{m} \sum_{i=1}^{m} Cost(h_\theta(\mathbf{x}^{(i)}), y^{(i)}) \tag{3.7}$$

#### 3.1.2.3. Das Gradientenverfahren für logistische Regression

Ebenso wie in der linearen Regression werden die Parameter gesucht, für die die Verlustfunktion minimal wird. Es wird also auch hier ein Gradientenabstieg durchgeführt. Um die Fallunterscheidung bei der Ableitung zu vermeiden, wird die Verlustfunktion der einzelnen Beispiele (Gleichung 3.6) noch einmal umgeformt und in die Summe in Gleichung 3.7 eingesetzt:

$$L(\theta) = -\frac{1}{m} \left[ \sum_{i=1}^{m} y^{(i)} \log(h_\theta(\mathbf{x}^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(\mathbf{x}^{(i)})) \right] \tag{3.8}$$

Die erste Ableitung dieser Funktion ist dieselbe wie die der Verlustfunktion der linearen Regression, bis auf die veränderte Hypothesenfunktion $h_\theta(\mathbf{x}^{(i)})$. Algorithmus 1, die Ableitung in Gleichung 3.3 und Algorithmus 2 können exakt übernommen werden. Auch die Bestimmung der Lernrate $\alpha$ und die Kontrolle des Abbruchkriteriums erfolgen analog durch Überwachung der Verlustfunktion während des Trainings. Der Parametervektor

wird mit kleinen Zufallszahlen oder mit Nullen initialisiert. Es ergibt sich Algorithmus 4 für logistische Regression, der bis hierher allerdings nur zwei Kategorien unterscheiden kann.

**Algorithm 4** Logistische Regression für zwei Klassen - Lernphase

**Eingabe**: $m$ Trainingsbeispiele, jeweils mit Merkmalsvektor $\mathbf{x} \in \mathbb{R}^{n+1}$ und Label $y \in \{0, 1\}$.
Skaliere Merkmale, so dass alle Werte ca. im Bereich $-1$ bis $+1$ liegen.
Initialisiere Parametervektor $\theta$ mit kleinen Zufallszahlen, oder alle Werte mit 0.
Bestimme Lernrate $\alpha$ und gewünschtes $\epsilon$.
**repeat**
  $temp_j := \theta_j - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} (\frac{1}{1+e^{-\theta^T \mathbf{x}}} - y^{(i)}) \cdot x_j^{(i)}$ (for $j = 0, ..., n$)
  $\theta_j := temp_j$ (for $j = 0, ..., n$)
  Berechne $L(\theta)$
  **if** $L(\theta) > L(\theta)_{old}$ **then**
    Abbruch; Neustart mit kleinerem $\alpha$
  **end if**
**until** $|L(\theta)$ - $L(\theta)_{old}| < \epsilon$
**Ausgabe**: der Parametervektor $\theta$

Im Anschluss kann wiederum der ermittelte Parametervektor $\theta$ dazu dienen, mittels der Gleichung $\hat{y} = \frac{1}{1+e^{-\theta^T \mathbf{x}}}$ Vorhersagen für neue Beispiele zu treffen.

#### 3.1.2.4. Qualitätsmaße in der logistischen Regression

Sowohl in der linearen als auch in der logistischen Regression wird der Lernerfolg durch Verwendung einer Validierungs- und einer Testmenge bewertet. Dieses allgemeine Vorgehen wird in Abschnitt 3.1.3 erläutert. Während in der linearen Regression die Betrachtung des *mittleren quadratischen Fehlers* als Verlustfunktion in der Regel genügt, ist dies in der logistischen Regression insbesondere bei unbalancierten Datensätzen nicht der Fall. Soll ein System zum autonomen Fahren entscheiden, ob sich eine Person auf der Straße befindet und entscheidet immer auf *nein*, so liegt es fast immer richtig. Eine Trefferquote von 99.99% wäre hier aber sehr trügerisch, da das System keine einzige Person auf der Straße erkennen würde. Daher bietet sich ein anderes Verfahren an, nämlich die Verwendung einer *Konfusionsmatrix*. Im Falle eines binären Klassifizierers handelt es sich hierbei um eine $2 \times 2$-Matrix, in welche die Häufigkeiten der vier möglichen Kombinationen von Ergebnis der Klassifikation und tatsächlichem Label des Beispiels eingetragen werden:

| | Vorhersage positiv | Vorhersage negativ |
|---|---|---|
| Tatsächlich positiv | *True Positive* (*TP*) | *False Negative* (*FN*) |
| Tatsächlich negativ | *False Positive* (*FP*) | *True Negative* (*TN*) |

Daraus lassen sich nun verschiedene Qualitätsmaße ableiten, von denen die folgenden hier Verwendung finden:

- *Relevanz* (auch *Genauigkeit*, engl. *precision*) $= \frac{\textit{richtig vorhergesagte Positive}}{\textit{alle positiv Vorhergesagten}} = \frac{TP}{TP+FP}$
- *Sensitivität* (auch *Trefferquote*, engl. *recall*) $= \frac{\textit{richtig vorhergesagte Positive}}{\textit{alle tatsächlich Positiven}} = \frac{TP}{TP+FN}$
- *Korrektklassifizierungsrate* (engl. *accuracy*) $= \frac{\textit{richtig vorherges. Bsp.}}{\textit{alle Beispiele}} = \frac{TP+TN}{TP+TN+FP+FN}$

Je nach Problem kann es wünschenswert sein, die Relevanz zu erhöhen, um nur dann ein Beispiel als positiv zu deklarieren, wenn es mit ziemlich großer Wahrscheinlichkeit (z.B. mindestens 80 Prozent) auch positiv ist. Dazu muss nur die Grenze, ab der ein Beispiel als positiv klassifiziert wird, entsprechend verschoben werden:

$$\text{Vorhersage} = 1, \text{wenn } h_\theta(\mathbf{x}) \geq 0.8$$
$$\text{Vorhersage} = 0, \text{wenn } h_\theta(\mathbf{x}) < 0.8$$

Dies hat zur Folge, dass die Sensitivität sinkt, dass also weniger tatsächlich positive Beispiele auch richtig vorhergesagt werden. Angenommen, es gäbe ein Machine Learning-System für Strafprozesse, in denen eine Person als schuldig oder unschuldig klassifiziert werden soll. Aufgrund der Unschuldsvermutung müsste dann die Entscheidungsgrenze sehr hoch gewählt werden, um keinen Unschuldigen zu verurteilen. Dafür wird in Kauf genommen, dass in einigen Fällen ein tatsächlich Schuldiger freigesprochen wird.

Umgekehrt gibt es Probleme, in denen eine hohe Sensitivität gewünscht ist. Dies geht entsprechend auf Kosten der Relevanz. Ein Beispiel hierfür wäre die Auswertung von astronomischen Daten bei der Suche nach Kometen oder erdähnlichen Planeten. Ein Objekt, das nur mit einer Wahrscheinlichkeit von 0.1 als positiv eingestuft wird, sollte dann dennoch als positiv klassifiziert werden, um es anschließend genauer untersuchen zu können.

Relevanz und Sensitivität sind also *negativ korreliert*. Jedoch addieren sie sich nicht etwa zu 1, und auch der Mittelwert ist bei unterschiedlichen Algorithmen zum selben Problem nicht derselbe. In der Mehrzahl der Probleme gilt lediglich das Ziel, „*möglichst gute*“ Vorhersagen zu treffen, also „*das beste*“ Modell zu wählen. Der Mittelwert von Relevanz und Sensitivität ist hier als Maß unbrauchbar, weil damit ein Algorithmus mit sehr hoher Relevanz und sehr niedriger Sensitivität (oder umgekehrt) ausgewählt werden könnte. Ein häufig verwendetes Maß für „*möglichst gut*“ ist dagegen der „*F1-Score*“, der den *harmonischen Mittelwert* der beiden Maße berechnet:

$$F_1 = \frac{2}{\frac{1}{\textit{Relevanz}} + \frac{1}{\textit{Sensitivität}}} = 2 \cdot \frac{\textit{Relevanz} \cdot \textit{Sensitivität}}{\textit{Relevanz} + \textit{Sensitivität}} \tag{3.9}$$

Liegen mehrere Algorithmen vor, so wird derjenige mit dem höchsten *F1-Score* ausgewählt. Das Produkt im Zähler bewirkt, dass der Score gegen 0 geht, wenn einer der beiden Werte sehr klein wird. Ist einer der Werte gleich 0, so wird er als 0 definiert, obwohl der linke Teil von Gleichung 3.9 nicht definiert wäre. Nahe beieinander liegende Werte, die zudem möglichst hoch sind, werden dadurch bevorzugt.

#### 3.1.2.5. Logistische Regression mit mehr als zwei Klassen

Bei mehr als zwei Klassen muss *pro Klasse* ein Modell gelernt werden, anhand dessen entschieden wird, ob ein konkretes Beispiel zu dieser Klasse oder zu irgendeiner der anderen Klassen gehört. Abbildung 3.4 illustriert dies für drei Klassen, die durch blaue Kreise, grüne Quadrate bzw. rote Dreiecke symbolisiert werden. Die blaue Linie trennt die Klasse der blauen Kreise von den beiden anderen Klassen. Entsprechend trennt die grüne Linie die grünen Quadrate ab und die rote Linie die roten Dreiecke.

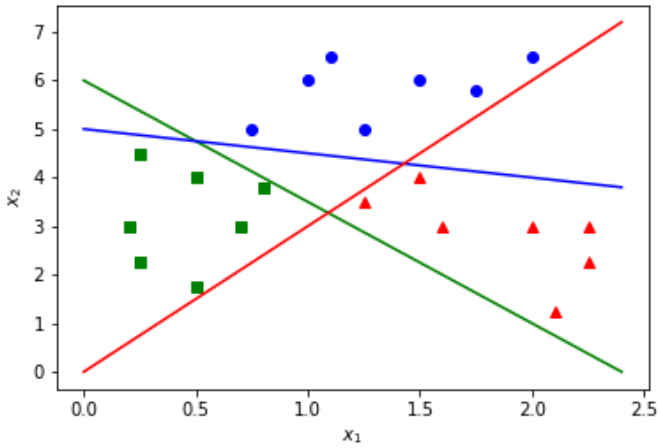



Abbildung 3.4.: Logistische Regression mit mehr als zwei Klassen

Die in Abschnitt 3.1.2.4 genannten Maße lassen sich leicht auf mehr als zwei Klassen verallgemeinern. Bei drei Klassen mit Labels von 0 bis 2 sieht die $3 \times 3$-Konfusionsmatrix folgendermaßen aus:

| | Vorhersage 0 | Vorhersage 1 | Vorhersage 2 |
|---|---|---|---|
| Tatsächlich 0 | ... | ... | ... |
| Tatsächlich 1 | ... | ... | ... |
| Tatsächlich 2 | ... | ... | ... |

Mit Hilfe der in dieser Matrix enthaltenen Werte können *Relevanz*, *Sensitivität* und der *F1-Score* wie in Abschnitt 3.1.2.4 beschrieben für jede Kategorie einzeln berechnet und daraus der Durchschnitt gebildet werden. Bei Berechnung der *Relevanz* werden demnach nur die Spalten der Tabellen betrachtet, bei Berechnung der *Sensitivität* die Zeilen. Die *Korrektklassifizierungsrate* teilt die Summe der Werte auf der Hauptdiagonalen durch die Summe aller Werte der Matrix. Bei gleich mächtigen Kategorien ist sie ebenso aussagekräftig wie der *F1-Score*.

### 3.1.3. Fehlerdiagnose bei linearer und logistischer Regression

Nur kurz angesprochen wurde bis zu dieser Stelle die Notwendigkeit, das gelernte Modell systematisch zu überprüfen. Nachdem nun die dazu notwendigen Qualitätsmaße sowie

die Verlustfunktionen definiert wurden, kann dies nachgeholt werden. Bereits zu Anfang des gesamten Machine-Learning-Prozesses werden die zur Verfügung stehenden gekennzeichneten Beispiele zufällig auf drei Gruppen verteilt: *Trainingsdaten*, *Validierungsdaten* und *Testdaten* [17]. Die Trainingsdaten sollten hierbei mit ca. 60% aller zur Verfügung stehenden Beispiele die größte Gruppe bilden, und nur sie werden zum Lernen verwendet. Die Validierungsdaten können während der Lernphase verwendet werden, um den Prozess zu beobachten und ggf. abzubrechen. Sie haben jedoch keinen Einfluss auf die Berechnung im Zuge des Gradientenverfahrens. Es wird lediglich nach jeder Iteration anhand der aktuellen Parameterwerte eine Vorhersage getroffen und das tatsächliche Label mit dieser verglichen. So kann auch auf den Validierungsdaten eine *Verlustfunktion* berechnet und im Falle eines Klassifikationsproblems die *Korrektklassifizierungsrate* (oder der *F1-Score*) bestimmt werden. Beide können erheblich von den entsprechenden Maßen auf den Trainingsdaten abweichen. In diesem Falle kann das Training vorzeitig abgebrochen werden, da die Verallgemeinerungsfähigkeit des berechneten Modells offensichtlich nicht gegeben ist. Der Vorgang kann dann mit neu eingestellten Hyperparametern neu gestartet werden. Diese Vorgehensweise empfiehlt sich insbesondere dann, wenn die Lernphase viele Stunden oder sogar Tage in Anspruch nimmt. Bei Aufgaben mit nur wenigen Trainingsbeispielen und wenigen Merkmalen kommen die Validierungsdaten oft erst nach Beendigung der Lernphase zum Einsatz.

Die Testdaten werden anfangs nicht einmal in Augenschein genommen. Sie kommen erst dann zum Einsatz, wenn mit Hilfe der Trainings- und Validierungsdaten alle Hyperparameter so eingestellt wurden, dass eine auf diesen Daten zufriedenstellende Modellfunktion zur Verfügung steht. Im Idealfall wird diese dann nur noch anhand der Testdaten bestätigt. Dazu trifft das gelernte Modell auf den Validierungs- und Testdaten seine Vorhersagen, die mit den tatsächlichen Labels der Beispiele verglichen werden. Bei linearer Regression genügt die Berechnung der Verlustfunktion, bei logistischer Regression werden die Ergebnisse zusätzlich in zwei Konfusionsmatrizen veranschaulicht, aus denen sich die Qualitätsmaße ergeben. Sollte das Ergebnis auf den Testdaten wesentlich schlechter ausfallen als das auf den Validierungsdaten, so könnte es sein, dass die Modellfunktion mit der Zeit zu sehr an die Validierungsdaten angepasst wurde. Dann muss der gesamte Prozess erneut durchlaufen werden.

Insbesondere dann, wenn nur eine kleine Datenmenge zur Verfügung steht, bietet sich zudem eine der beiden folgenden Maßnahmen an:

- *Kreuzvalidierung*: Zunächst werden nur die Testdaten beiseite gelegt. Die übrigen Daten werden in vier Teile geteilt, von denen jeweils drei zum Training verwendet werden und eins zur Validierung. Auf diese Weise wird sichergestellt, dass so viele Informationen wie möglich in die Entwicklung der Modellfunktion einfließen.

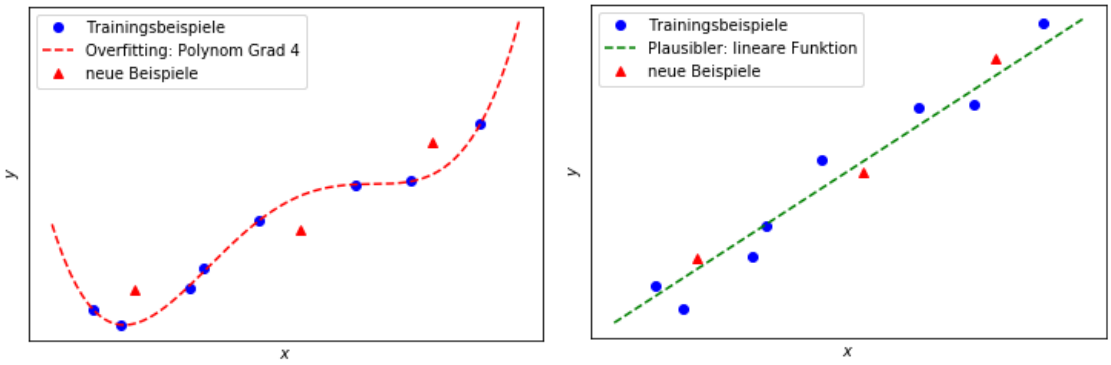
- *Datenaugmentation*: In der Bild- oder Handschriftenerkennung ist es üblich, die vorhandenen Motive durch Drehen oder Verzerren, Einfügen eines künstlichen Rauschens o.ä. zu verändern und alle veränderten Bilder in die Trainingsmenge aufzunehmen. Dies lässt sich auf die meisten anderen Bereiche übertragen. In der

Musik bietet es sich an, jedes Stück der Trainingsmenge in alle 12 Dur- (bzw. Moll-) Tonarten zu transponieren.

#### 3.1.3.1. Unter- und Überanpassung, Verzerrung und Varianz

In jedem der erläuterten Algorithmen des überwachten Lernens möchte man ein Modell finden, das einerseits die Trainingsdaten möglichst genau erfasst, sich andererseits aber auch gut verallgemeinern lässt, so dass die Labels neuer Beispiele richtig vorhergesagt werden. Beides gleichzeitig ist in der Regel schwer zu erreichen.

Bei ausreichender Anzahl Merkmale und genügend Training kann ein beliebig genaues Ergebnis auf den Trainingsdaten erzielt werden. Voraussetzung ist lediglich, dass es nicht mehrere Beispiele mit gleichen Merkmalswerten, aber unterschiedlichen Labels gibt. Die Modellfunktion passt sich dann sehr genau den Trainingsdaten an, lässt sich aber in der Regel nicht mehr auf unbekannte Beispiele verallgemeinern. Diese Fehlerquelle wird als *hohe Varianz* bezeichnet, und das Resultat ist eine *Überanpassung* (engl. *Overfitting*) der Modellfunktion. Abbildung 3.5 zeigt dieses Phänomen am Beispiel der polynomiellen Regression. Es liegen relativ wenige Trainingsdaten vor, und augenscheinlich wäre die grüne Linie in (b) eine plausible Modellfunktion. Das rote Polynom vierten Grades in (a) passt sich sehr genau den Trainingsdaten an, lässt sich aber kaum auf neue Daten verallgemeinern. Erkennbar ist dies in der Praxis daran, dass die Verlustfunktion im Training bei einem sehr kleinen Wert konvergiert, in der Validierung aber bei einem deutlich höheren.

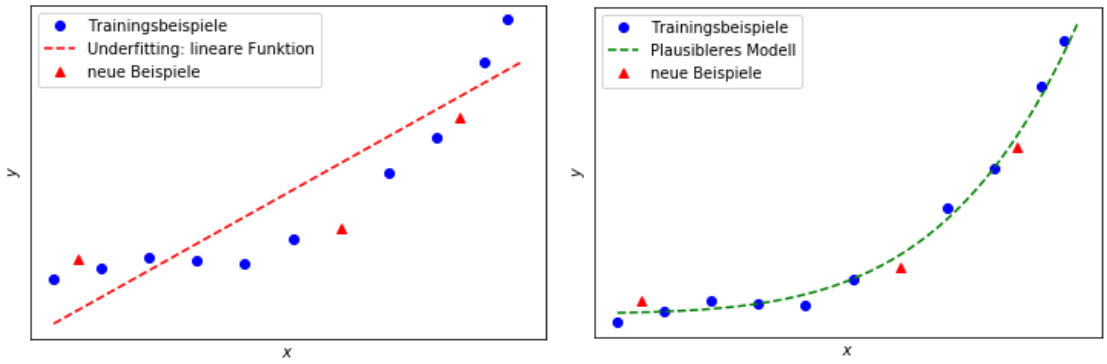



(a) Überanpassung an die Trainingsdaten

(b) Plausibleres Modell verallgemeinert besser

Abbildung 3.5.: Beispiel für Overfitting

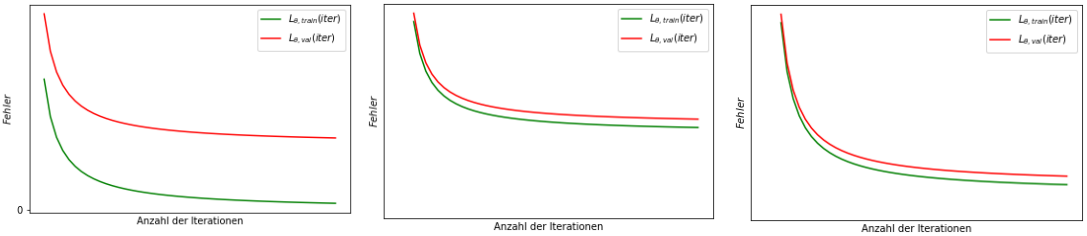
Das Gegenteil ist der Fall, wenn zu wenige Merkmale zur Verfügung stehen oder eine zu einfache Modellfunktion angenommen wird (z.B. ein lineares Modell statt eines Polynoms). Die entsprechende Fehlerquelle ist die *hohe Verzerrung* (engl. *high bias*), das Resultat eine *Unteranpassung* (engl. *Underfitting*). Die Verlustfunktionen $L_{train}$ und $L_{val}$ konvergieren dann im Laufe des Trainings zwar auf einem ähnlichen Wert, dieser bleibt aber relativ hoch. Abbildung 3.6 zeigt ein solches Beispiel.




(a) Unteranpassung an die Trainingsdaten (b) Plausibleres Modell

Abbildung 3.6.: Beispiel für Underfitting

Abbildung 3.7 demonstriert noch einmal im Überblick das Verfahren, mit dessen Hilfe festgestellt wird, ob Über- oder Unteranpassung vorliegt: Bei Überanpassung konvergiert die Verlustfunktion auf den Trainingsdaten bei einem Wert nahe 0, auf den Validierungsdaten allerdings deutlich höher. Bei Unteranpassung konvergiert sie in beiden Fällen auf einem unbefriedigend hohen Wert. Wurde ein gutes Modell ermittelt, so konvergiert die Verlustfunktion beider Datenmengen auf einem akzeptablen niedrigen Wert.




(a) Kosten bei Overfitting (b) Kosten bei Underfitting (c) Kosten bei gutem Modell

Abbildung 3.7.: Verhalten der Verlustfunktion bei Overfitting und Underfitting

Für beide Probleme gibt es mehrere mögliche Ursachen und daher auch Strategien, um diesen zu begegnen. Grundsätzlich gilt, dass bei Überanpassung die Modellfunktion zu komplex ist. Sie vermutet Zusammenhänge dort, wo es keine gibt. Entweder es stehen zu viele Merkmale zur Verfügung, von denen einige keinen Einfluss auf das Label haben, oder der gewählte Polynomgrad ist zu hoch. Eine mögliche Strategie ist demnach die Reduktion der Merkmale. Dies kann von Hand durchgeführt werden, es gibt aber auch entsprechende Algorithmen. Die zweite mögliche Strategie ist die Verwendung eines *Regularisierungsterms* und die Anpassung des darin enthaltenen *Regularisierungsparameters* $\lambda$. Wie der Term aussieht und der Parameter angepasst wird, wird in Abschnitt 3.1.3.2 erläutert.

Um einen passenden Polynomgrad zu finden, empfiehlt sich die Betrachtung der beiden

Verlustfunktionen $L_{train}(\theta)$ und $L_{val}(\theta)$, jeweils abhängig vom Polynomgrad. Es wird also der komplette Lern-Algorithmus mit verschiedenen Polynomgraden ausgeführt, die Ergebnisse werden anschließend an den Validierungsdaten getestet. Bei kleinem Grad werden beide Fehlerwerte auf einem ähnlich hohen Niveau liegen und bei steigendem Grad fallen. Ab einem bestimmten Punkt wird $J_{train}(\theta)$ weiter fallen, $J_{val}(\theta)$ jedoch wieder steigen. Dies ist ein Zeichen von zu hoher Varianz und der sich daraus ergebenden Überanpassung an die Trainingsdaten. Abbildung 3.8 veranschaulicht das Verfahren. Das beste Modell wäre in diesem Beispiel ein Polynom vierten Grades, denn hier hat die Funktion $J_{val}(\theta)$ ihr Minimum. Eine ähnliche Strategie ließe sich auch mit der Anzahl von Merkmalen durchführen. Da es aber bei realen Problemen meist sehr viele Merkmale gibt und die Anzahl der Kombinationsmöglichkeiten dann unübersichtlich groß wird, ist dies in der Praxis eher schwierig. Gesucht ist in jedem Fall eine Kombination von Merkmalen und ein Polynomgrad, für welche der Wert der Verlustfunktion auf den Validierungsdaten am Ende des Trainings minimal wird und auch nicht wesentlich höher liegt als der entsprechende Wert auf den Trainingsdaten.

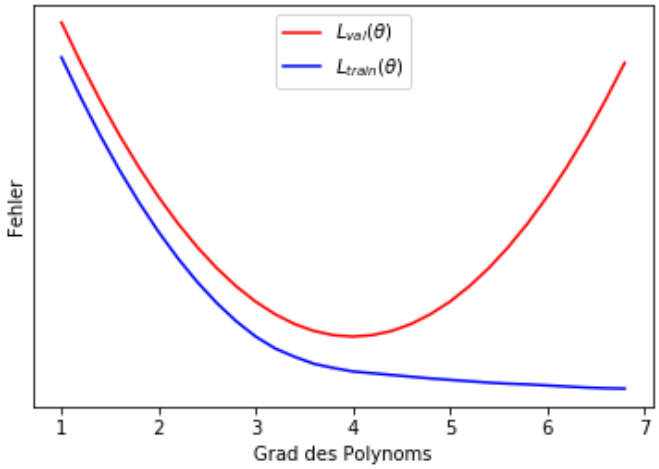



Abbildung 3.8.: $L_{train}(\theta)$ und $L_{val}(\theta)$ als Funktion des Polynomgrades der Modellfunktion

#### 3.1.3.2. Ein Hyperparameter zur Regularisierung

Folgende Überlegungen, Formeln und der Algorithmus gelten sowohl für lineare als auch für logistische Regression. Der Unterschied liegt ausschließlich in der Definition der Hypothesenfunktion $h_\theta(\mathbf{x}^{(i)})$, die aus diesem Grund hier nicht konkretisiert wird.

Durch Reduktion der Merkmale geht, sofern es sich nicht um linear abhängige Variablen handelt, ein Teil der zur Verfügung stehenden Information verloren. Ist also nicht feststellbar, welche Informationen zur Bestimmung des Labels tatsächlich notwendig sind, so gibt es die Möglichkeit, durch Einstellung des *Regularisierungsparameters* $\lambda$ den

Einfluss der Parameter und damit die Höhe der Varianz zu verringern. Der Regularisierungsterm $\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$ wird in die Verlustfunktion $L(\theta)$ wie folgt eingebaut:

$$L(\theta) = \frac{1}{2m}\Big[\sum_{i=1}^{m}(h_\theta(\mathbf{x}^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\Big] \tag{3.10}$$

Die beiden schwierig zu vereinenden Ziele, die durch Minimierung der Verlustfunktion verfolgt werden, sind folgende: Zum einen sollen die Parameter möglichst genau angepasst werden, zum anderen sollen sie möglichst klein gehalten werden, um die Hypothese zu „*vereinfachen*“ und so die Gefahr der Überanpassung zu vermindern. Die erste enthaltene Summe dient der Parameteranpassung, die zweite sorgt dafür, dass hohe Parameterwerte bestraft werden. Der Bias-Term $\theta_0$ wird bei der Regularisierung nicht berücksichtigt. Während der Minimierung der Verlustfunktion dient der Hyperparameter $\lambda$ dazu, die beiden genannten Ziele gegeneinander abzuwiegen und einen guten Kompromiss zu finden. Je größer $\lambda$ ist, desto kleiner werden die Parameter und damit die „*Ausschläge*“ der Modellfunktion. Im Extremfall, bei einem sehr großen Hyperparameter, fallen die Merkmale praktisch nicht mehr ins Gewicht, es bleibt nur noch der Biasterm $\theta_0$, und die Modellfunktion wird nahezu konstant. Bei einem extrem kleinen Hyperparameter werden große Parameterwerte nicht mehr bestraft, und die Gefahr der Überanpassung bei zu hohem Polynomgrad oder zu vielen Parametern steigt. Die Frage ist nun zunächst, wie sich der zugefügte Regularisierungsterm auf die partielle Ableitung der Verlustfunktion auswirkt. Im Anschluss wird es um die optimale Einstellung von $\lambda$ gehen.

Die partielle Ableitung der Verlustfunktion (siehe Gleichung 3.3) ändert sich nun für $j = 1, ..., n$ wie folgt:

$$\frac{\partial}{\partial\theta_j}L(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(\mathbf{x}^{(i)}) - y^{(i)})\cdot x_j^{(i)} + \frac{\lambda}{m}\cdot\theta_j \tag{3.11}$$

Entsprechend wird der Algorithmus des Gradientenverfahrens (siehe Algorithmus 2) angepasst [17]:

**Algorithm 5** Gradientenverfahren mit Regularisierung

**repeat**
- $temp_0 := \theta_0 - \alpha\cdot\frac{1}{m}\sum_{i=1}^{m}(h_\theta(\mathbf{x}^{(i)}) - y^{(i)})$
- $temp_j := \theta_j - \alpha\cdot\Big[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(\mathbf{x}^{(i)}) - y^{(i)})\cdot x_j^{(i)} + \frac{\lambda}{m}\cdot\theta_j\Big]$ (for $j = 1, ..., n$)
- $\theta_j = temp_j$ (for $j = 0, ..., n$)

**until** convergence

Zuletzt bleibt die Frage, wie der Hyperparameter $\lambda$ optimiert wird. Hier gibt es leider keine bessere Möglichkeit als systematisches Ausprobieren. Professor Andrew Ng schlägt zwölf verschiedene Werte für $\lambda$ vor [17]:

0 (also keine Regularisierung), 0.01, 0.02, 0.04, 0.08, ..., 10.24. Die Werte werden also ab 0.01 jeweils verdoppelt. Der komplette Lern-Algorithmus wird nun 12 mal unter Verwendung jeweils eines dieser 12 Werte für $\lambda$ auf den Trainingsdaten ausgeführt, bis in jedem Durchlauf $i$ (für $i = 1, ..., 12$) ein Parametervektor $\theta^{(i)}$ ermittelt wurde. Dieser Vektor wird anhand des Fehlers der Verlustfunktion $L_{val}(\theta^{(i)})$ auf den Validierungsdaten bewertet. Der Parametervektor mit dem kleinsten Fehler wird ausgewählt und noch einmal anhand der Verlustfunktion $L_{test}(\theta^{(i)})$ auf den Testdaten überprüft.

Etwas anschaulicher wird der beschriebene Prozess beim Betrachten der Graphen der Verlustfunktionen $L_{train}(\theta^{(i)})$ und $L_{val}(\theta^{(i)})$ abhängig vom Parameter $\lambda$ in Abbildung 3.9. Dies entspricht den Graphen in Abbildung 3.8, in der dieselben Funktionen, jedoch in Abhängigkeit vom Polynomgrad der Modellfunktion, dargestellt werden.

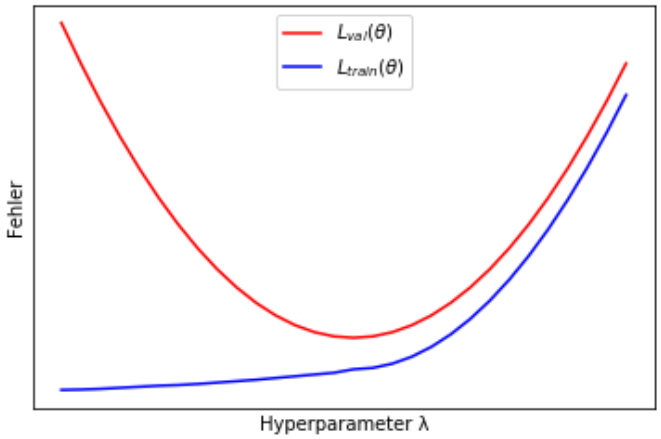



Abbildung 3.9.: $L_{train}(\theta)$ und $L_{val}(\theta)$ als Funktion des Hyperparameters $\lambda$

### 3.1.4. Zusammenfassung: Feinabstimmung bei linearer und logistischer Regression

Sowohl in der linearen als auch in der logistischen Regression müssen nicht nur die Werte des Parametervektors $\theta$ angepasst werden, sondern auch die beiden Hyperparameter $\alpha$ (Lernrate) und $\lambda$ (Regularisierungsparameter). Die Anpassung der Lernrate ist bereits in den Algorithmen 3 und 4 (Lernphase der linearen und logistischen Regression) enthalten. Um nun zusätzlich einen geeigneten Wert für den Regularisierungsparameter $\lambda$ zu ermitteln, muss idealerweise die gesamte Lernphase von einer Schleife über die möglichen Werte für $\lambda$ eingeschlossen werden. Sollte nun trotz allem am Ende ein unbefriedigendes Ergebnis auf den Validierungsdaten erzielt werden, so gibt es mehrere mögliche Vorgehensweisen, von denen jede allerdings nur entweder bei Über- oder Unteranpassung Erfolg verspricht. Eine Methode, mit deren Hilfe festgestellt werden kann, welches der beiden Probleme vorliegt, wurde bereits in Abschnitt 3.1.3.1 vorgestellt. Sobald eine der folgenden Maßnahmen durchgeführt wurde, muss erneut die gesamte Trainingsphase

ausschließlich auf den Trainingsdaten durchlaufen werden, und die ermittelten Parameterwerte auf den Validierungsdaten überprüft. Die möglichen Maßnahmen sind:

- bei Überanpassung:
  - mehr Trainingsbeispiele besorgen
  - Reduktion der Merkmale
  - Erhöhung des Regularisierungsparameters $\lambda$
- bei Unteranpassung:
  - zusätzliche Merkmale finden
  - Erhöhung des Polynomgrades
  - Senkung des Regularisierungsparameters $\lambda$

Insbesondere bei sehr großen Datenmengen mit extrem vielen Merkmalen ist das beschriebene systematische Vorgehen in der Praxis kaum konsequent durchführbar. Da ohnehin nicht alle denkbaren Werte der Hyperparameter in jeder Kombination durchprobiert werden können, helfen nur die Erfahrung und Intuition des Programmierers bei deren Einstellung. Denkbar wären in nicht allzu ferner Zukunft auch komplexere Algorithmen, die die Werte auf den Validierungsdaten selbstständig überwachen, bei Bedarf abbrechen, die Hyperparameter anpassen und den Lernvorgang von Neuem starten.

## 3.1.5. Neuronale Netze

### 3.1.5.1. Aufbau und Funktion des menschlichen Gehirns

Die Tatsache, dass selbst sehr kleine Gehirne wie die von Mäusen oder Vögeln zu Leistungen fähig sind, die bislang kein Computer zustande bringt, hat Menschen zu dem Versuch inspiriert, Gehirne zu imitieren und sogenannte *künstliche neuronale Netze* zu programmieren. Um die Idee dahinter zu illustrieren, sei hier kurz das natürliche Vorbild erläutert.

Die kleinsten Einheiten eines menschlichen oder auch tierischen Gehirns sind die *Nervenzellen* (*Neuronen*). Sehr vereinfacht dargestellt besitzt ein Neuron neben seinem *Zellkörper* (*Soma*) mehrere *Dendriten* zur Aufnahme von Signalen anderer Neuronen oder Rezeptorzellen und ein *Axon* zur Weiterleitung eines Signals an andere Zellen. Nun wird allerdings nicht jedes Signal weitergeleitet, sondern die eintreffenden Signale müssen zusammen einen bestimmten *Schwellenwert* überschreiten. Nur dann *feuert* das Neuron, indem es ein *Aktionspotenzial* immer gleicher Stärke („*Alles-oder-Nichts-Gesetz*“) entlang des Axons schickt und dadurch andere Neuronen erreicht. Kenntnisse, wie dies im Einzelnen abläuft, sind zum Grundverständnis künstlicher Intelligenz (*KI*) zunächst nicht notwendig. Doch obwohl sich die Leistungsfähigkeit von *KI*-Systemen im kognitiven Bereich in den letzten Jahren rasant gesteigert hat, gibt es nach wie vor viele Aufgaben, in denen Menschen und sogar kleine Tiere weitaus besser abschneiden. Insbesondere was emotionale und soziale Intelligenz angeht, leisten Maschinen bisher fast

nichts. Doch auch in vielen Bereichen der kognitiven Intelligenz wie Handschriften- oder Bilderkennung sind Menschen nach wie vor überlegen. Selbst ein kleines Kind braucht nur wenige Hunde zu sehen, um auf jedem Bild, in jedem Film und in der Realität alle Hunde egal welcher Rasse als Hund zu identifizieren. Computer erkennen zwar mittlerweile ähnlich gut Hunde, Gesichter, Autos oder auch handgeschriebene Buchstaben, sie brauchen dazu aber Millionen von Trainingsbeispielen und enorme Rechenkapazitäten. Ein Vorteil der Computer ist ganz einfach die Möglichkeit, diese Millionen Beispiele in kurzer Zeit betrachten zu können, ohne dabei müde zu werden, und ohne auch nur eines der Beispiele außer Acht zu lassen. Der zweite Vorteil ist die Fähigkeit, Millionen von Berechnungen in kürzester Zeit fehlerfrei durchzuführen. Beide Fähigkeiten haben zunächst nichts mit künstlicher Intelligenz zu tun. Doch nur so ist es möglich, aus der riesigen Menge an Beispielen so gut zu verallgemeinern, dass noch nie gesehene Hunde, Menschen oder handschriftliche Buchstaben der richtigen Kategorie zugeordnet werden. Die Tatsache, dass natürliche Nervensysteme bestimmte Aufgaben mit weitaus weniger Neuronen und Rechenkapazität lösen als Computer, zeigt, dass noch viel Forschungsarbeit geleistet werden muss, in Biologie, Chemie, Medizin und Informatik. Dennoch soll hier diese kurze Einführung in die Neurologie genügen.

#### 3.1.5.2. Aufbau eines künstlichen neuronalen Netzes

Das große Fernziel der *KI*-Forschung ist die Entwicklung einer Intelligenz, die der menschlichen nahe kommt oder sie sogar übertrifft. Jedoch betrifft dies nur die sogenannte „*starke KI*“. Obwohl es mittlerweile vereinzelt Maschinen gibt, die den „*Turing-Test*[1]“ bestehen, ist dies nach wie vor eine Utopie. Die „*schwache KI*“ beschränkt sich dagegen auf einzelne Anwendungsbereiche wie z.B. Sprach- oder Bilderkennung, Expertensysteme oder Computerspiele, in denen menschliche Intelligenz bereits sehr erfolgreich simuliert wird. Inspiration erhalten die Forscher zweifellos aus dem Aufbau eines natürlichen Gehirns. Jedoch ist man weit davon entfernt, es in allen Einzelheiten nachzubilden.

Ein natürliches Neuron besitzt, wie oben beschrieben, mehrere Eingänge (*Dendriten*) und einen Ausgang (*Axon*). Das Axon verzweigt sich, so dass das Signal an mehrere andere Neuronen weitergeleitet werden kann. Ein künstliches Neuron besitzt ebenfalls mehrere Eingänge für Signale und ein Ausgangssignal, das wiederum an mehrere Neuronen weitergeleitet werden kann. Im Inneren eines Neurons findet die Verarbeitung in zwei Schritten statt: Zunächst werden durch die *Übertragungsfunktion* $\Sigma$ alle eingehenden Signale zu einem einzigen vereint, anschließend modifiziert die *Aktivierungsfunktion* $\varphi$

[1] Alan Turing formulierte 1950 eine Idee, wie die Intelligenz einer Maschine überprüft werden könnte: Ein Mensch befragt einige Minuten lang schriftlich und durch eine geschlossene Tür einen Menschen und eine Maschine. Sollte er danach nicht sicher bestimmen können, welches der beiden Gegenüber die Maschine ist, so könne der Maschine ein dem Menschen ebenbürtiges Denkvermögen bescheinigt werden [26]. Jedoch wurde dieser Test 1980 von *John Searle* für ungeeignet befunden. Sein Gegenargument beinhaltet ein Gedankenexperiment, das als „*chinesisches Zimmer*“ Bekanntheit erlangte: Einem Menschen, der kein Chinesisch versteht, sei es bei Bereitstellung geeigneter Trainingsbeispiele möglich, nach einer gewissen Übungszeit Fragen zu einem chinesischen Text schriftlich derart zu beantworten, dass ein Chinese den Eindruck bekommt, es mit einem Einheimischen zu tun zu haben [23].

dieses Signal noch einmal. Abbildung 3.10 zeigt solch ein Neuron. Die *Übertragungsfunktion* berechnet eine Linearkombination der Eingangssignale, wobei die Gewichte $\theta_{ji}$ im Laufe des Trainings gelernt werden. Die *Aktivierungsfunktion* ist im allgemeinen Fall eine frei wählbare, differenzierbare Funktion. Im Falle von Klassifikationsproblemen wird fast immer eine *Sigmoid*-Funktion verwendet. In der Regel ist dies die *logistische Funktion* ($\mathrm{sig}(\mathbf{x})$), der *Tangens hyperbolicus* ($\tanh(\mathbf{x})$), der Ausgaben zwischen $-1$ und $+1$ erzeugt, oder die *Softmax-Funktion* ($\sigma(\mathbf{x})$), die jedem Knoten eine Wahrscheinlichkeit derart zuordnet, dass sich die Werte in dieser Schicht zu eins addieren. Insbesondere in der Ausgabeschicht kann dies sinnvoll sein. In verdeckten Schichten findet auch oft die sogenannte *relu*-Funktion Verwendung, die positive Werte auf sich selbst abbildet und negative auf 0. Die Aktivierungsfunktion muss im übrigen nicht notwendigerweise nichtlinear sein. In Netzen, die zur Regression dienen, kommt häufig die *Identitätsfunktion* zum Einsatz. Die Ausgabe $o_j$ des Neurons $j$ wird als dessen *Aktivierung* bezeichnet.

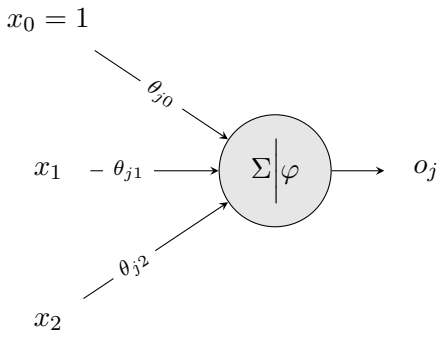



Abbildung 3.10.: Ein künstliches Neuron mit Übertragungs- und Aktivierungsfunktion

Ein künstliches neuronales Netz besteht in der Regel aus mindestens drei Schichten, die jeweils aus mindestens einem Neuron bestehen. (Ein Netz aus nur drei Neuronen wäre primitiv, widerspricht aber nicht der Definition.) Die *Eingabeschicht* (engl. *input layer*) repräsentiert die Eingabewerte, die *Ausgabeschicht* (engl. *output layer*) die Ausgabewerte. Dazwischen liegen beliebig viele *verdeckte Schichten* (engl. *hidden layers*). Die Bezeichnung „*verdeckt*“ rührt daher, dass die Ausgaben dieser Neuronen von außen unsichtbar sind. Die Funktionsweise ist aber dieselbe wie die der Ausgabeschicht.

Die Neuronenzahl der Eingabe- und Ausgabeschicht ist durch das vorliegende Problem festgelegt. Die Struktur des Netzes (Anzahl der verdeckten Schichten, Neuronen pro Schicht sowie deren Verbindungen), wird als *Topologie* des Netzes bezeichnet und muss zu Beginn festgelegt werden. In dem hier zunächst erläuterten Typ des *vollverbundenen Feedforward-Netzes* leitet jedes Neuron seine Aktivierung zu jedem Neuron der nachfolgenden Schicht weiter, entsprechend erhält auch jedes Neuron Eingangssignale von jedem Neuron der vorangehenden Schicht. So streng und unflexibel sind die Neuronen und Nervenbahnen in natürlichen Systemen nicht angeordnet. Zudem gilt in künstlichen Netzen in der Regel *nicht* das *Alles-oder-Nichts-Gesetz*.

Die in Abbildung 3.1 dargestellte *logistische Funktion* ist ein häufig verwendeter Spezialfall der *Sigmoid-Funktion*. Die Ausgaben der Knoten entstammen hier dem gesamten Wertebereich zwischen 0 und 1. Würde man sich hier auf 0 oder 1 beschränken, so wären die Leistungen weitaus schlechter. Andererseits ändert sich in natürlichen Neuronen ständig der Schwellenwert, dessen Überschreitung es „*feuern*“ lässt. Auch die Verbindung zwischen zwei Neuronen wird größer, wenn sie z.B. immer gemeinsam feuern. („*What fires together, wires together*“). Exakter formuliert es Donald Hebb: „*Wenn ein Axon der Zelle A* [...] *Zelle B erregt und wiederholt und dauerhaft zur Erzeugung von Aktionspotentialen in Zelle B beiträgt, so resultiert dies in Wachstumsprozessen oder metabolischen Veränderungen in einer oder in beiden Zellen, die bewirken, dass die Effizienz von Zelle A in Bezug auf die Erzeugung eines Aktionspotentials in B größer wird* [11].“ Solange diese Prozesse in künstlichen Netzen nur unzureichend simuliert werden können, ist es wohl ein sinnvoller Kompromiss, dies durch Nichteinhaltung der *Alles-oder-Nichts-Regel* zu kompensieren.

Abbildung 3.11 zeigt ein sehr einfaches Netz mit drei Eingabeknoten in der ersten Schicht, einer verdeckten Schicht mit ebenfalls drei Knoten und einem einzigen Knoten in der Ausgabeschicht. Ein solches Netz kann nur zwei Klassen unterscheiden. Gibt es mehr Kategorien, so braucht das Netz so viele Ausgabeknoten wie Kategorien. Die hochgestellte ‘(2)’ in der verdeckten Schicht sagt lediglich, dass es sich um die zweite Schicht handelt, die Indizes nummerieren die Knoten in jeder einzelnen Schicht. Entsprechend der logistischen Regression enthält das Netz die Knoten $x_0$ und $a_0^{(2)}$. Dabei handelt es sich in allen Schichten außer der Ausgabeschicht um das *Bias-Neuron*. Da sein Wert auch hier immer 1 beträgt und es daher keinen Eingang aus den Knoten der vorigen Schicht benötigt, wird es ab sofort beim Zeichnen von Netzen und beim Zählen der Knoten wie allgemein üblich weggelassen. Die Anzahl der Knoten pro Schicht ist unabhängig von der Anzahl der Knoten anderer Schichten. Wie viele verdeckte Schichten und Knoten pro Schicht ein Netz benötigt, ist bei komplexen Problemen oft schwer zu bestimmen. Einige Richtlinien hierzu werden in Abschnitt 3.1.5.7 genannt.

Im Verlauf der Arbeit stellte sich heraus, dass ein *vollverbundenes Feedforward-Netz* im Zusammenhang mit Musik nicht mächtig genug ist. Daher werden insbesondere den *Convolutional Neural Networks* (ein deutscher Name wie *faltendes neuronales Netzwerk* ist hier nicht gebräuchlich) und den *LSTM-Netzen* (einem Spezialfall *rekurrenter Netze*) gesonderte Abschnitte gewidmet.

#### 3.1.5.3. Die Modellfunktion eines neuronalen Netzes

Die Funktionsweise eines neuronalen Netzes soll hier zunächst am Beispiel des Netzes in Abbildung 3.11 erklärt werden. Vieles baut auf den Prinzipien der logistischen Regression auf. Die Knoten der Eingabeschicht repräsentieren lediglich die Eingabewerte. In jeder weiteren Schicht erhält jedes Neuron von jedem Knoten der vorherigen Schicht (plus Bias-Neuron) ein Eingabesignal. Die *Aktivierung* eines Neurons (also der Wert, der über das „*Axon*“ an die Knoten der nächsten Schicht weitergeleitet wird), ergibt sich entsprechend der logistischen Regression aus einer Linearkombination der Eingabesignale, die mittels

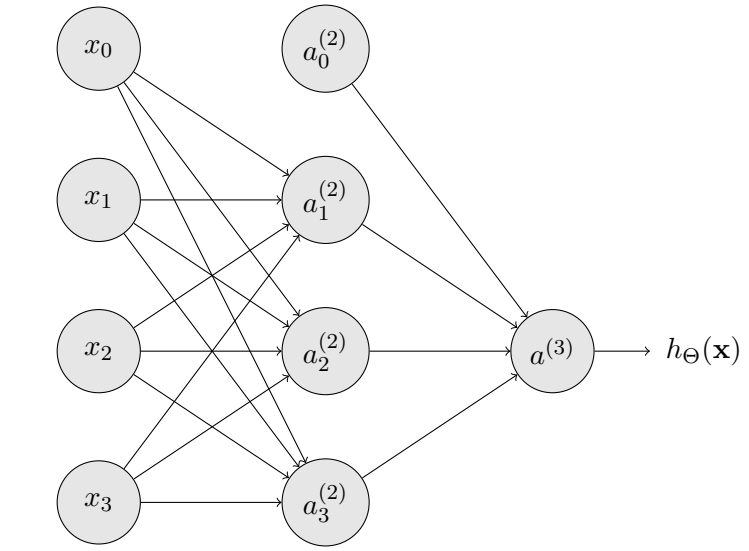



Abbildung 3.11.: Aufbau eines einfachen neuronalen Netzes

der logistischen Funktion $\text{sig}(t)$ auf einen Wert zwischen 0 und 1 abgebildet wird. Die Gewichtung der einzelnen Signale wird durch die Matrix $\Theta^{(l-1)}$ vorgegeben, wobei $l$ die gerade zu berechnende Schicht bezeichnet. Sei $s_{l-1}$ die Anzahl der Knoten in Schicht $l-1$ und $s_l$ die Anzahl der Knoten in Schicht $l$ (jeweils ohne das Bias-Neuron), so ist $\Theta^{(l-1)}$ eine $s_l \times s_{l-1}+1$ - Matrix. Im Beispiel-Netzwerk hätte demnach $\Theta^{(1)}$ drei Zeilen und vier Spalten, und $\Theta^{(2)}$ hätte eine Zeile und vier Spalten, wäre also ein Zeilenvektor:

$$\Theta^{(1)} = \begin{bmatrix} \Theta_{10}^{(1)} & \Theta_{11}^{(1)} & \Theta_{12}^{(1)} & \Theta_{13}^{(1)} \\ \Theta_{20}^{(1)} & \Theta_{21}^{(1)} & \Theta_{22}^{(1)} & \Theta_{23}^{(1)} \\ \Theta_{30}^{(1)} & \Theta_{31}^{(1)} & \Theta_{32}^{(1)} & \Theta_{33}^{(1)} \end{bmatrix}, \qquad \Theta^{(2)} = \begin{bmatrix} \Theta_{10}^{(2)} & \Theta_{11}^{(2)} & \Theta_{12}^{(2)} & \Theta_{13}^{(2)} \end{bmatrix}$$

Auf diese Art ergeben sich die Aktivierungen aller Neuronen aus der Linearkombination der Werte der Vorgängerschicht mit den Parametern der entsprechenden Zeile der dazwischen liegenden Matrix. Der Grund, weshalb alle Gewichte in Matrizen gespeichert werden, liegt in der übersichtlichen Notation und effizienten Berechnung mit Hilfe von *Vektorisierung*:

$$\mathbf{a}^{(2)} = \text{sig}(\Theta^{(1)}\mathbf{x}), \qquad h_\Theta(\mathbf{x}) = \mathbf{a}^{(3)} = \text{sig}(\Theta^{(2)}\mathbf{a}^{(2)})$$

Auf diese Art werden die Informationen aus den Eingabedaten vorwärts durch das Netz propagiert und verarbeitet. In jeder Schicht $l$ (für $l \geq 2$) findet eine Matrix-Vektor-Multiplikation $\Theta^{(l-1)}\mathbf{a}^{(l-1)}$ statt, auf deren Ergebnisvektor elementweise die logistische Funktion $\text{sig}(t)$ angewendet wird. Der sich ergebende Vektor $\mathbf{a}^{(l)}$ enthält die Aktivierungswerte der einzelnen Knoten in Schicht $l$.

Da das beschriebene Vorgehen problemlos auch auf mehrere Ausgabeknoten angewendet werden kann, soll zur Vermeidung weiterer Fallunterscheidungen bereits hier auf diesen Fall verallgemeinert werden. Abbildung 3.12 zeigt ein Netzwerk mit zwei verdeckten Schichten und drei Ausgabeknoten. Jeder Ausgabeknoten dient zur binären Klassifikati-

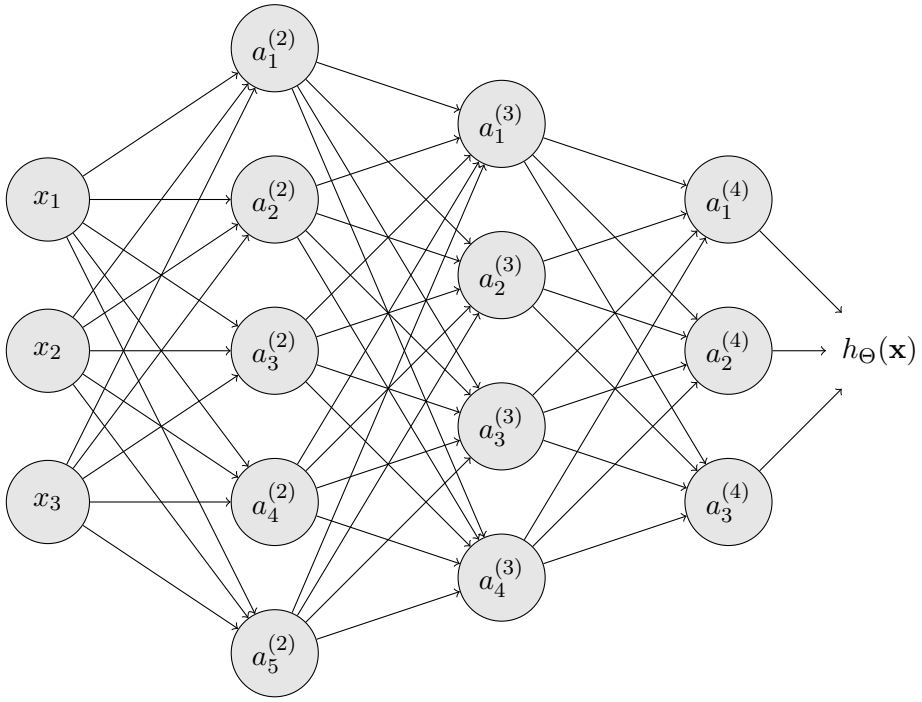



Abbildung 3.12.: Aufbau eines Netzes zur Unterscheidung von drei Klassen

on, entsprechend der logistischen Regression mit mehreren Klassen. Knoten $a_1^{(4)}$ könnte z.B. für „*Bach – ja oder nein*“ stehen, Knoten $a_2^{(4)}$ für „*Mozart – ja oder nein*“ und Knoten $a_3^{(4)}$ für „*Schubert – ja oder nein*“. Im Idealfall hätte man gerne einen Vektor $\mathbf{a}^{(4)}$, dessen Werte nahe an

$$\begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\end{bmatrix} \text{ oder } \begin{bmatrix}0\\0\\1\end{bmatrix}$$

liegen. Diese drei Vektoren bezeichnen allerdings nur die drei möglichen *Labels* der Trainingsbeispiele. In der Realität wird der Vektor der letzten Schicht nie genau so aussehen, da die logistische Funktion zwar gegen 0 und 1 konvergiert, diese Werte jedoch nie erreicht. In einem ungünstigen Fall könnten sogar zwei oder alle binären Klassifizierer auf „*ja*“ (oder eben auf „*nein*“) entscheiden. Dies ist in der Praxis jedoch kein Problem. In der Lernphase werden die Fehler aller Ausgabeknoten einzeln betrachtet und durch das Netzwerk zurück propagiert, und später in der Produktivumgebung bestimmt das Maximum der Elemente des Ausgabevektors, zu welcher Kategorie das Beispiel gehören soll. Um tatsächlich Wahrscheinlichkeiten zu erhalten, die sich zu 1 addieren, kann die

logistische Funktion in der Ausgabeschicht auch durch die *Softmax*-Aktivierungsfunktion ersetzt werden.

Die Frage ist nun noch, woher die Gewichte in den Matrizen $\Theta^{(l)}$ stammen und wie sie nach und nach angepasst werden. Ebenso wie in der logistischen Regression werden sie zu Beginn initialisiert. Verwendet werden in der Regel Zufallszahlen aus dem Intervall [-1, +1]. Nach jeder Vorwärtspropagation eines Trainingsbeispiels durch das Netzwerk tritt am Ausgabeknoten ein Fehler im Vergleich zum tatsächlichen Label $y$ auf, nämlich $h_\Theta(\mathbf{x}) - y$. (Bei mehreren Ausgabeknoten tritt solch ein Fehler an jedem Knoten auf.) Aus den Fehlern aller Trainingsbeispiele ergibt sich die im folgenden Abschnitt 3.1.5.4 benannte *Verlustfunktion*, die es wiederum zu minimieren gilt. Hierzu werden die Fehler zurück durch das Netzwerk geführt und dabei die Parameter-Matrizen $\Theta^{(l)}$ korrigiert. Dies ist Thema des Abschnittes 3.1.5.5.

#### 3.1.5.4. Die Verlustfunktion eines neuronalen Netzes

Auch die Verlustfunktion lässt sich gut aus derjenigen der logistischen Regression herleiten. Der geschlossenen Form ohne Regularisierung (siehe Gleichung 3.8) wird lediglich eine weitere Summe zugefügt, die innerhalb der Summe über alle Trainingsbeispiele alle Ausgabeknoten durchläuft. (Diese zusätzliche Summe entfällt also bei einem einzelnen Ausgabeknoten.) Der Regularisierungsterm enthält nun drei verschachtelte Summationen anstatt einer. Dieser Unterschied rührt daher, dass die Parameter in der logistischen Regression in einem einzelnen *Vektor* gespeichert sind, dessen quadrierte Werte aufsummiert und anschließend normiert werden. Ein neuronales Netz braucht dagegen zwischen zwei aufeinander folgenden Schichten *jeweils* eine Parameter*matrix*. Insofern müssen nun die quadrierten Gewichte aller Zeilen (innere Summe) und Spalten (mittlere Summe) aller $L-1$ Matrizen (äußere Summe) aufsummiert werden. In der mittleren Summe (Aufsummierung der Spalten) ist zu beachten, dass die Bias-Terme in Spalte 0 bei der Regularisierung nicht berücksichtigt werden und die Summe daher ebenfalls bei 1 und nicht bei 0 beginnt. Sei nun $L$ die Anzahl der Schichten und $s_l$ die Anzahl der Neuronen in Schicht $l$. Es ergibt sich folgende angepasste Verlustfunktion:

$$L(\Theta) = -\frac{1}{m}\Big[\sum_{i=1}^{m}\sum_{k=1}^{s_L} y_k^{(i)} \log(h_\Theta(\mathbf{x}^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(\mathbf{x}^{(i)}))_k)\Big] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2 \tag{3.12}$$

Im Jahre 2014 veröffentlichte das Team um Geoffrey Hinton einen Artikel [25], in dem das *Dropout*-Verfahren vorgestellt wird, welches sich speziell in neuronalen Netzen als Alternative zur Regularisierung anbietet. Da dieses sich zumindest im vorliegenden Anwendungsfall der Klassifikation von Musikstücken als effektiver herausstellte, wird der Regularisierungsterm ab hier nicht mehr berücksichtigt. Der letzte Summand in Gleichung 3.12 entfällt damit. Dies vereinfacht auch den im folgenden Abschnitt beschriebenen *Backpropagation-Algorithmus*.

#### 3.1.5.5. Fehlerrückführung - ein Spezialfall des Gradientenabstieges

Die Grundlagen des Verfahrens der Fehlerrückführung (engl. *Backpropagation*) wurden bereits Anfang der 1960er Jahre gelegt [22]. Das große Potenzial im Zusammenhang mit neuronalen Netzen zeigte sich 1986, als David Rumelhart, Geoffrey Hinton und Ronald Williams den Algorithmus im *Deep Learning* anwendeten [19].

Für jedes Trainingsbeispiel tritt an jedem Ausgabeknoten ein Fehler im Vergleich zum entsprechenden Element des Labels $\mathbf{y}$ (0 oder 1) auf, der mit bestimmten Kosten belegt ist. Es ergibt sich die in Gleichung 3.12 definierte Verlustfunktion, die es nun wieder zu minimieren gilt. Im Gegensatz zu den Verlustfunktionen der linearen und logistischen Regression ist diese jedoch nicht konvex. Daher ist nicht garantiert, dass durch Gradientenabstieg das globale Minimum gefunden wird. Die Funktion kann auch in einem lokalen Minimum konvergieren. Um ein gutes Ergebnis zu erzielen, muss die Lernphase mehrmals mit unterschiedlich initialisierten Parametermatrizen $\Theta^{(l)}$ durchlaufen werden.

Nun stellt sich aber zunächst die Frage, wie der Gradientenabstieg hier überhaupt abläuft, denn Fehler treten nicht nur an den Ausgabeknoten, sondern auch an allen inneren Knoten auf. Die tatsächlichen Werte sind dort nicht bekannt, denn die idealen Parametermatrizen sollen ja erst ermittelt werden. Jedoch kommt hier der Ausdruck „*Fehlerrückführung*“ ins Spiel. Die Fehler $\delta_j^{(L)}$ an den Ausgabeknoten in Schicht $L$ sind bekannt, ebenso die Rechenvorschriften der Vorwärtspropagation, aufgrund derer sie zustande kamen. Für jede Schicht $l$ mit $2 \leq l \leq L$ gilt nämlich

$$\mathbf{a}^{(l)} = \text{sig}(\Theta^{(l-1)}\mathbf{a}^{(l-1)}) \qquad (\text{mit } \mathbf{a}^{(1)} = \mathbf{x} \text{ und } h_\Theta(\mathbf{x}) = \mathbf{a}^{(L)}). \tag{3.13}$$

Diese Informationen genügen, um in umgekehrter Richtung, d.h. beginnend von der Ausgabeschicht zurück bis zu Schicht 2, für jeden Knoten $j$ in jeder Schicht $l$ des Netzes mit $2 \leq l \leq L$ den Fehlerwert $\delta_j^{(l)}$ zu bestimmen:

$$\begin{aligned}
\boldsymbol{\delta}^{(L)} &= \mathbf{a}^{(L)} - \mathbf{y} \\
\boldsymbol{\delta}^{(l)} &= (\Theta^{(l)})^T \boldsymbol{\delta}^{(l+1)} .* \text{sig}'(\Theta^{(l-1)}\mathbf{a}^{(2)}) \quad (\text{for } l = L-1, ..., 2)
\end{aligned}$$

(Der Operator „.*“ bezeichnet dabei die elementweise Multiplikation.)

Die partiellen Ableitungen der Verlustfunktion lassen sich mit Hilfe dieser Werte überraschend gut vereinfachen:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} L(\Theta) = a_j^{(l)} \delta_i^{(l+1)} \tag{3.14}$$

Nun kann der Backpropagation-Algorithmus wie folgt zusammengefasst werden [17]:

**Algorithm 6** Backpropagation-Algorithmus

**Eingabe**: $m$ Trainingsbeispiele $\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), ..., (\mathbf{x}^{(m)}, \mathbf{y}^{(m)})\}$
Initialisiere $\Delta_{ij}^{(l)} = 0$ für alle $l, i$ und $j$
// Matrix $\Delta$ summiert die Gewichtsänderungen aller Trainingsbeispiele auf.
**for** i=1 to m **do**
Setze $\mathbf{a}^{(1)} = \mathbf{x}^{(i)}$ // Anlegen des nächsten Trainingsbeispiels
Vorwärtspropagation zur Berechnung aller Aktivierungswerte $\mathbf{a}^{(l)}$ für $2 \leq l \leq L$
Berechne Fehler aller Neuronen der Ausgabeschicht: $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - \mathbf{y}^{(i)}$
Fehlerrückführung zur Berechnung aller Fehlerwerte $\boldsymbol{\delta}^{(l)}$ für $L - 1 \geq l \geq 2$
Aktualisiere $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
**end for**
Berechne $D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$
**Ausgabe**: die partielle Ableitung der Verlustfunktion $\frac{\partial}{\partial \Theta_{ij}^{(l)}} L(\Theta) = D_{ij}^{(l)}$

Nach Ablauf des Backpropagation-Algorithmus können mit Hilfe der ermittelten Matritzen $D_{ij}^{(l)}$ die Parametermatrizen $\Theta^{(l)}$ korrigiert werden. Dies geschieht wiederum simultan, entsprechend Algorithmus 1, hier über die Werte aller Zeilen und Spalten der $L - 1$ Matrizen. Nach jeder Iteration (jeweils über alle Trainingsbeispiele) wird die Verlustfunktion $L(\Theta)$ berechnet, um sicherzustellen, dass sie im Mittel fällt. Sobald dies über mehrere Epochen hinweg nicht der Fall ist, ist ein Neustart mit verkleinerter Lernrate $\alpha$ notwendig. Zudem muss sichergestellt werden, dass die Verlustfunktion am Ende des Trainings konvergiert. Dazu wird entweder ein Abbruchkriterium festgelegt, oder die Anzahl der Iterationen wird anhand von Erfahrungswerten im Voraus festgelegt und das Ergebnis überprüft. Da nicht garantiert werden kann, dass es sich bei dem ermittelten Minimum um das globale Minimum der Verlustfunktion handelt, muss die gesamte Lernphase mit unterschiedlich initialisierten Parametermatrizen durchlaufen werden.

#### 3.1.5.6. Ein einfaches Beispiel für Vorwärtspropagation und Fehlerrückführung

Der Ablauf der Forward- und Backpropagation wird hier an einem sehr einfachen Beispiel veranschaulicht. Auf die logistische Funktion als Aktivierungsfunktion wird verzichtet und statt dessen die Identitätsfunktion verwendet. Es handelt sich demnach um ein *Regressionsproblem*, die Ausgabe ist keine Klasse, sondern ein *Zielwert*. Gegeben sei das Netz in Abbildung 3.13, an dem bereits das einzige Trainingsbeispiel mit $\mathbf{x} = [-2 \quad 3]^T$ und dem Label $y = 1$ angelegt ist. Als Lernrate wird $\alpha = 0.1$ ausgewählt. Die Gewichte wurden bereits zufällig initialisiert mit

$$\Theta^{(1)} = \begin{bmatrix} 0.5 & 0.3 & 0.6 \\ 0.2 & 0.8 & 0.1 \end{bmatrix}, \quad \Theta^{(2)} = \begin{bmatrix} 0.7 & 0.4 & 0.9 \end{bmatrix}.$$

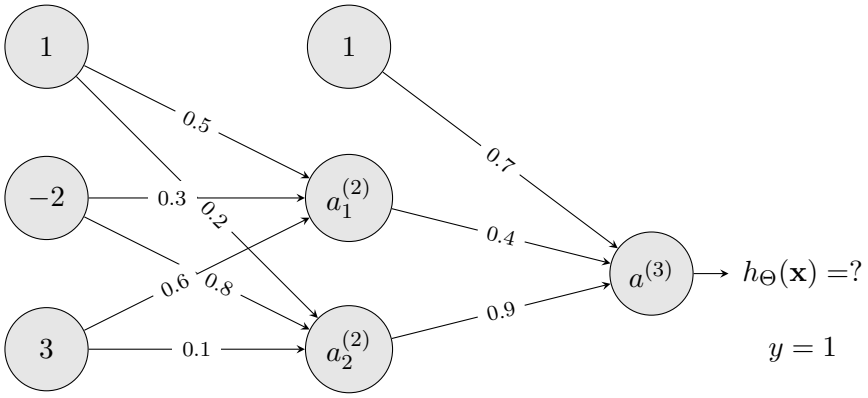



Abbildung 3.13.: Beispiel neuronales Netz, Ausgangssituation

Mittels Vorwärtspropagation werden nun die Werte für $a_1^{(2)}$, $a_2^{(2)}$ und $a^{(3)}$ berechnet:

$$\begin{aligned} \mathbf{a}^{(2)} = \Theta^{(1)}\mathbf{x}, \quad & \text{also} \quad a_1^{(2)} = 0.5 \cdot 1 + 0.3 \cdot (-2) + 0.6 \cdot 3 = 1.7 \\ & \text{und} \quad a_2^{(2)} = 0.2 \cdot 1 + 0.8 \cdot (-2) + 0.1 \cdot 3 = -1.1 \\ \mathbf{a}^{(3)} = \Theta^{(2)}\mathbf{a}^{(2)}, \quad & \text{also} \quad a^{(3)} = 0.7 \cdot 1 + 0.4 \cdot 1.7 + 0.9 \cdot (-1.1) = 0.39 \end{aligned}$$

Der Fehler am Ausgabeknoten kann nun berechnet werden durch

$$\delta^3 = h_\Theta(\mathbf{x}) - y = 0.39 - 1 = -0.61.$$

Damit ergibt sich die in Abbildung 3.14 gezeigte Situation. Die berechneten Aktivierungswerte sind blau hervorgehoben, der Fehler am Ausgabeknoten rot.

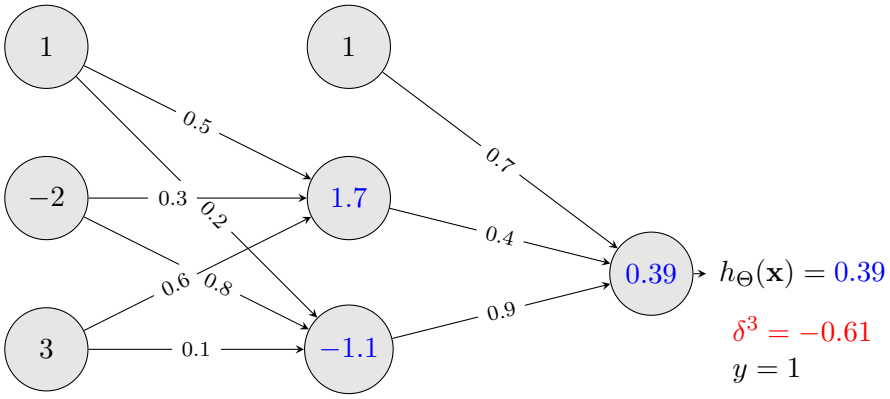



Abbildung 3.14.: Beispiel neuronales Netz, nach Vorwärtspropagation

Nun wird der Fehler am Ausgabeknoten zurück durch das Netz propagiert. Dabei ergeben sich die Fehler an den Knoten der verdeckten Schicht, also $\delta_1^{(2)}$ und $\delta_2^{(2)}$, durch folgende Berechnung:

$$\boldsymbol{\delta}^{(2)} = (\Theta^{(2)})^T \delta^{(3)} = \begin{bmatrix} 0.7 \\ 0.4 \\ 0.9 \end{bmatrix} \cdot (-0.61) = \begin{bmatrix} -0.427 \\ -0.244 \\ -0.549 \end{bmatrix}$$

Nun werden die partiellen Ableitungen der Verlustfunktion bestimmt, mit Hilfe derer (unter Einbeziehung der Lernrate $\alpha$) im Anschluss die Gewichte in $\Theta^{(2)}$ aktualisiert werden:

$$\frac{\partial}{\partial \Theta_{10}^{(2)}} L(\Theta) = D_{10}^{(2)} = a_0^{(2)} \delta_1^{(3)} = 1 \cdot (-0.61) = -0.61$$
$$\frac{\partial}{\partial \Theta_{11}^{(2)}} L(\Theta) = D_{11}^{(2)} = a_1^{(2)} \delta_1^{(3)} = 1.7 \cdot (-0.61) = -1.037$$
$$\frac{\partial}{\partial \Theta_{12}^{(2)}} L(\Theta) = D_{12}^{(2)} = a_2^{(2)} \delta_1^{(3)} = -1.1 \cdot (-0.61) = 0.671$$

$$\Theta_{10}^{(2)} = 0.7 - 0.1 \cdot (-0.61) = 0.761$$
$$\Theta_{11}^{(2)} = 0.4 - 0.1 \cdot (-1.037) = 0.5037$$
$$\Theta_{12}^{(2)} = 0.9 - 0.1 \cdot 0.671 = 0.8329$$

Dasselbe wird für $\Theta^{(1)}$ durchgeführt. Die partiellen Ableitungen der Verlustfunktion sind folgende:

$$\frac{\partial}{\partial \Theta_{10}^{(1)}} L(\Theta) = D_{10}^{(1)} = a_0^{(1)} \delta_1^{(2)} = 1 \cdot (-0.244) = -0.244$$
$$\frac{\partial}{\partial \Theta_{11}^{(1)}} L(\Theta) = D_{11}^{(1)} = a_1^{(1)} \delta_1^{(2)} = -2 \cdot (-0.244) = 0.488$$
$$\frac{\partial}{\partial \Theta_{12}^{(1)}} L(\Theta) = D_{12}^{(1)} = a_2^{(1)} \delta_1^{(2)} = 3 \cdot (-0.244) = -0.732$$
$$\frac{\partial}{\partial \Theta_{20}^{(1)}} L(\Theta) = D_{20}^{(1)} = a_0^{(1)} \delta_2^{(2)} = 1 \cdot (-0.549) = -0.549$$
$$\frac{\partial}{\partial \Theta_{21}^{(1)}} L(\Theta) = D_{21}^{(1)} = a_1^{(1)} \delta_2^{(2)} = -2 \cdot (-0.549) = 1.098$$
$$\frac{\partial}{\partial \Theta_{22}^{(1)}} L(\Theta) = D_{22}^{(1)} = a_2^{(1)} \delta_2^{(2)} = 3 \cdot (-0.549) = -1.647$$

Nun können die Gewichte der Matrix $\Theta^{(1)}$ wie folgt aktualisiert werden:

$$
\begin{aligned}
\Theta_{10}^{(1)} &= 0.5 - 0.1 \cdot (-0.244) = 0.5244 \\
\Theta_{11}^{(1)} &= 0.3 - 0.1 \cdot 0.488 = 0.2512 \\
\Theta_{12}^{(1)} &= 0.6 - 0.1 \cdot (-0.732) = 0.6732 \\
\Theta_{20}^{(1)} &= 0.2 - 0.1 \cdot (-0.549) = 0.2549 \\
\Theta_{21}^{(1)} &= 0.8 - 0.1 \cdot 1.098 = 0.6902 \\
\Theta_{22}^{(1)} &= 0.1 - 0.1 \cdot (-1.647) = 0.2647
\end{aligned}
$$

In Abbildung 3.15 sind die Fehler an den Knoten in Rot, die aktualisierten Gewichte in Blau eingetragen.

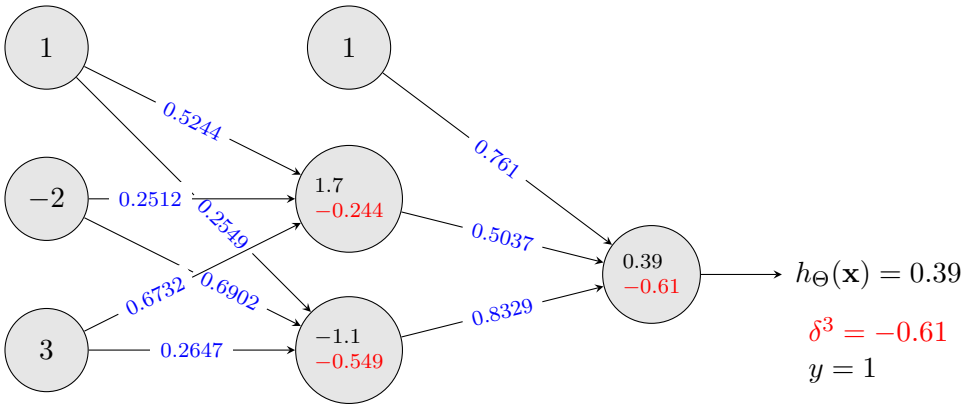



Abbildung 3.15.: Beispiel neuronales Netz, neue Gewichte nach Fehlerrückführung

Mit den aktualisierten Gewichtsmatrizen $\Theta^{(1)}$ und $\Theta^{(2)}$ kann nun erneut eine Vorwärtspropagation durchgeführt werden:

$$
\begin{aligned}
\mathbf{a}^{(2)} = \Theta^{(1)}\mathbf{x}, \quad &\text{also} \quad a_1^{(2)} = 0.5244 \cdot 1 + 0.2512 \cdot (-2) + 0.6732 \cdot 3 = 2.0416 \\
&\text{und} \quad a_2^{(2)} = 0.2549 \cdot 1 + 0.6902 \cdot (-2) + 0.2647 \cdot 3 = -0.3314 \\
\mathbf{a}^{(3)} = \Theta^{(2)}\mathbf{a}^{(2)}, \quad &\text{also} \quad a^{(3)} = 0.761 \cdot 1 + 0.5037 \cdot 2.0416 + 0.8329 \cdot (-0.3314) \approx 1.5133
\end{aligned}
$$

Der Fehler $\delta^3$ am Ausgabeknoten beträgt jetzt $1.5133 - 1 = 0.5133$, der Betrag hat also abgenommen. Abbildung 3.16 enthält die aktualisierten Werte.

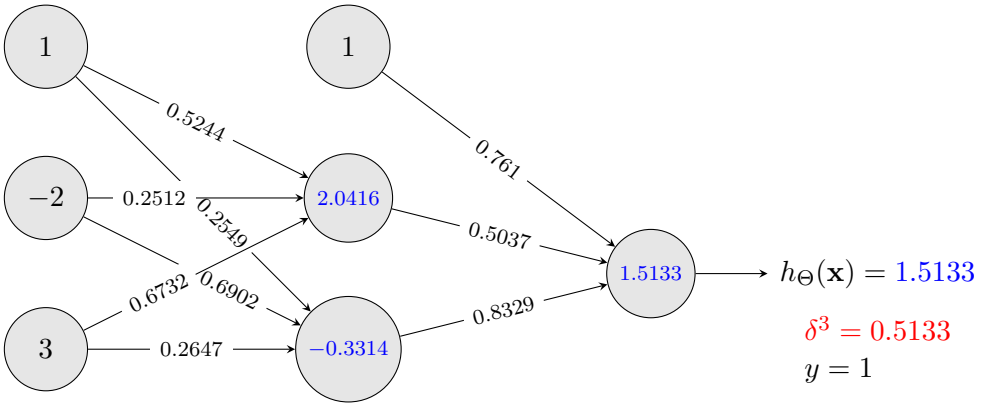



Abbildung 3.16.: Beispiel neuronales Netz, nach der zweiten Vorwärtspropagation

Nun müssten weitere Iterationen folgen. Die Prozedur wird so lange wiederholt, bis die Differenz der Verlustfunktion zweier aufeinanderfolgender Iterationen unter einen gewünschten Betrag $\epsilon$ sinkt. Bei $\epsilon = 0.001$ wäre dies hier die 20. Iteration.

Wie bereits erwähnt, handelt es sich um ein stark vereinfachtes Beispiel ohne logistische Funktion, ohne Regularisierung und mit einem einzigen Trainingsbeispiel. In realistischen Anwendungen werden die Fehler aller Beispiele aufaddiert und normiert. Der Gradientenabstieg wird auf Grundlage dieses normierten Fehlers durchgeführt. Die Verlustfunktion im angeführten Beispiel konvergiert gegen 0, wenn die Lernrate klein genug gewählt wird. Selbst ohne verdeckte Schicht würde sie dies tun. Mit den hier verwendeten Werten beträgt die Ausgabe nach der 20. Vorwärtspropagation ca. 1.0003, die Differenz der letzten beiden Fehlerwerte 0.0007. Doch das heißt nicht, dass die „richtigen“ Parameterwerte gefunden wurden. Hier gibt es sogar beliebig viele Matrizen, die zum Zielwert führen. Die Güte der Modellfunktion lässt sich nur an Validierungsdaten überprüfen.

#### 3.1.5.7. Feinabstimmung eines neuronalen Netzes

Ein großer Vorteil und gleichzeitig ein Nachteil neuronaler Netze ist ihre große Flexibilität. Es gibt viele Hyperparameter und andere Einstellungs- und Auswahlmöglichkeiten, die Einfluss auf die Qualität der Ergebnisse haben. Da all diese voneinander unabhängig sind, lassen sie sich beliebig miteinander kombinieren, und es ist unmöglich, alle Varianten zu testen. Daher werden in der folgenden Aufzählung einige Standardwerte oder Richtlinien genannt, die bei der Konfiguration eines Netzes helfen.

- Die **Topologie des Netzes**: Theoretisch genügt immer eine einzige verdeckte Schicht, die dann aber möglicherweise extrem viele Neuronen benötigt. Jedoch wäre diese Variante im Falle komplexer Funktionen ineffizient und nicht praktikabel, denn die Anzahl der benötigten Neuronen sinkt exponentiell mit der Anzahl der Schichten [8]. Dadurch lassen sich Netze mit mehr Schichten und entsprechend weniger Neuronen schneller trainieren. Je mehr Schichten es gibt, desto größer wird

allerdings auch die Gefahr der Überanpassung. Es empfiehlt sich daher für übliche Anwendungen, mit einer Schicht aus einigen Hundert Neuronen zu beginnen und die Anzahl der Schichten schrittweise zu erhöhen, bis eine Überanpassung an die Trainingsdaten erkennbar wird.

Die Anzahl der Neuronen in den verdeckten Schichten sollte ebenfalls nicht zufällig ausgewählt werden. In der Regel werden wesentlich mehr Eingabe- als Ausgabeneuronen benötigt. Diese Zahlen sind durch die Problemstellung festgelegt. Es kann nun sinnvoll sein, die Anzahl der Neuronen im Inneren mit jeder Schicht zu reduzieren. Intuitiv ist dies dadurch erklärbar, dass sich oft viele einfache Merkmale (z.B. die Form von Augen, Nase, Mund, Augenbrauen, ...) zu wenigen komplexen (einem Gesicht) zusammensetzen. Oft werden aber auch in allen Schichten gleich viele Neuronen verwendet. Dies hat den Vorteil, dass nur diese eine Anzahl optimiert werden muss anstatt pro Schicht ein Hyperparameter.

- Die **Lernrate** $\alpha$ bestimmt die Schrittweite beim Gradientenabstieg. Die notwendigen Maßnahmen zu ihrer Optimierung wurden in Abschnitt 3.1.1.2 im Zusammenhang mit linearer Regression erläutert. Bei neuronalen Netzen ist zusätzlich zu beachten, dass die Verlustfunktion nicht konvex ist und deshalb nicht direkt abgebrochen darf, wenn ihr Wert zwischendurch steigt. Die Beobachtung des Verlaufs muss über einen längeren Zeitraum stattfinden.

- Der **Hyperparameter** $\lambda$, der das Ausmaß der **Regularisierung** bestimmt, hat Einfluss auf die Varianz der Modellfunktion. Wie er angepasst wird, wurde bereits in Abschnitt 3.1.3.2 beschrieben. Dies kann so für neuronale Netze übernommen werden.

- Alternativ oder zusätzlich zur Regularisierung kann in neuronalen Netzen *Dropout* verwendet werden. Hierzu muss die **Dropout-Rate** eingestellt werden, die den Prozentsatz der Knoten festlegt, die in jeder Iteration des Backpropagation-Algorithmus auf einem einzelnen Stapel zufällig aus dem Netz entfernt werden. Da Dropout in der vorliegenden Implementierung eine wichtige Rolle spielt, wird in Abschnitt 3.1.5.10 die Idee dahinter genauer erläutert.

- Die **Initialisierung der Gewichte** entscheidet darüber, ob das globale Minimum der Verlustfunktion oder nur ein lokales Minimum gefunden wird. Wie bereits erwähnt, sind hier unbedingt mehrere Durchläufe erforderlich.

- Die **Verlustfunktion** wird zu Beginn für die gesamte Trainingsphase festgelegt. Für Regressionsprobleme wird in der Regel der *mittlere quadratische Fehler* (*MSE*) verwendet, für Klassifikationsprobleme je nach Anzahl der Klassen die *binäre* oder *kategorische Kreuzentropie*.

- die **Aktivierungsfunktion** kann für jede Schicht separat gewählt werden. Im Zuge der Implementierung werden in Abschnitt 4.4.2 einige Varianten vorgestellt.

- Die **Anzahl der Trainingsbeispiele**: Wenn alle anderen Parameter richtig eingestellt sind, so verbessert sich mit der Anzahl der Trainingsbeispiele die Genauigkeit der Klassifikation, bis sie irgendwann aufgrund des Rauschens der Trainingsdaten konvergiert. In den meisten realen Problemen ist eine Genauigkeit von 100 Prozent nicht erreichbar. (Beispielsweise sind manche handschriftliche Ziffern so unleserlich, dass nicht einmal der Schreiber selbst sie noch entziffern kann.) Es gibt jedoch auch Fälle, in denen trotz Verhundertfachung der Trainingsbeispiele keine Verbesserung mehr erzielt wird, obwohl noch Optimierungsmöglichkeiten bestehen. Dies kann z.B. daran liegen, dass ein *Underfitting* vorliegt, weil das Netzwerk zu wenige Schichten oder zu wenige Knoten in den Schichten verwendet.

- Die **Anzahl der Merkmale**: Im Allgemeinen wäre zu vermuten, dass mit der Anzahl der Merkmale auch die Genauigkeit der Klassifikation steigt. Dies stimmt jedoch nicht notwendigerweise. Liegt bei zu wenigen Trainingsbeispielen eine Überanpassung vor, so helfen zusätzliche Merkmale nichts. Im Gegenteil sollte deren Anzahl sogar nach Möglichkeit reduziert werden. Ist die Modellfunktion unterangepasst, so sollte der Versuch gestartet werden, neue Merkmale hinzuzufügen. Dies kann in der Praxis jedoch schwierig sein, wenn es sich um bereits vorhandene Datensätze handelt, z.B. um die Blutwerte von Millionen von Patienten.

Diese Auflistung soll zunächst als Überblick genügen. Am konkreten Beispiel in den Kapiteln 4 und 5 folgen praktische Anwendungen dieser Überlegungen.

#### 3.1.5.8. Konvolutionale neuronale Netze

*Konvolutionale neuronale Netze* (engl. *Convolutional Neural Networks, CNN*) stellen das Mittel der Wahl im maschinellen Sehen dar. Ein einfaches *CNN* besteht aus einer Reihe von abwechselnd angeordneten *Conv2D-* und *MaxPooling*-Layern. Dadurch wird das Eingabebild schrittweise zu sogenannten *Feature-Maps* (*Merkmalskarten*) transformiert, deren Höhe und Breite immer kleiner werden, die Anzahl der Kanäle jedoch immer größer. Die *Shape* des *Eingabetensors* wird der ersten *Conv2D*-Schicht als Argument mitgegeben. Der Begriff des *Tensors* verallgemeinert in diesem Zusammenhang Vektoren und Matrizen auf array-artige Objekte, die beliebig viele Achsen beliebiger Dimensionalität besitzen können. Die *Shape* definiert diese Form im konkreten Fall. Im maschinellen Sehen entspricht die erste Achse eines Datensatzes der Höhe des Bildes, die zweite der Breite. In der Musik könnte man dies auf die zur Verfügung stehenden Töne und die codierten Zeitpunkte übertragen, was dem Abscannen eines Notentextes nachempfunden wäre. Die Dimension der dritten Achse entspricht der Anzahl der Farbkanäle. In der Musik wäre diese Achse nur eindimensional und enthielte beispielsweise einen Wert, der angibt, ob ein Ton gerade angeschlagen wird, noch klingt oder nicht klingt. Die beiden erforderlichen Argumente aller *Conv2D*-Layer sind die Anzahl der auszugebenden Kanäle und die Größe des *Filterkerns*. In der Regel werden $3 \times 3$- oder $5 \times 5$-Filter verwendet. Die Anzahl der Ausgabekanäle steigt gewöhnlich mit jeder konvolutionalen Schicht. Die zwischengeschalteten *MaxPooling*-Layer sorgen für ein schnelleres *Down-Sampling*, denn sie wählen aus $2 \times 2$ oder $3 \times 3$ Einträgen der Matrix

jeweils den höchsten Wert aus. Damit halbieren oder dritteln sich jeweils die beiden ersten Dimensionen. Nach Abschluss der *Faltungs-* und *MaxPooling*-Operationen muss die *3D*-Ausgabe in einen *1D*-Vektor umgewandelt werden, um ihn zur Klassifikation in ein vollverbundenes Netz zu geben. [4]

Da sich recht schnell die Annahme bestätigte, dass ein *LSTM*-Netz bei der Klassifikation von *MIDI*-Dateien einem konvolutionalen Netz überlegen ist, sollen diese allgemeinen Angaben hier genügen. Jedoch gibt es auch *1D-konvolutionale Netze*, die sich besser für sequenzielle Daten eignen. Ein solches Netz wird im Zuge der Klassifikation einem *LSTM*-Netz vorgeschaltet, um zu sehen, ob dies die Ergebnisse verbessern kann.

#### 3.1.5.9. Rekurrente neuronale Netze

Die Bezeichnung „*rekurrent*“ bezieht sich zunächst nur darauf, dass Neuronen einer bestimmten Schicht nun auch Verbindungen zu Neuronen derselben oder einer vorangehenden Schicht besitzen können. Direkte Rückkopplungen verbinden den eigenen Ausgang eines Neurons wieder mit einem eigenen Eingang.

Die einfachste Form eines rekurrenten neuronalen Netzes (*RNN*) nimmt zu jedem Zeitpunkt $t$ einen Input-Vektor $\mathbf{x}_t$ entgegen, verarbeitet ihn zusammen mit dem aktuellen Status-Vektor $\mathbf{s}_t$ und leitet die Ausgabe $out_t$ als neuen Status wieder an den Eingang des Netzes. Abbildung 3.17, die einem Blog-Beitrag von Christopher Olah über *LSTM*-Netze entnommen wurde [18], veranschaulicht dies anhand eines „abgerollten“ Netzes. Mit „A“ wird also jeweils ein- und dasselbe Netz bezeichnet.

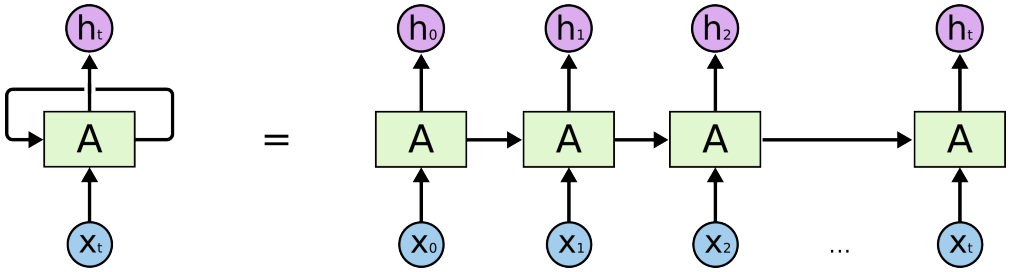



Abbildung 3.17.: links ein rekurrentes neuronales Netz, rechts dasselbe Netz abgerollt [18]

In jedem Verarbeitungsschritt kommen zwei unterschiedliche Gewichtsmatrizen zum Einsatz, deren Einträge mit Hilfe des Backpropagation-Algorithmus aktualisiert werden müssen: die erste ($\Theta_{\mathbf{x}}$) bestimmt die Gewichtung der Merkmale im Input-Vektor $\mathbf{x}_t$, die zweite ($\Theta_{\mathbf{s}}$) die Gewichtung der Merkmale im Status-Vektor $\mathbf{s}_t$. Der Output-Vektor $out_t$ berechnet sich dann wie folgt:

$$out_t = \text{sig}\left((\Theta_{\mathbf{x}} \cdot \mathbf{x}_t) + (\Theta_{\mathbf{s}} \cdot \mathbf{s}_t) + bias\right),$$

wobei die Sigmoid-Funktion durch jede andere Aktivierungsfunktion ausgetauscht werden kann. Der Status-Vektor zum Zeitpunkt $t = 0$ wird mit $\mathbf{0}$ initialisiert und ändert sich

in jedem Verarbeitungsschritt. Er stellt das Gedächtnis des Netzes dar, denn theoretisch enthält er zu jedem Zeitpunkt $t$ Informationen aus allen früheren Schritten. Dass in der Praxis dieses Gedächtnis nur über sehr kurze zeitliche Abstände funktioniert, demonstrierte 1994 ein Team um Yoshua Bengio anhand eines sehr einfachen Beispiels [3]. Sie verwenden ein rekurrentes Netz, um zwei Kategorien von Sequenzen zu klassifizieren, deren relevante Merkmale nur im vorderen Teil enthalten sind, während der hintere Teil bei der Klassifikation keine Rolle mehr spielt. Der Grund dafür, dass dies nur unzureichend gelingt, liegt in dem Problem der *explodierenden und verschwindenden Gradienten*. Diese entstehen während der Ausführung des „*Backpropagation through Time*"-Algorithmus (*BPTT*). Bei langen Sequenzen wird das abgerollte Netz sehr tief, was zur Folge hat, dass durch wiederholte Multiplikation von Werten kleiner bzw. größer als 1 verschwindend kleine oder aber extrem große Gradienten entstehen können. Im ersten Fall bedeutet dies, dass weiter zurückliegende Sequenzabschnitte praktisch keinen Einfluss mehr auf den Lernprozess haben. Im zweiten Fall springen die Gewichte mit jedem Lernschritt hin und her, so dass keine Konvergenz mehr zu erreichen ist.

### LSTM-Netze

Sepp Hochreiter und Jürgen Schmidhuber stellten bereits 1997 ein Modell vor, mit dessen Hilfe das Problem der verschwindenden Gradienten gelöst werden kann [12]. Es handelt sich um einen Spezialfall rekurrenter Netze, nämlich die *LSTM*-Netze. Das Akronym steht für *Long Short-Term Memory*. Informationen können so bei Bedarf auch über längere Zeiträume gespeichert werden. 1999 erweiterten Felix Gers und sein Team dieses Modell noch einmal um das *Forget Gate* [9]. Das große Potenzial der *LSTM*-Netze zeigte sich ab 2015, da erst zu dieser Zeit ausreichend große Datenmengen verarbeitet werden konnten. Ebenfalls 2015 veröffentlichte Christopher Olah, ein Mitarbeiter von *Google Brain*, den oben bereits erwähnten Blogbeitrag [18], der das Prinzip der drei *Gates* (*Forget Gate*, *Input Gate* und *Output Gate*) anhand zahlreicher Abbildungen verdeutlicht. Hieran lehnt sich die folgende Erläuterung an, und hier können auch die formalen Definitionen nachgelesen werden.

Abbildung 3.18 zeigt einen Ausschnitt aus einem abgerollten *LSTM*-Netz. Die grünen Einheiten bezeichnen auch hier jeweils ein- und denselben *LSTM*-Layer. Jedes gelbe Rechteck bezeichnet jedoch ebenfalls einen kompletten Layer mit Übertragungs- und Aktivierungsfunktion. *Ein* sogenannter *LSTM*-Layer beinhaltet somit tatsächlich *vier* Layer, die miteinander interagieren, und für die jeweils eine Gewichtsmatrix gelernt wird. Drei der vier Layer, nämlich die mit $\sigma$ markierten, bilden in Verbindung mit der dahinter geschalteten elementweisen Multiplikation jeweils ein *Gate*. Zu beachten ist zunächst, dass eine *LSTM*-Einheit in jedem Verarbeitungsschritt zwei Ausgaben besitzt. Die obere, waagerecht verlaufende Linie stellt den *Zellstatus* dar. Dieser wird im Verlaufe der Berechnung linear modifiziert, durch elementweise Multiplikation und Addition. Dadurch wird das oben beschriebene Problem der verschwindenden Gradienten gelöst. Der untere Pfeil auf der linken Seite entspricht dem Ausgabevektor $h_{t-1}$ des vorhergehenden Schrittes, der zu Beginn auch hier mit $\mathbf{0}$ initialisiert wird.

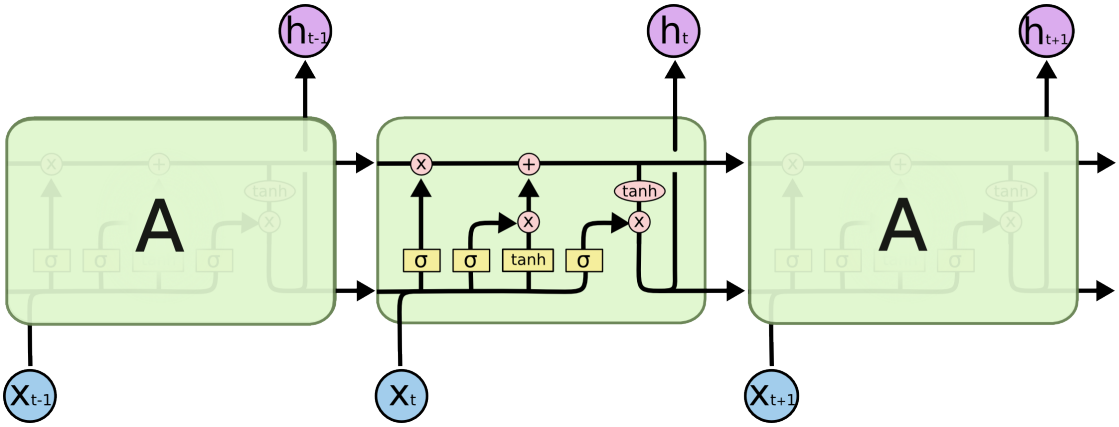



Abbildung 3.18.: Ausschnitt aus einem abgerollten LSTM-Netz [18]

Im ersten Schritt wird nun abhängig von $h_{t-1}$ und dem in diesem Schritt zu verarbeitenden Input $\mathbf{x}_t$ entschieden, welche Informationen aus dem Zellstatus entfernt („*vergessen*“) werden sollen. Daher wird dieser Teil des Netzes als *Forget Gate Layer* bezeichnet. Der Sigmoid-Layer gibt für jedes Element des Zellstatus eine Zahl zwischen 0 und 1 aus, wobei 0 bedeutet, dass alles vergessen wird, 1, dass alles behalten wird. Der Zellstatus wird mit diesen berechneten Werten elementweise multipliziert.

Im zweiten Schritt wird, wiederum abhängig von $\mathbf{x}_t$ und $h_{t-1}$, entschieden, welche neuen Informationen im Zellstatus gespeichert werden. Hierzu sind zwei Layer erforderlich. Der *Input Gate Layer* (wieder mit Sigmoid-Aktivierung) bestimmt zunächst, welche Werte in welchem Ausmaß aktualisiert werden sollen. Im Anschluss berechnet der *tanh*-Layer den Kandidatenvektor $\tilde{C}_t$, der neue zu speichernde Werte aus dem Wertebereich [-1, 1] enthält. Dieser wird noch einmal mit der zuvor berechneten Ausgabe des *Input Gates* elementweise multipliziert. Das Ergebnis wird zum Zellstatus addiert. Damit ist der neue Zellstatus vollständig bestimmt.

Der letzte Schritt dient dazu, die Ausgabe $h_t$ zu berechnen, die eine gefilterte Version des Zellstatus ist. Hierzu bestimmt zunächst ein *Sigmoid-Layer*, welche Werte des Zellstatus in welchem Maß ausgegeben werden sollen. Dann wird der Zellstatus mit Hilfe der *tanh*-Funktion in den Wertebereich zwischen -1 und +1 skaliert und das Ergebnis mit der Ausgabe des Sigmoid-Layers elementweise multipliziert. Zusammen bilden der Sigmoid-Layer in Verbindung mit der Multiplikation den *Output Gate Layer*, der das letzte der drei Gates realisiert.

Zur Feinabstimmung von *LSTM*-Netzen gelten zunächst ähnliche Prinzipien wie die in Abschnitt 3.1.5.7 dargestellten. Zusätzlich finden sich nützliche Ideen in einem Artikel eines Forschungsteams um Rafal Jozefowicz [13], in dem Tests Tausender von *RNN*-Architekturen ausgewertet werden. Da insbesondere *MIDI*-Dateien in ein *LSTM*-Netz eingegeben wurden, werden einige dieser Vorschläge in der vorliegenden Anwendung umgesetzt.

#### Bidirektionale Netze

Ein rekurrentes Netz kann *unidirektional* oder *bidirektional* arbeiten. In einem bidirektionalen Netz wird die Anzahl der Einheiten implizit verdoppelt, wobei eine Hälfte des Netzes die Sequenz in umgekehrter Richtung durchläuft. So wie sich typischerweise in der deutschen Sprache der Sinn eines Satzes manchmal erst mit dem letzten Wort erschließt, gibt es entsprechende Phänomene auch in der Musik. Dissonanzen müssen in der Regel zwingend irgendwann aufgelöst werden. Passiert dies nicht, so handelt es sich um ein Stück des 20. Jahrhunderts. Manchmal sind also Informationen „aus der Zukunft“ notwendig, um die Gegenwart zu verstehen. Ein bidirektionales Netz kann hier hilfreich sein.

### 3.1.5.10. Regularisierung vs. Dropout in neuronalen Netzwerken

Wie bereits in Abschnitt 3.1.5.4 erwähnt, stammt die Idee des *Dropout* von Geoffrey Hinton und seinem Team [25]. Es handelt sich um ein verblüffend einfaches Verfahren, bei dem in jedem Lernschritt jeder Knoten mit einer im Voraus festzulegenden Wahrscheinlichkeit mitsamt seiner ein- und ausgehenden Verbindungen aus dem Netz herausgenommen wird. Ein Lernschritt bezeichnet dabei die Ausführung der Vorwärtspropagation und der Fehlerrückführung auf einem einzelnen *Mini-batch*. Die daraus resultierende Korrektur der Gewichte kann infolgedessen auch nur die in diesem Schritt vorhandenen Kanten berücksichtigen. Bei $n$ Knoten im Netz gibt es $2^n$ mögliche Teilmengen. Nur auf solchen Teilmengen findet das Training statt. Um die Teilmengen in der Validierungs- und Testphase wieder vereinen zu können, wird dann jedes Gewicht mit dem Prozentsatz multipliziert, welcher die Wahrscheinlichkeit beschreibt, mit der diese Kante während eines einzelnen Trainingsschrittes im Netz vorhanden war. Auch in rekurrenten Netzen wurden bereits große Erfolge durch Dropout erzielt, was aus einer Veröffentlichung von Yarin Gal und Zoubin Ghahramani aus dem Jahre 2016 hervorgeht [7].

Die Berechnungen im Zuge der Verwendung von Dropout sind effizienter durchzuführen als die entsprechenden Berechnungen bei Verwendung von Regularisierungstermen. Zudem zeigt sich in der Praxis, dass die Dropoutrate einfacher einzustellen ist und bessere Ergebnisse liefert. Hinton schlägt vor, mit einer Rate von 0.5 zu beginnen und bei Bedarf nach oben oder unten zu korrigieren [25].

## 3.1.6. Andere Methoden des überwachten Lernens

Es gibt zahlreiche andere Methoden des überwachten Lernens, die außer der Tatsache, dass sie mit gekennzeichneten Trainingsdaten arbeiten, wenig mit neuronalen Netzwerken gemeinsam haben. Daher werden hier nur exemplarisch zwei der bekanntesten Verfahren erwähnt und kurz deren Prinzipien erläutert.

#### Support Vector Machines

*Support Vector Machines* gehören zu den *Large Margin Classifiers* („*Breiter-Rand-Klassifizierern*“), weil sie die Entscheidungsgrenzen zwischen Objekten verschiedener Ka-

tegorien so ziehen, dass um sie herum ein möglichst breiter Rand frei von Objekten bleibt. Dies bringt zwei große Vorteile mit sich, nämlich zum einen eine besonders gute Verallgemeinerungsfähigkeit und zum anderen effiziente Berechnungsmöglichkeiten, da nur die nächstliegenden Trainingsbeispiele die Lage der Grenze beeinflussen. Grundsätzlich wird nach einer *linearen* Grenze gesucht. Dies wäre im zweidimensionalen Raum eine Gerade, im $n$-dimensionalen Raum allgemein eine *Hyperebene* der Dimension $n-1$. Lassen sich die Daten so nicht separieren, weil sie verrauscht sind oder die Kategorien sich natürlicherweise überlappen, so können Objekte auf der „falschen Seite“ der Grenze liegen, werden dann aber mit „Strafen“ belegt, die es zu minimieren gilt. Handelt es sich aber tatsächlich um ein nichtlineares Problem, so kann mit Hilfe des sogenannten *Kernel-Tricks* dennoch eine Grenze gefunden werden. Die Daten werden dann in einen höherdimensionalen Raum überführt, in dem eine lineare Separation möglich ist. Bei der Rücktransformation entsteht eine nichtlineare Fläche, welche die Daten trennt. [20]

Die grüne, durchgezogene Gerade in Abbildung 3.19 veranschaulicht die Idee der *SVM*. Sie hat den größtmöglichen Abstand zum nächsten Beispiel beider Klassen. Die schwarze Gerade klassifiziert die Trainingsbeispiele ebenfalls korrekt, hat jedoch nur einen sehr kleinen Abstand zu den nächsten Beispielen.

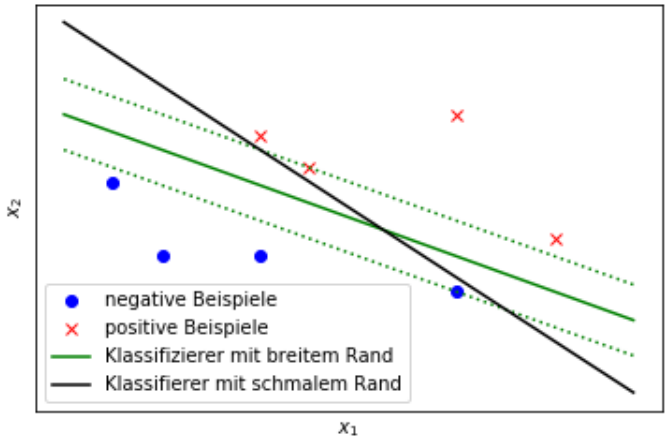



Abbildung 3.19.: Zwei Klassifizierer, einer mit breitem, einer mit schmalem Rand

### Entscheidungsbäume

Das Lernen von Entscheidungsbäumen ist ein in der Praxis sehr erfolgreiches und eines der intuitivsten Verfahren zur Klassifikation von Objekten. Ein Objekt kann ein Gegenstand, aber auch etwas Abstraktes wie ein Krankheitsbild, eine Situation oder auch ein Musikstück sein. Es wird durch eine Menge von Attribut-/Wertpaaren beschrieben. Die Wurzel und die inneren Knoten des Baums sind mit jeweils einem Attribut markiert. Sie repräsentieren damit eine Abfrage nach dem Wert dieses Attributes. Die von solch einem Knoten ausgehenden Kanten sind mit den möglichen Attributwerten markiert, die Blätter des Baums mit den Kategorien der Klassifikation. Abbildung 3.20 zeigt einen

sehr stark vereinfachten Entscheidungsbaum zur Klassifikation von Musikstücken. Soll nun ein Objekt einer Kategorie zugeordnet werden, so werden ausgehend vom Wurzelknoten die Attribute abgefragt, bis ein Blattknoten erreicht ist. Der diesem Blattknoten zugeordnete Wert entspricht der Kategorie, der das Objekt angehört.

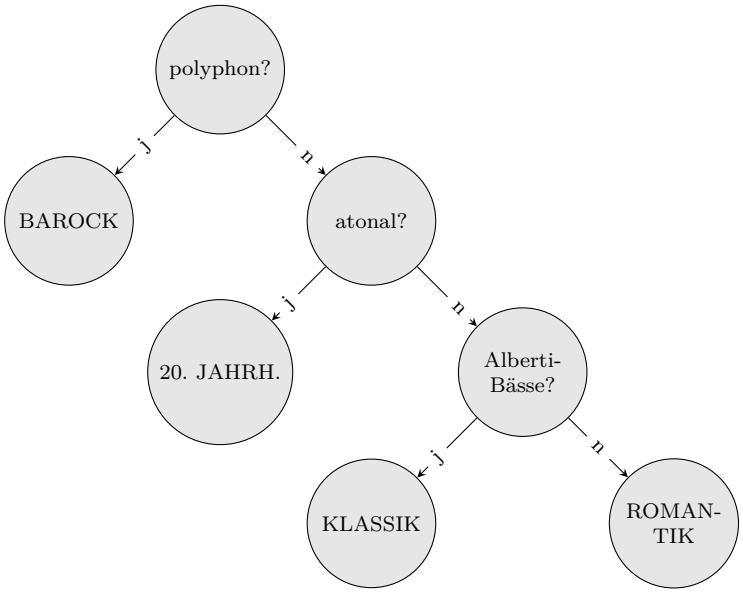



Abbildung 3.20.: Ein stark vereinfachter Entscheidungsbaum

Beim maschinellen Lernen geht es darum, solch einen Entscheidungsbaum aus einer Menge von gekennzeichneten Beispielen automatisch zu generieren. Prinzipiell ist es sehr einfach, *irgendeinen* Baum zu konstruieren, der alle Trainingsbeispiele richtig zuordnet. Es müsste lediglich für jedes Beispiel ein kompletter Pfad existieren, der alle Attribute abfragt. Solch ein Baum könnte allerdings nicht verallgemeinern, er wäre überangepasst. Gesucht ist also ein möglichst *kompakter* Baum, oder anders ausgedrückt, die „*einfachste Hypothese, die konsistent mit allen Beobachtungen ist*“ [2]. Dieses Prinzip entspricht dem heuristischen *Sparsamkeitsprinzip*, das unter der Bezeichnung *Ockham's Rasiermesser* bekannt ist. Der rekursive *ID3*-Algorithmus realisiert eine Heuristik, die in der Regel solch einen kompakten Baum findet. Bei der Wahl des Attributes, das den nächsten Knoten markieren soll, wird immer das ausgewählt, für welches der höchste Informationsgewinn erwartet wird. Der Begriff der *Entropie* aus der Informationstheorie spielt hierbei eine wichtige Rolle, soll aber hier nicht näher erläutert werden. [2]

## 3.2. Unüberwachtes Lernen

Beim unüberwachten Lernen sind die Trainingsdaten nicht gekennzeichnet. Es gibt keinerlei Vorgaben, mit Ausnahme der Anzahl der Kategorien bzw. Cluster, die in einigen Algorithmen im Voraus festgelegt werden muss. Aber auch dies lässt sich automatisieren. Aufgabe eines Systems ist es ganz allgemein, Muster bzw. Strukturen irgendeiner Art in den Daten aufzuspüren und die Daten auf dieser Basis zu gruppieren. Neben den *Clustering*-Algorithmen wird im Folgenden auch das *Lernen von Assoziationsregeln* kurz betrachtet.

### Clustering

Clustering-Algorithmen dienen der Aufdeckung von Ähnlichkeitsstrukturen in sehr großen Datensätzen, insbesondere dann, wenn im Vorfeld nichts über die Struktur der Klassen bekannt ist. Lediglich die Anzahl der zu findenden Cluster muss in der Mehrzahl der Fälle angegeben werden. Aufgabe eines Clustering-Algorithmus ist es, die gegebenen Datenpunkte so in Cluster zu gruppieren, dass die Punkte innerhalb eines Clusters sich ähnlicher sind als Punkte aus verschiedenen Clustern. Dazu ist zunächst ein geeignetes Ähnlichkeits- oder Distanzmaß notwendig.

Es gibt viele verschiedene Clustering-Algorithmen, die meisten gehören zu den *partionierenden* oder den *hierarchischen* Verfahren. Um das Prinzip des Clustering zu verdeutlichen, sei hier der bekannteste Algorithmus, *k-Means-Clustering*, in der Version von Lloyd (1957) skizziert, der zu den *partitionierenden* Verfahren gehört.

Der Algorithmus bekommt als Eingabe $m$ Datenpunkte und eine Zahl $k$ zu findender Cluster, mit $k < m$. Im ersten Schritt werden $k$ Clusterzentren zufällig aus dem Datensatz ausgewählt. Im zweiten Schritt wird jeder Datenpunkt dem Clusterzentrum zugeordnet, zu dem die Distanz am geringsten ist. Im dritten Schritt werden schließlich die Clusterzentren neu berechnet, nämlich als Mittelwert der dem Cluster zugehörigen Punkte. Die Schritte zwei und drei werden wiederholt, bis sich die Zuordnungen nicht mehr ändern.

Es gibt Weiterentwicklungen und Variationen des Verfahrens. So versucht *k-Means++*, die Clusterzentren günstig zu initialisieren, *X-Means* automatisiert die Bestimmung der Anzahl $k$ von Clustern. Der Algorithmus realisiert eine Heuristik und findet nicht notwendigerweise die beste Lösung, da diese entscheidend von der Initialisierung abhängt. In der Praxis wird er mehrmals ausgeführt, um zu einem guten Ergebnis zu kommen.

Die *hierarchischen* Verfahren, die die zweite große Gruppe der Clustering-Algorithmen bilden, gliedern sich wiederum in die *agglomerativen* (*Bottom-up*) und die *divisiven* (*Top-down*) Algorithmen. Die agglomerativen Algorithmen beginnen mit $m$ Clustern, die schrittweise immer weiter zusammengefasst werden, die divisiven beginnen mit einem einzigen Cluster, der schrittweise geteilt wird. Beides führt schließlich zu einer hierarchischen Ordnung der Elemente, wobei in der Regel Experten beurteilen müssen, welche Ebenen der Hierarchien sinnvoll sind. Dies bedeutet einen hohen Analyseaufwand. Auch

die Komplexität der Algorithmen selbst ist vergleichsweise hoch. Das Ergebnis der Analyse kann schließlich als Baumdiagramm veranschaulicht werden, das von der Wurzel bis zu den Blättern die ursprüngliche Menge von Objekten schrittweise in kleinere Partitionen teilt.

### Lernen von Assoziationsregeln

Das Lernen von Assoziationsregeln zählt ebenfalls zu den unüberwachten Lernalgorithmen, hat aber nur wenige Gemeinsamkeiten zum Clustering. *Assoziationen* sind Zusammenhänge zwischen mehreren Merkmalen, *Assoziationsregeln* beschreiben Zusammenhänge zwischen verschiedenen Objekten. Typische Anwendungsfälle sind die Warenkorbanalyse, medizinische Diagnose oder auch die Aufdeckung von Zusammenhängen zwischen bestimmten Umweltgiften und Krankheiten. Wird das Verfahren konsequent angewendet, so können zufällige Korrelationen mit großer Sicherheit von Kausalitäten unterschieden werden. Ein im Januar 2019 aktuelles Beispiel ist die Debatte darüber, ob die von Dieselmotoren ausgestoßenen Stickstoffdioxide verantwortlich für bestimmte Lungenerkrankungen sind, oder ob die in Großstädten häufiger auftretenden Erkrankungen eher auf andere Umweltgifte oder Bewegungsmangel zurückzuführen sind. Der *Apriori-Algorithmus* sucht Regeln der Art „*90% der Kunden, die Sahne und Butter kaufen, kaufen auch Milch.*“ oder „*25% der Einwohner deutscher Großstädte leiden an Atemwegserkrankungen.*“. Entscheidend ist, dass die Regeln zwei Bedingungen erfüllen: Sie müssen einen festzulegenden Mindestwert für den *Support* (die „*Unterstützung*“) erreichen, d.h. ein bestimmter Prozentsatz der Datensätze in der Datenbasis muss beide Seiten der Regel enthalten. Und sie muss einen Mindestwert für die *Konfidenz* (das „*Vertrauen*“) erreichen. Dieser Wert bezeichnet den Anteil der Datensätze, der beide Seiten der Regel enthält, an den Datensätzen, die mindestens den linken Teil enthalten. Anschaulich bedeutet das am Beispiel der Großstädte, dass zum einen ein bestimmter Prozentsatz der Datenbasis Menschen bezeichnen muss, die in einer Großstadt wohnen *und* an einer Atemwegserkrankung leiden, und dass nicht zu viele Datensätze Großstadtbewohner bezeichnen, die *nicht* daran erkrankt sind.

Der Algorithmus macht es sich zu Nutze, dass alle Teilmengen einer „*häufigen Menge*“ ebenfalls häufig sind, und umgekehrt alle Obermengen einer nicht-häufigen Menge ebenfalls nicht häufig sind. So können zunächst alle einelementigen Mengen auf ihren Support untersucht werden, anschließend die zweielementigen Obermengen der häufigen einelementigen Mengen usw., bis keine der gefundenen Obermengen mehr den Mindestsupport erreicht. Nachdem diese *häufigen Mengen* bestimmt wurden, werden daraus die gesuchten *Assoziationsregeln* erstellt. Auch hier kann wieder eine Teilmengenbeziehung ausgenutzt werden: Zunächst werden die Regeln mit einelementiger Konklusion gebildet und auf ihre Konfidenz hin überprüft. Liegt eine Regel nicht über der Schwelle, so wird sie aus der Regelmenge entfernt. Sobald alle gültigen Regeln mit einelementiger Konklusion bestimmt wurden, werden zweielementige Obermengen der Konklusionen gebildet, die in den häufigen Mengen enthalten sind, und die entsprechenden Regeln wieder auf ihre Konfidenz hin überprüft. Dies wird wiederum so fortgeführt, bis keine Regel einer Ebene mehr die Schwelle überschreitet. [2]

## 3.3. Reinforcement Learning

Das *Reinforcement Learning* (*verstärkendes Lernen*) kann weder dem überwachten noch dem unüberwachten Lernen zugeordnet werden, da es keine Beispiele dafür gibt, was ein Agent zu tun hat. Allgemein formuliert besitzt ein Agent *Sensoren*, um seine Umgebung wahrzunehmen, und *Aktuatoren*, durch die er handeln kann. Betrachtet man einen Menschen als Agenten, so sind seine Sinnesorgane die Sensoren, die Hände, Füße, der Mund usw. die Aktuatoren. Ein Roboter-Agent kann durch Kameras und Mikrofone visuelle und akustische Signale aufnehmen, er kann Sensoren zum Tasten besitzen usw. Seine Aktuatoren können beispielsweise Räder oder Arme und Beine sein. Es gibt auch Software-Agenten, die auf Tastatureingaben oder Dateiinhalte reagieren, indem sie etwas auf dem Bildschirm ausgeben. Der Begriff *Agent* ist also sehr weit gefasst.

Das Verhalten eines Agenten wird durch die *Agentenfunktion* bestimmt, die es im Falle von *intelligenten* Agenten, um die es beim *Reinforcement Learning* geht, zu optimieren gilt. In jedem Falle hängt sie von den bisherigen Wahrnehmungen ab. Entscheidend ist, dass der Agent irgendeine Form von *Feedback* bekommt. Er lernt dann mit der Zeit für immer mehr Situationen, durch welche Handlung viele Belohnungen, wenige Bestrafungen oder am Ende ein bestimmter *Zielzustand* zu erwarten sind. Das Feedback muss fest einprogrammiert sein und kann entweder nach jeder Handlung oder erst am Ende einer Aufgabe erfolgen. Prominente Beispiele für intelligente Agenten sind Schach- oder Go-spielende Computerprogramme und das 2015 eröffnete japanische Hotel *Henn'na*, in dem Roboter einen Großteil der Aufgaben übernehmen sollten.

Obwohl *Reinforcement Learning* zunächst auf einem anderen Konzept beruht als das Lernen aus Beispielen, können intelligente Agenten durchaus intern neuronale Netze verwenden. So werden beim autonomen Fahren konvolutionale Netze benötigt, um Autos, Menschen, Brücken usw. erkennen zu können. Zudem entstehen während des Lernprozesses eines Agenten immer mehr Beispiele für Situationen, in denen bestimmte Handlungen zu einem „*Gut*“ oder „*Schlecht*“ geführt haben. Mit diesen Informationen können neuronale Netze trainiert werden. Auch unüberwachte Lerntechniken können im Agentenprogramm integriert sein. Insofern stehen die drei großen Bereiche des maschinellen Lernens nicht isoliert voneinander, sondern werden in Zukunft immer mehr ineinander übergreifen. [20]

# 4. Implementierung des neuronalen Netzes zur Klassifikation klassischer Musik

## 4.1. System, Programmiersprache, Entwicklungsumgebung und Bibliotheken

Zur Entwicklung des Prototyps zur Klassifikation von klassischer Musik in der Programmiersprache *Python* wurde die interaktive Umgebung *Jupyter Notebook* verwendet. Das Training wurde auf einer *Intel Core i5*-CPU ausgeführt. Die Verwendung einer zum maschinellen Lernen geeigneten Grafikkarte würde die Laufzeit um ein Vielfaches verringern, was die Möglichkeit bieten würde, mit kleineren Lernraten und mehr Epochen bessere Ergebnisse zu erzielen. Da dies zur Verdeutlichung der Prinzipien und zur Ermittlung des am besten geeigneten Netzes nicht erforderlich ist, wurde darauf verzichtet. So sind alle Jupyter-Notebooks auf gängigen Rechnern in akzaptabler Zeit ausführbar.

*Python* bietet eine Reihe von frei zugänglichen Bibliotheken zur effizienten Datenverarbeitung und zum maschinellen Lernen. Folgende finden Verwendung:

- *Mido* dient zum Parsen von *MIDI*-Dateien und Extrahieren der Merkmale.
- *NumPy* steht für *Numerical Python* und enthält Werkzeuge zu wissenschaftlichen Berechnungen, insbesondere zur Verarbeitung von Matrizen und Vektoren. Die zentrale Datenstruktur ist das *n-dimensionale Array*.
- *Matplotlib* dient zur Visualisierung von Funktionen und statistischen Zusammenhängen.
- *Scikit-Learn* beinhaltet zahlreiche Algorithmen des maschinellen Lernens, darunter solche zur Klassifikation, Regression und zum Clustering, zur Dimensionsreduktion sowie zur Analyse der Daten und Auswahl des besten Modells. Die Funktionalität baut auf *NumPy*, *SciPy* und *Matplotlib* auf. Die ersten neuronalen Netze zur Klassifikation der *MIDI*-Dateien wurden mit *Scikit-Learn* erstellt. Da rekurrente Netze jedoch nicht unterstützt werden, wurde der Wechsel zu *Keras* notwendig. Zur Analyse wird nach wie vor *Scikit-Learn* verwendet.
- *Keras* ist eine High-Level Bibliothek speziell zur Arbeit mit neuronalen Netzen. Unterstützt werden alle Deep-Learning-Modelle, vom *vollverbundenen Feedforward-Netz* bis hin zu *konvolutionalen* und *rekurrenten* Netzen sowie deren Kombination. Keras benötigt ein *Backend*, auf dem es aufbaut. Zur Wahl stehen hier *TensorFlow*, *Theano* und *CNTK*. In dieser Thesis fiel die Wahl auf *TensorFlow*, was die Verwendung von *Python 3* erforderlich machte.

- *TensorFlow* ist die zur Zeit populärste Open-Source-Bibliothek für künstliche Intelligenz, und sie steht auch für andere Programmiersprachen zur Verfügung. Entstanden ist sie im Rahmen des *Google Brain*-Projekts. Viele Google-Dienste bauen darauf auf. Insbesondere können Berechnungen auf viele Hundert Server verteilt werden. Bei Installation des Pakets kann auf Wunsch die grafikprozessorfähige Version gewählt werden, was die Rechenzeit erheblich verkürzt. Jedoch ist dazu eine zum maschinellen Lernen geeignete Grafikkarte erforderlich.

## 4.2. Datenerhebung und Aufbereitung der Daten

### 4.2.1. Verwendete Musikdateien im MIDI-Format

Zum Trainieren und Testen der Modelle wurden *MIDI*-Dateien von folgenden Webseiten heruntergeladen:

- `http://www.piano-midi.de/`: Diese Dateien sind OpenSource, die Lizenz liegt bei Bernd Krüger unter `http://creativecommons.org/licenses/by-sa/3.0/de/deed.en`.
- `http://www.kunstderfuge.com/`: Hier stehen 17 326 *MIDI*-Dateien zum Download, doch dieser ist kostenpflichtig (einmalig 40,-€ für einen *Academy-Account*).
- `www.classic-arietta.de/`: Hier steht eine begrenzte Auswahl von Werken zum freien Download zur Verfügung. Diese sind von besonders hoher Qualität, deshalb wurden die entsprechenden Stücke aus anderen Quellen durch diese ersetzt.
- `http://www.bach-js.net/`: Diese Seite bietet *MIDI*-Dateien fast aller Stücke von Johann Sebastian Bach.
- `http://www.klausmeier.de/midi/klmidi.htm`: Hier stehen relativ viele Dateien kostenlos zum Download zur Verfügung.
- `http://tirolmusic.blogspot.com/`: Auf dieser Seite ist eine zwar recht begrenzte, aber hochwertige Auswahl an *MIDI*-Dateien zu finden.
- `http://www.curtisclark.org/emusic/`: Dies ist eine spezielle Seite für alte Musik. Es können 249 Dateien von Stücken aus dem Mittelalter und der Renaissance heruntergeladen werden. Vor die Dateinamen wurde jeweils „*ma-re*“ (für „*Mittelalter/Renaissance*“ gesetzt, denn eine Klassifikation nach Komponist ist hier ohnehin kaum möglich.
- `http://www.trachtman.org/ragtime/`: Hierbei handelt es sich um eine Seite, auf der es eine große Auswahl von Ragtimes verschiedener Komponisten gibt. Alle Stichproben sind von sehr guter Qualität.
- `http://www.midiworld.com/classic.htm`: Auch hier ist kostenloser Download möglich, jedoch sind die Stücke von gemischter Qualität.

- `http://www.klavier-noten.com/`: Auf dieser Seite steht eine sehr große Auswahl an *MIDI*-Dateien klassischer Musik zur Verfügung, fast durchweg von hoher Qualität.
- `https://www.classicalarchives.com/`: Hier können mit einem freien Account bis zu fünf *MIDI*-Dateien pro Tag heruntergeladen werden. Dies wurde nur sporadisch genutzt, um einiges zu ergänzen.

Um zu gewährleisten, dass jedes Stück nur einmal in der vereinigten Menge von Trainings-, Validierungs- und Testdaten enthalten ist, wurden die Dateinamen einheitlich so modifiziert, dass sie mit dem Komponistennamen beginnen und der jeweilige Titel erkennbar ist. Dadurch ergab sich auch die Möglichkeit, die falsch klassifizierten Stücke zu erfassen, um die Ursachen für das Scheitern zu ergründen. Vorhandene Kürzel für das Copyright wurden selbstverständlich unverändert übernommen. Im übrigen sind alle Original-Dateien unter ihrem ursprünglichen Dateinamen so gesichert, dass erkennbar ist, aus welcher Quelle sie stammen.

### 4.2.2. Zusammenstellung der Datenmengen

Feinabstimmung und Training eines neuronalen Netzes sind sehr zeitaufwändig und benötigen viel Speicherplatz. Da sich zudem recht früh herauskristallisierte, dass bei fünf Kategorien kein wesentlich besseres Ergebnis als 90% Trefferquote zu erzielen ist, fiel die Entscheidung, sich zunächst auf fünf Kategorien zu beschränken, was dann prinzipiell der Klassifikation nach Epoche gleichkommt. Aus der Zeit des Mittelalters und der Renaissance gibt es viele Komponisten, von denen aber jeweils nur wenige Werke überliefert sind und noch weniger Werke im *MIDI*-Format zur Verfügung stehen. Daher wurden diese Werke zu einer Kategorie zusammengefasst. Die vollständigsten Sammlungen von *MIDI*-Dateien gibt es von Komponisten der Barockzeit. Um die Überrepräsentation dieser Epoche zu vermeiden, wurde zunächst Georg Friedrich Händel als Repräsentant ausgewählt. Als Komponist der Wiener Klassik wurde Joseph Haydn gewählt, als Komponist der Romantik Frédéric Chopin. Hauptauswahlkriterien waren hier wiederum Anzahl und Qualität der vorhandenen *MIDI*-Dateien. Die Komponisten des 20. und 21. Jahrhunderts unterscheiden sich zum Teil erheblich voneinander. Zudem gibt es keinen, von dem ausreichend viele Werke im *MIDI*-Format zur Verfügung stehen. Daher wurde hier als Unterkategorie die Gattung *Ragtime* ausgewählt.

Obwohl ausschließlich die Trainingsdaten in die Berechnung des Gradientenabstiegs einfließen, besteht die Gefahr, dass im Laufe der Zeit die Topologie des Netzes und die Einstellung der Hyperparameter auf die Validierungsdaten hin optimiert werden. Aus diesem Grund ist es üblich, eine dritte Datenmenge bereit zu halten, die erst dann zum Einsatz kommt, wenn ein Netz zur Verfügung steht, das auf den Trainings- und Valdierungsdaten ein akzeptables Ergebnis liefert [17]. Da jedoch in der ersten Phase die beste Kombination aus Netztyp und Merkmalscodierung gesucht wird, um diese in der zweiten Phase weiter zu optimieren, wird in Phase eins, die ab hier als *Netzauswahlphase* bezeichnet wird, auf den Einsatz einer Testmenge verzichtet. Da bis zu deren Abschluss

mehr Daten erhoben werden konnten, bot sich die Möglichkeit, für die Optimierungsphase eine neue Datenmenge mit teilweise ausgetauschten Komponisten zusammenzustellen, und diese dann auf drei verschiedene Ordner zu verteilen.

Diese Aufteilung wurde in allen Fällen von Hand derart vorgenommen, dass in den zwei bzw. später drei Mengen die Anzahl der Beispiele aus den verschiedenen Kategorien jeweils identisch ist. Ein weiterer wichtiger Grund für die Aufteilung von Hand ist der, dass ausschließlich die Trainingsdaten durch Transponieren in alle Tonarten verzwölffacht werden. Würde der Split automatisch nach dem Einlesen durchgeführt, so wäre dasselbe Stück in verschiedenen Tonarten in Trainings- und Testdaten enthalten. Um dies zu vermeiden, werden die Dateien aus den verschiedenen Ordnern getrennt voneinander eingelesen.

Folgende Datenmengen wurden zusammengestellt:

- **Datenset 1** für die **Netzauswahlphase** enthält jeweils 140 Stücke
  - aus der Zeit des **Mittelalters** und der **Renaissance**
  - aus der **Barockzeit**, vertreten durch Georg Friedrich Händel
  - aus der **Wiener Klassik**, vertreten durch Joseph Haydn
  - aus der **Romantik**, vertreten durch Frédéric Chopin
  - aus der Gattung des **Ragtime**

  Von diesen jeweils 140 Stücken werden 105 zum Training verwendet, die übrigen zur Validierung.

- **Datenset 2** für die **Optimierungsphase** enthält jeweils 165 Stücke aus denselben Epochen, wobei die Barockzeit hier durch Johann Sebastian Bach repräsentiert wird und die Romantik durch Robert Schumann. Von 165 Stücken dienen je 105 zum Training, 30 zur Validierung während des Trainings und 30 zum Testen des konfigurierten Netzes.

Nach Abschluss der Optimierungsphase wird das dann zur Verfügung stehende Netz dazu verwendet, auf vier weiteren Datenmengen zu trainieren. Zunächst sollen die drei Hauptvertreter der barocken Klaviermusik (Johann Sebastian Bach, Georg Friedrich Händel und Domenico Scarlatti) voneinander unterschieden werden. Anschließend soll eine größere Anzahl von Komponisten, von denen jeweils zu wenige Stücke im *MIDI*-Format vorhanden sind, um sie getrennt klassifizieren zu können, jeweils ihrer Epoche zugeordnet werden. Dann wird versuchsweise eine Klassifikation nach Komponist bei 15 Kategorien durchgeführt, und zuletzt wieder eine Klassifikation nach Epoche, wobei alle bis dahin erkannten Fehlerquellen beseitigt werden. Die benötigten Datenmengen sind folgende:

- **Datenset 3** für die **Klassifikation der drei Barock-Komponisten** enthält jeweils 154 Stücke von **Johann Sebastian Bach**, **Georg Friedrich Händel** und **Domenico Scarlatti**. Davon werden jeweils 104 zum Training verwendet, 25 zur Validierung und 25 zum Test.

- **Datenset 4** für die **Klassifikation verschiedener Komponisten nach sechs Epochen** enthält jeweils 165 Stücke
  - aus dem Zeitalter des **Mittelalters** und der **Renaissance**
  - aus der **Barockzeit**, vertreten durch Bach, Händel und Scarlatti
  - aus der **Wiener Klassik**, vertreten durch Haydn, Mozart und Beethoven
  - aus der **Romantik**, vertreten durch Schubert, Schumann, Mendelssohn, Chopin, Liszt, Brahms, Grieg und Tschaikowski
  - aus dem **20. Jahrhundert**, vertreten durch Bartók und Scriabin
  - aus der Kategorie des **Ragtime**, vertreten durch diverse Komponisten

  Von diesen jeweils 165 Stücken werden 105 zum Training verwendet, 30 zur Validierung und 30 zum Test.

- **Datenset 5** für die **Klassifikation nach Komponist bei 15 Kategorien** enthält jeweils 55 Stücke
  - aus dem Zeitalter des **Mittelalters** und der **Renaissance**
  - von **Johann Sebastian Bach**, **Georg Friedrich Händel** und **Domenico Scarlatti**,
  - von **Joseph Haydn**, **Muzio Clementi**, **Wolfgang Amadeus Mozart** und **Ludwig van Beethoven**
  - von **Franz Schubert**, **Robert Schumann**, **Frédéric Chopin** und **Johannes Brahms**,
  - aus der Kategorie des **Ragtime**
  - von **Béla Bartók** und **Alexander Scriabin**

  Von diesen jeweils 55 Stücken werden 40 zum Training verwendet und 15 zur Validierung. Auf eine Testmenge wird verzichtet.

- **Datenset 6** für die **Klassifikation verschiedener Komponisten nach fünf Epochen** ähnelt Datenset 4, verzichtet aber auf das frühe 20. Jahrhundert und die am schwierigsten einzuordnenden Komponisten aus Datenset 5. Zudem wurden nur solche Komponisten aufgenommen, von denen mindestens 55 *MIDI*-Dateien zur Verfügung standen. Ursprünglich war diese Menge nicht vorgesehen. Sie wurde später aufgrund der Erkenntnisse aus allen anderen Netzen zusammengestellt, um bei Stücken vieler verschiedener Komponisten eine möglichst gute Korrektklassifizierung nach Epoche zu erreichen. Konkret enthält die Menge jeweils 165 Stücke
  - aus dem Zeitalter des **Mittelalters** und der **Renaissance**
  - aus der **Barockzeit**, vertreten durch Bach, Händel und Scarlatti
  - aus der **Wiener Klassik**, vertreten durch Haydn, Mozart und Beethoven
  - aus der **Romantik**, vertreten durch Schumann, Chopin, und Brahms

– aus der Kategorie des **Ragtime**, vertreten durch diverse Komponisten

Von diesen jeweils 165 Stücken werden 120 zum Training verwendet und 45 zur Validierung.

### 4.2.3. Vorüberlegungen zur Aufbereitung der Daten zur Eingabe in ein neuronales Netz

Benötigt werden Merkmalsvektoren bzw. Merkmalsmatrizen gleicher Dimension, um sie an die Eingabeschicht eines neuronalen Netzes anzulegen. Bei Musikstücken handelt es sich ebenso wie bei Sprache um *sequenzielle Daten*, in dem Sinne, dass verschiedene Klänge aufeinanderfolgen und die Reihenfolge eine Rolle spielt. Obwohl ein Musikinstrument zunächst einen kontinuierlichen Datenstrom erzeugt, kann dieser bei digitalen Aufnahmen und ebenso in *MIDI*-Dateien nur in zeitdiskreten Signalen erfasst werden. In einer *MIDI*-Datei sind die kleinsten Zeiteinheiten sogenannte „*Ticks*“. Wie lange solch ein *Tick* dauert, wird individuell im Header der Datei festgelegt. In jedem Fall ist es ein Bruchteil einer Viertelnote, der etwa zwischen 1/1000 und 1/24 liegt. Zur Eingabe in ein neuronales Netz macht es keinen Sinn, den Klang in derart kleinen Zeitabständen zu erfassen. Selbst bei ausreichender Rechen- und Speicherkapazität würde dies immer zu einer Überanpassung an die Trainingsdaten führen. Es bieten sich verschiedene Möglichkeiten für musikalisch sinnvolle Quantisierungen an. Betrachtet man einen *Takt* als musikalische Einheit, so kann dieser in eine feste Anzahl Teile zerlegt werden. Auch ein *Schlag* (meist eine Viertel- oder eine Achtelnote) kann zerlegt werden. Musikalisch weniger sinnvoll ist die Zerlegung einer *Sekunde*, jedoch simuliert dies die Verarbeitung von Audiodateien. All diese Möglichkeiten wurden im Rahmen der Arbeit getestet, jeweils mit unterschiedlich feiner Zerlegung. Eine Sequenz und damit ein Trainingsbeispiel entspricht dann den jeweils ersten Takten, Schlägen oder Sekunden eines Musikstückes.

Die Frage ist nun, wie die relevanten Informationen aus den Dateien extrahiert werden können. *MIDI*-Dateien sind zunächst einmal reiner Byte-Code, der mit Hilfe eines Hex-Editors leicht umgewandelt werden kann. Dies hilft jedoch nur, um den Aufbau der Dateien und die Idee hinter *MIDI* nachzuvollziehen. Es stehen auch Parser zur Verfügung, welche die Dateien in das von Menschen intuitiv lesbare *csv*-Format umwandeln. Auch solche Dateien sind für das neuronale Netz zunächst unbrauchbar, da zum einen die verschiedenen Nachrichtentypen unterschiedlich viele Attribute besitzen und es zum anderen beliebig viele *Meta-Messages* zwischen den codierten Tönen gibt. Benötigt werden jedoch Vektoren gleicher Länge und gleicher Semantik. Um nicht direkt auf dem Hexadezimal-Code arbeiten zu müssen, wurde die in Abschnitt 2.3.3 beschriebene *Python*-Bibliothek *Mido* zu Hilfe genommen.

Da *MIDI*-Dateien in erster Linie zur Wiedergabe und zur Kommunikation zwischen verschiedenen Hardware-Komponenten erstellt werden, kommt es häufig zu Fehlern in der Notation, wenn sie in einem Editor oder Notensatzprogramm geöffnet werden. Zum einen können bei live eingespielten Stücken niemals Beginn und Ende aller Töne auf den Tick genau erfasst werden. Dasselbe gilt für automatisch in *MIDI* umgewandelte Audio-Dateien. Um die Sequenzen dennoch wie oben beschrieben zu quantisieren, muss

*gerundet* werden. Ist nun jedoch die Granularität der Codierung zu grob, um alle Töne zu unterschiedlichen Zeitpunkten darzustellen, so wäre es wünschenswert, nur solche Töne auf denselben Zeitpunkt zu runden, die tatsächlich gleichzeitig erklingen sollten, und weniger relevante Töne zu ignorieren.

Ein zweites Problem stellt die *Taktart* dar, die in vielen *MIDI*-Dateien falsch angegeben ist. So werden fälschlicherweise viele Walzer im 4/4-Takt notiert, so dass selbst Experten die musikalische Struktur wie z.B. achttaktige Phrasen nicht auf Anhieb erfassen können. Auftakte werden häufig ignoriert, so dass dadurch alle Taktstriche entsprechend verschoben sind.

Auch die angegebene *Tonart* ist in vielen Fällen falsch. Teilweise wird statt einer Moll-Tonart die parallele Dur-Tonart codiert. Dies ändert zwar nichts am Notenbild, jedoch kann die Angabe der Tonart nicht als Merkmal verwendet werden. Wenn nicht einmal die Anzahl der Vorzeichen korrekt ist (d.h. die Versetzungzeichen werden direkt vor der Note notiert statt zu Beginn des Stückes), ist es auch nicht ohne weiteres möglich, alle Stücke in dieselbe Tonart (z.B. *C-Dur* bzw. *a-Moll*) zu transponieren. Da aufgrund zu weniger Trainingsbeispiele die Entscheidung fiel, im Zuge einer Datenaugmentation jedes Stück in alle 12 Tonarten zu transponieren, wurde darauf verzichtet, die korrekte Tonart auf andere Art zu bestimmen.

Ebensowenig zuverlässig sind Angaben bezüglich *Tempowechsel* oder *Pedalgebrauch*. Bei live eingespielten Stücken wechselt das Tempo selbst in Werken der Wiener Klassik auf jedem Schlag. In romantischen Werken wird teilweise auf den Einsatz des Pedals verzichtet, was völlig untypisch ist. Aus diesem Grund wurde schließlich die Verwendung von Meta-Events zur Klassifikation auf ein notwendiges Minimum beschränkt. Dazu zählen nur solche Meta-Events, die das Tempo und die Taktart bestimmen.

Es stellt sich die Frage, bei welcher Art der Codierung die aufgezählten Mängel am wenigsten ins Gewicht fallen. Hierzu werden drei Versionen der Merkmalserstellung implementiert und getestet. Allen *gemeinsam* ist zunächst die Verwendung einer Matrix, deren erste Dimension die zu erfassenden Zeitpunkte codiert. Die zweite Dimension ist durch den Tonumfang des Klaviers bestimmt, zuzüglich der Töne, die durch das Transponieren in alle Tonarten auftreten können und über diesen Umfang hinausgehen. Ein Klavier hat 88 Tasten. Aufgrund des Transponierens in 11 weitere Tonarten müssen 99 Töne darstellbar sein. Der Grund dafür, dass als erste Dimension die Zeit gewählt wird und nicht, wie die Notenschrift es intuitiv nahelegen würde, die Tonhöhe, liegt in der schlussendlichen Verwendung von *LSTM*-Netzen, die dies so verlangen. Zur Eingabe in ein *vollverbundenes* Netz wird die Matrix ohnehin zu einem Vektor transformiert, also sozusagen „flach gedrückt“.

Ebenfalls *gemeinsam* ist den Merkmalserstellungen die Tatsache, dass ein Ton im Moment des Anschlags durch den Wert 1 codiert wird, während alle folgenden Matrixeinträge der entsprechenden Spalte bis zum `note_off`-Event den Wert 0.2 (für „*klingt noch*“) erhalten. Ein nicht-klingender Ton wird durch den Wert 0 repräsentiert.

Der *Unterschied* zwischen den Merkmalserstellungen liegt in der Art der *Quantisierung*, d.h. in der Auswahl der Zeitpunkte, zu denen die gerade klingenden Töne erfasst werden. Die drei Versionen sind folgende:

- **Version 1** ist die **taktabhängige Codierung**. Hier wird die angegebene Taktart verwendet und ein Takt in eine feste Anzahl Abschnitte unterteilt. Sinnvoll ist einerseits eine Zahl mit möglichst vielen Teilern, so dass in allen denkbaren Taktarten die Taktschwerpunkte und möglichst viele kleinere Notenwerte genau erfasst werden können. Andererseits darf die Anzahl der Merkmale auch nicht zu groß werden, da aufgrund der begrenzten Anzahl von Werken klassischer Komponisten die Gefahr eines *Overfitting* ohnehin gegeben ist. Die besten Ergebnisse wurden mit 33 Takten und 12 Abschnitten pro Takt erzielt. Die Zahl 33 liegt in der Tatsache begründet, dass Auftakte in *MIDI*-Dateien in der Regel einen vollen Takt beanspruchen, der mit Pausen beginnt. Dennoch soll ein meist sinnvoller Abschnitt von vier mal acht Takten vollständig erfasst werden. Intuitiv sollten mit dieser Merkmalserstellung auch die insgesamt besten Ergebnisse erzielt werden können, da auf den Taktschwerpunkten in der Regel die relevanten Töne stehen, die ein menschlicher Interpret auch etwas hervorheben würde. Der Grund dafür, dass die beiden im Folgenden erläuterten Codierungen auf ähnlich hohe Werte kommen, liegt sicher in den oben erwähnten Mängeln des *MIDI*-Codes.

- **Version 2** ist die **schlagabhängige Codierung**. Sie berücksichtigt nur den Wert eines Grundschlages, d.h. den Wert, der bei der Taktangabe im Nenner steht. In einem 4/4- oder 3/4-Takt wäre dies eine Viertelnote, in einem 3/8- oder 6/8-Takt eine Achtelnote. Ein solcher Schlag wird nun wiederum in eine feste Anzahl Zeiteinheiten geteilt. Recht gute Ergebnisse wurden bereits ohne weitere Unterteilung erzielt, die besten bei Unterteilung eines Schlages in vier oder sechs Abschnitte. Betrachtet werden jeweils die ersten 132 Schläge der Stücke. Grund für die Implementierung dieser Version ist die Tatsache, dass dadurch nicht-berücksichtigte Auftakte und 4/4- statt 3/4-Takte keine negativen Auswirkungen mehr haben.

- **Version 3** ist die **zeitabhängige Codierung**. Dies wurde getestet, da hier die genannten Probleme am wenigsten ins Gewicht fallen sollten. Jedoch wurden zumindest keine besseren Ergebnisse erzielt als mit anderen Versionen. Notwendig ist hier in jedem Falle eine verhältnismäßig feine Granularität, da Stücke ansonsten selbst für Experten kaum erkennbar wären. Hier wurden die besten Ergebnisse mit 25 Sekunden und 12 Teilen pro Sekunde erzielt.

## 4.3. Vorüberlegungen zu Dimensionierung und Ausgestaltung des Netzes

Selten hat ein Komponist mehr als einige Hundert Werke komponiert. Zudem wird ab der Barockzeit hier nur Klaviermusik berücksichtigt. Das Instrument wird bei der Merkmalserstellung zwar nicht erfasst, jedoch könnte bei Orchesterwerken schon allein die Anzahl der gleichzeitig erklingenden Töne einen Hinweis auf den Komponisten geben. Somit ist die Menge der zur Verfügung stehenden Daten im Vergleich zu anderen Problemen des maschinellen Lernens sehr beschränkt und die Gefahr einer Überanpassung besonders groß. Daher wird ein eher kleines Netz benötigt.

Bei Musik handelt es sich ebenso wie bei Sprache um *sequenzielle* Daten. Ein Akkord steht nicht für sich alleine, sondern hängt von den vorherigen ab und beeinflusst die folgenden. Es bietet sich ein *rekurrentes* Netz an, wie es auch zur Verarbeitung von Textdateien und natürlicher Sprache verwendet wird. Dennoch werden hier drei verschiedene Typen neuronaler Netze untersucht:

- ein *vollverbundenes Feedforward-Netz*: Zunächst werden zwei innere Schichten mit nur 32 bzw. 16 Einheiten verwendet, zur Regularisierung dient Dropout. Geoffrey Hinton schlägt vor, mit einer Dropout-Rate von 0.5 zu beginnen und diese je nach Ergebnissen zu variieren [25].
- ein *LSTM*-Netz, da dieses für sequenzielle Daten prädestiniert ist. Es bietet sich auch hier ein kleines Netz mit nur 16 oder 32 Knoten an, das durch Dropout reguliert wird. Auch ein bidirektionales Netz wird getestet, da die Betrachtung einer Sequenz in umgekehrter Richtung zusätzliche Erkenntnisse bringen kann.
- eine Kombination aus einem *1D-konvolutionalen* und einem *LSTM*-Netz. Der *LSTM*-Einheit werden konvolutionale Schichten vorgeschaltet, um zunächst Merkmale zu extrahieren und die Sequenzen dadurch zu verkürzen [29].

Jede der drei genannten Netz-Architekturen wird jeweils mit den drei in Abschnitt 4.2.3 erläuterten Möglichkeiten zur Merkmalscodierung getestet, um die optimale Kombination zu ermitteln.

## 4.4. Details der Implementierung

### 4.4.1. Details der Merkmalserstellung

Die Implementierung als *Jupyter-Notebook* sieht so aus, dass in einer Schleife alle Dateien des gewählten Ordners durchlaufen werden. Aus jedem Dateinamen wird zunächst der Komponist herausgelesen, mit Hilfe eines *Dictionaries* als Integer-Wert codiert und in Spalte 0 des zu erzeugenden Vektors abgelegt. Hierbei handelt es sich um das *Label* ($y$), das vor der Eingabe ins Netz von den Merkmalen abgetrennt wird. Bei den Trainingsdaten dient innerhalb der Schleife eine weitere Schleife dazu, das aktuelle Stück in alle 12 Tonarten zu transponieren, indem zu dem Wert der Tonhöhe in den `note_on`- und `note_off`-Events jeweils eine ganze Zahl des Wertebereiches [-5, 6] addiert wird. Dies ist beim Einlesen der Validierungs- und Testdaten nicht notwendig. Nun werden der Reihe nach alle Messages der Datei durchlaufen und deren *time*-Attribut dazu verwendet, die einzelnen Zeiteinheiten (im Programm als „`pulse`“ bezeichnet) aufzuaddieren. Die Länge einer solchen Einheit ergibt sich in der zeitabhängigen Version direkt durch Division einer Sekunde durch die im Voraus festzulegende Konstante *PARTS_PER_SECOND*. In der takt- und schlagabhängigen Version wird sie anhand der Message-Typen *set_tempo* und *time_signature* sowie der Konstanten berechnet. Hier werden für die Konstanten direkt die Werte angegeben, die sich als die günstigsten herausstellten:

- **Taktabhängige Version**:
  - Festlegung der Konstanten:

$$\begin{aligned} \texttt{NUMBER\_BARS} &= 33 \\ \texttt{PARTS\_PER\_BAR} &= 12 \\ \texttt{NUMBER\_PULSES} &= \texttt{NUMBER\_BARS} * \texttt{PARTS\_PER\_BAR} \end{aligned}$$

  - `set_tempo`-Message:

$$\begin{aligned} \texttt{tempo} &= \texttt{msg.tempo}/1000000. \\ \texttt{pulse} &= (\texttt{factor} * \texttt{numerator} * \texttt{tempo})/\texttt{PARTS\_PER\_BAR} \end{aligned}$$

  - `time_signature`-Message:

$$\begin{aligned} \texttt{numerator} &= \texttt{msg.numerator} \\ \texttt{factor} &= 4./\texttt{msg.denominator} \\ \texttt{pulse} &= (\texttt{factor} * \texttt{numerator} * \texttt{tempo})/\texttt{PARTS\_PER\_BAR} \end{aligned}$$

- **Schlagabhängige Version**:
  - Festlegung der Konstanten:

$$\begin{aligned} \texttt{NUMBER\_BEATS} &= 132 \\ \texttt{PARTS\_PER\_BEAT} &= 4 \\ \texttt{NUMBER\_PULSES} &= \texttt{NUMBER\_BEATS} * \texttt{PARTS\_PER\_BEAT} \end{aligned}$$

  - `set_tempo`-Message:

$$\begin{aligned} \texttt{tempo} &= \texttt{msg.tempo}/1000000 \\ \texttt{pulse} &= (\texttt{factor} * \texttt{tempo})/\texttt{PARTS\_PER\_BEAT} \end{aligned}$$

  - `time_signature`-Message:

$$\begin{aligned} \texttt{factor} &= 4./\texttt{msg.denominator} \\ \texttt{pulse} &= (\texttt{factor} * \texttt{tempo})/\texttt{PARTS\_PER\_BEAT} \end{aligned}$$

- **Zeitabhängige Version**:
  - Festlegung der Konstanten:

$$\begin{aligned} \texttt{NUMBER\_SECONDS} &= 25 \\ \texttt{PARTS\_PER\_SECOND} &= 12 \\ \texttt{NUMBER\_PULSES} &= \texttt{NUMBER\_SECONDS} * \texttt{PARTS\_PER\_SECOND} \end{aligned}$$

  - Festlegung der Länge einer Zeiteinheit:

$$\texttt{pulse} = 1./\texttt{PARTS\_PER\_SECOND}$$

Mit jedem `note_on`- und `note_off`-Event wird die zunächst mit Nullen initialisierte Notenmatrix bearbeitet. Die 99 Spalten dieser Matrix entsprechen den Tonhöhen, die Zeilen den einzelnen Zeiteinheiten. Bei `note_on` erhält die der Note entsprechende Spalte genau in der Zeile den Wert 1, die dem berechneten Zeitpunkt am nächsten liegt. Alle folgenden Zeilen erhalten den Wert 0.2, der signalisiert, dass ein Ton noch klingt. Bei `note_off` werden ab der entsprechenden Stelle alle Werte wieder auf 0 gesetzt. Um sicherzustellen, dass auch kleine Notenwerte berücksichtigt werden, bei denen Beginn und Ende auf dieselbe Zeilennummer gerundet werden, wird eine eventuelle 1 nicht auf 0 gesetzt, sondern nur die Einträge mit dem Wert 0.2. Bis hierher ist die Merkmalserstellung unabhängig von der Art des zur Klassifizierung verwendeten neuronalen Netzes. Die Anpassung erfolgt erst im Anschluss:

Bei einem *vollverbundenen Feedforward-Netz* wird die gebildete Matrix sozusagen flach gedrückt, indem alle Zeilen in einen einzigen Zeilenvektor übertragen werden. Als Trainings- und Testmenge werden wieder zweidimensionale Matrizen verwendet, deren Zeilen den Vektoren der einzelnen Beispiele entsprechen.

Ein *LSTM*-Netz und ebenso ein *1D-konvolutionales Netz* benötigen *3D-Tensoren*. Hierzu muss die gebildete Notenmatrix nicht verändert werden. Die Matrizen der einzelnen Beispiele werden lediglich aneinander gehängt.

Nun werden noch die Labels abgetrennt und *kategorisiert*. Diese Kategorisierung wird als *One-hot-Codierung* bezeichnet. Jedes Label wird als Vektor dargestellt, dessen Einträge aus Nullen und einer Eins (für die tatsächliche Kategorie) besteht. Die Labels in One-hot-Codierung befinden sich im Anschluss in einem separaten zweidimensionalen Array.

### 4.4.2. Details des Netzaufbaus und des Trainings

Nachdem die Daten in der gewünschten Form vorliegen, besteht das weitere Vorgehen aus drei Schritten, die im Folgenden näher erläutert werden. Zunächst wird ein **Modell erzeugt**, dann wird es **kompiliert**, und schließlich kann das Netz mit der Trainingsmenge **trainieren**. Das Training kann anhand der Validierungsmenge überwacht und bei Bedarf von Hand abgebrochen werden

Die wichtigste Datenstruktur in *Keras* ist das *Modell*. Hier wird nur der Haupttyp eines Modells benötigt, das *sequenzielle Modell*. Dabei handelt es sich ganz einfach um einen Stapel von Schichten (`layers`). Mittels `model = models.Sequential()` kann ein solches Modell **erzeugt** werden. Durch die Methode `model.add(...)` werden dem Modell die verschiedenen Schichten nacheinander hinzugefügt.

- In einem **vollverbundenen Netz** sind dies beliebig viele Schichten des Typs `layers.Dense`. Jede Einheit einer Schicht ist so mit jeder Einheit der vorhergehenden und der nachfolgenden Schicht verbunden. Erforderliches Argument ist in diesem Fall nur die Anzahl der Einheiten dieser Schicht, und in der ersten Schicht die `input_shape`.
- Im **LSTM-Netz** werden eine oder mehrere Schichten des Typs `layers.LSTM` benötigt. Im allen vorliegenden Fällen führt mehr als eine Schicht zu einer Überanpassung. Jedoch kann eine *Bidirektionalität* die Ergebnisse leicht verbessern. Die Angabe der `input_shape` ist nicht notwendig, jedoch ist es wichtig, dass die erste Dimension eines einzelnen Datensatzes der Zeitachse entspricht.
- Im **konvolutionalen LSTM-Netz** werden der *LSTM*-Schicht eine oder mehrere Schichten des Typs `layers.Conv1D` mit Angabe der Anzahl an Ausgabekanälen vorangeschaltet. Auf jede konvolutionale Schicht folgt jeweils eine Schicht des Typs `layers.MaxPooling1D` mit Angabe der Anzahl der Ausgabekanäle und der Filterkerngröße.

Für die *Aktivierungsfunktionen* aller vollverbundenen, *LSTM*- und konvolutionalen Schichten gibt es Default-Werte. Beim `dense`-Layer und in konvolutionalen Layern ist dies allerdings `None`, sinnvollerweise sollte daher eine Funktion festgelegt werden. In *LSTM*-Layern ist `tanh` voreingestellt, dies wurde so belassen. Allgemein kann die Aktivierungsfunktion entweder der Schicht als String übergeben oder als separate Schicht angefügt werden.

Nachdem das Modell vollständig definiert ist, wird es **kompiliert**. Dabei muss ein *Optimierer* gewählt werden. Diesem werden optional verschiedene Argumente übergeben, die ebenfalls Default-Werte besitzen. Dennoch sollten sie experimentell verändert werden, um das Modell zu optimieren. Das wichtigste Argument ist die Lernrate (`lr`), aber auch die Einführung eines `decay` bringt oft Vorteile. Alle Optimierer in *Keras* besitzen die optionalen Parameter `clipnorm` und `clipvalue`. Diese dienen dazu, dem Problem der *explodierenden Gradienten* zu begegnen. Im *LSTM*-Netz ist dies notwendig, weil ansonsten die Verlustfunktion den Wert `NaN` annimmt. Der `clipvalue` beschränkt die absoluten Werte der Einträge des Gradienten, die `clipnorm` die *L2-Norm* des Gradienten. Die Wahl fiel auf die Verwendung der `clipnorm`, da diese gewährleistet, dass die Abstiegsrichtung beibehalten wird.

Um das Training überwachen und das Modell validieren zu können, sind zudem eine *Verlustfunktion* und mindestens eine *Metrik* notwendig, die ebenfalls dem Optimierer mitgegeben werden. Als Metrik genügt die *Korrektklassifizierungsrate* (`accuracy`), die

nach jeder Epoche des Trainings gespeichert und ausgegeben wird. Als Verlustfunktion dient hier in allen Netzen die Funktion `categorical_crossentropy(y_true, y_pred)`. Hierbei handelt es sich um die in Abschnitt 3.1.2.2 dargestellte und auf mehrere Kategorien verallgemeinerte Verlustfunktion für logistische Regression. Dies ist der Standard für Klassifikationsaufgaben.

Vor dem Training wird ein `ModelCheckpoint` erzeugt, der die Möglichkeit bietet, nach Abschluss des Trainings die Gewichte des Modells mit denen zu überschreiben, die zum besten Ergebnis auf den Validierungsdaten geführt haben. Dadurch ist es nicht mehr notwendig, *die* ideale Epochenzahl zu finden, bei welcher der Höhepunkt am Ende liegt. Dies wäre ohnehin sehr schwierig, da der Verlauf des Trainings von der zufälligen Initialisierung der Gewichte abhängt und im Falle des tatsächlich besten Wertes nicht erkennbar ist, ob dieser noch zu steigern wäre. Mit Hilfe des Checkpoints kann also getrost eine höhere Epochenzahl gewählt werden, so dass mit großer Sicherheit davon auszugehen ist, dass die Werte später nicht noch weiter steigen würden.

Das eigentliche **Training** wird durch die Methode `model.fit()` angestoßen. Diese erhält zunächst die Trainings-Datensätze und Labels. Zusätzlich werden Stapelgröße und Anzahl der Epochen gewählt und die Validierungsdaten sowie der zuvor erzeugte `ModelCheckpoint` als `callback` übergeben. Alle weiteren Argumente behalten hier ihre Default-Werte. Die Methode gibt ein `History`-Objekt zurück, das die Werte der Verlustfunktion und der Korrektklassifizierungsrate aller Epochen auf den Trainings- und Validierungsdaten speichert.

Bevor im folgenden Kapitel die konkreten Konfigurationen der Modelle inklusive Einstellung der Hyperparameter vorgestellt werden, seien hier zunächst die verwendeten Aktivierungsfunktionen und Optimierer aufgelistet, da die bereits ausführlich erläuterten Standardversionen häufig nicht die besten Ergebnisse liefern.

- **Aktivierungsfunktionen**: Die Aktivierungsfunktion kann für jede Schicht separat ausgewählt werden.
  - `sigmoid`: Hierbei handelt es sich um die in Abschnitt 3.1.2.1 erläuterte *logistische Funktion*.
  - `relu`: Die Bezeichnung *ReLU* steht für *Rectified Linear Unit*. Die Funktion bildet positive Werte auf sich selbst ab und negative auf 0. Xavier Glorot et al. zeigten 2011, dass sie in tiefen neuronalen Netzen trotz ihrer Nichtlinearität und der fehlenden Differenzierbarkeit bei 0 bessere Ergebnisse liefert als die logistische Funktion [10].
  - `tanh`: Der *Tangens hyperbolicus* liefert Werte aus dem Bereich zwischen $-1$ und $+1$. Verwendung findet diese Funktion implizit bei Berechnung der Ausgabe im *LSTM*-Netz. Dies wurde in Abschnitt 3.1.5.9 dargestellt.
  - `softmax`: Die *Softmax*-Funktion wird hier in der Ausgabeschicht aller Netze verwendet. Bei nur zwei Kategorien entspricht sie der logistischen Funktion. Bei mehr als zwei Kategorien eignet sie sich jedoch besser, da die Werte

aller Kategorien für ein Beispiel sich jeweils zu eins addieren und damit als Wahrscheinlichkeiten interpretiert werden können.

- **Optimierer**
  - `SGD`: Das Akronym steht für *Stochastic Gradient Descent*. Der Gradientenabstieg wurde in den Abschnitten 3.1.1.2 und 3.1.2.3 erklärt, die stochastische Variante in Abschnitt 3.1.1.3. In *Keras* ist hiermit allerdings das *Mini-Batch*-Gradientenverfahren gemeint, welches ebenfalls in Abschnitt 3.1.1.3 erwähnt wird. Die Stapelgröße ist frei wählbar, hier wird sie in allen Versionen auf 64 festgelegt. Alle Argumente besitzen Default-Werte. Mindestens die Lernrate (`lr`= 0.01) sollte jedoch variiert werden. Zudem wird in einigen Fällen ein `decay` verwendet. Dabei handelt es sich um eine schrittweise Verringerung der Lernrate, um am Ende nicht immer wieder über das Minimum hinweg zu springen.
  - `RMSprop`: Bei *Root Mean Square Propagation* handelt es sich um eine der vielen Optimierungsmöglichkeiten des Gradientenabstiegs. Die Lernrate für jedes einzelne Gewicht wird durch den gleitenden Mittelwert der Größen der bisherigen Gradienten für dieses Gewicht geteilt. Dies dient dazu, das Verfahren zu beschleunigen. In der Dokumentation von *Keras* wird diese Variante insbesondere für rekurrente Netze empfohlen. Die initiale Lernrate sollte auch hier variiert werden, um den besten Wert zu ermitteln. Die übrigen Argumente sollten laut Dokumentation bei ihren Default-Werten belassen werden.

# 5. Konfiguration, Auswertung und Auswahl des besten Netzes

## 5.1. Konfiguration der neun zu bewertenden Netze

Es kommen drei verschiedene Netztypen zum Einsatz, deren Leistungen bei der Klassifikation von *MIDI*-Dateien anhand einer Validierungsmenge verglichen werden. Diese Netztypen werden im Folgenden erläutert. Sobald die Kombination aus Netztyp und Merkmalscodierung feststeht, die die besten Ergebnisse auf den Validierungsdaten liefert, wird diese Variante in Kapitel 6 weiter optimiert. Der Grund dafür, dass erst dort eine Testmenge zum Einsatz kommt, ist der, dass dies zum Vergleich verschiedener Netze noch nicht zwingend erforderlich ist. Die Feinabstimmung eines Netzes erfordert extrem viel Zeit, daher wird dies nur auf dem Netztyp durchgeführt, auf dem die besten Ergebnisse zu erwarten sind.

- **Vollverbundenes Feedforward-Netz**: Als günstig erweisen sich lediglich zwei verdeckte Schichten mit jeweils 32 Einheiten. Die *Shape* des Eingabe-Tensors muss in der ersten verdeckten Schicht angegeben werden. Sie beträgt in jedem Fall 99 mal die Anzahl der codierten Zeiteinheiten. Als *Aktivierungsfunktion* aller verdeckten Schichten dient `relu`. Auf jede verdeckte Schicht folgt eine Dropout-Schicht. Eine *Dropout-Rate* von 0.4 bringt in der taktabhängigen Version der Merkmalscodierung die besten Ergebnisse. In der schlag- und zeitabhängigen Version liegt die Dropout-Rate mit 0.45 etwas höher. Die Ausgabeschicht besitzt so viele Einheiten wie es Kategorien gibt, Aktivierungsfunktion ist hier `softmax`, da sich Wahrscheinlichkeiten am besten zur Auswertung eignen.

  Beim Kompilieren des Modells wird der *Optimierer* ausgewählt. Mit `SGD` können eine bis zum Minimum monoton fallende und anschließend wieder steigende Verlustfunktion und entsprechend eine bis zu diesem Punkt annähernd monoton steigende Korrektklassifizierungsrate sowohl auf den Trainings- als auch den Validierungsdaten erzielt werden. Abbildung 5.1 zeigt ein solches Ergebnis.

  Mit `RMSprop` geht diese Monotonie verloren, dennoch wird ein insgesamt besseres Optimum gefunden. Die initiale Lernrate beträgt in allen Konfigurationen der drei Versionen 0.001, der `decay` 0.0001. Als Verlustfunktion dient hier und in allen im Folgenden vorgestellten Netztypen `categorical_crossentropy`, was bei einer Klassifikation mit mehr als zwei Kategorien so empfohlen wird.

  Abbildung 5.2 zeigt den Verlauf der Verlustfunktion und der Korrektklassifizierungsrate des besten Durchlaufs auf den Trainings- und Validierungsdaten. Das Optimum wird bereits in Epoche 13 erzielt.

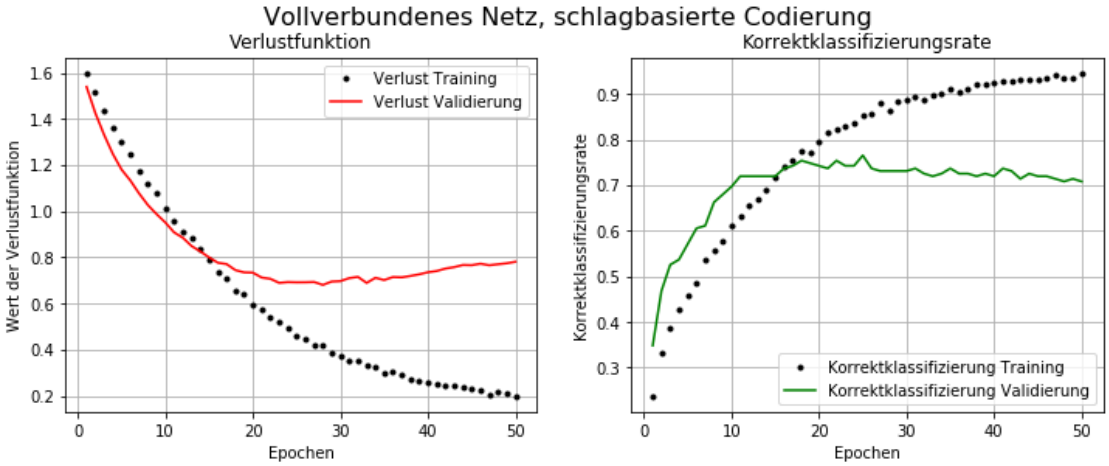



Abbildung 5.1.: Verlustfunktion und Korrektklassifizierungsrate in einem vollverbundenen Netz mit SGD, der Grundform des Gradientenabstiegs

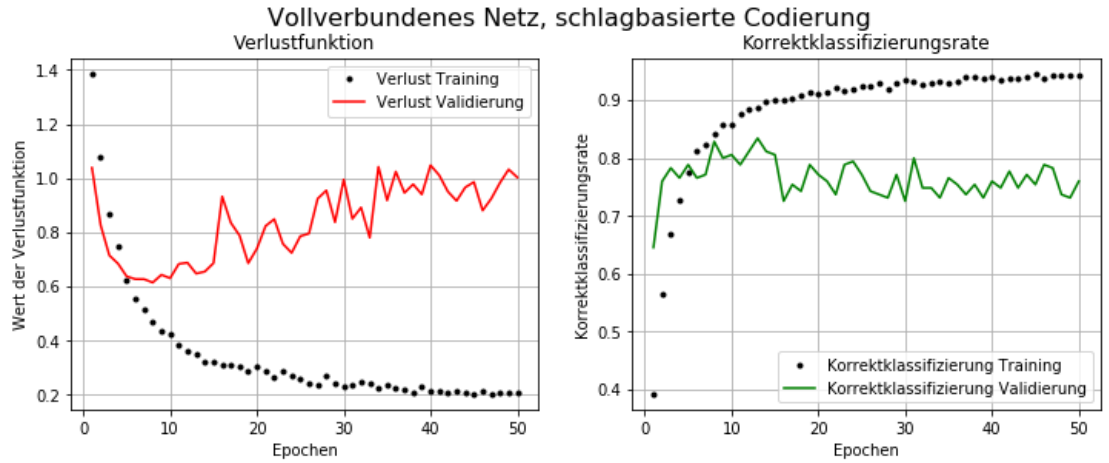



Abbildung 5.2.: Verlustfunktion und Korrektklassifizierungsrate in einem vollverbundenen Netz mit RMSprop, einer optimierten Form des Gradientenabstiegs

In Abbildung 5.3 sind die Konfigurationen der drei vollverbundenen Netze noch einmal schematisch dargestellt.

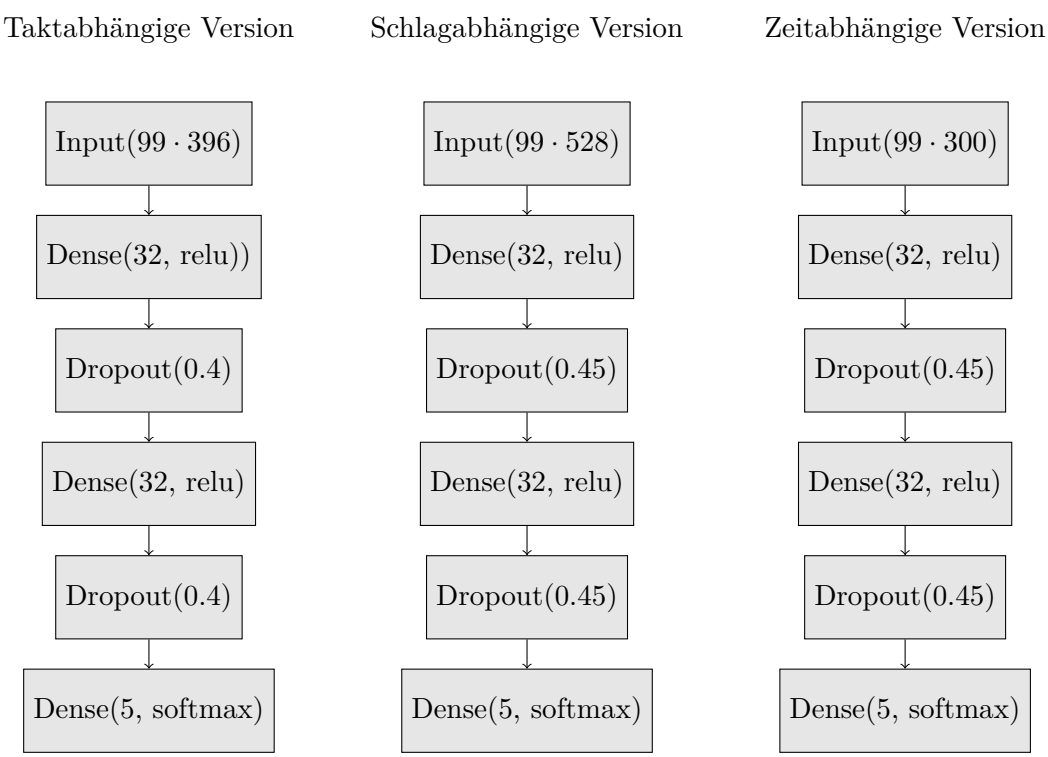



Abbildung 5.3.: Schematische Darstellung der drei vollverbundenen Netze

- ***LSTM*-Netz**: Die besten Ergebnisse erzielt die taktbasierte Version in einer Konfiguration mit einem *bidirektionalen* Layer aus 16 Einheiten, einer *Dropout-Rate* von 0.21, einem *rekurrenten Dropout* von 0.31 sowie einem *Dropout auf die Ausgabe* von 0.29. Der Hintergrund dieser Differenzierung ist einem Artikel von Yarin Gal und Zoubin Ghahramani über die Anwendung von Dropout in rekurrenten Netzen aus dem Jahr 2016 zu entnehmen [7]. In Abbildung 3.18 wird der reguläre Dropout in vertikaler Richtung angewendet, er betrifft demnach die Transformation der $\mathbf{x}_t$ zu den $h_t$ in einem einzelnen Zeitschritt. Konkret werden Teile der Eingabe vorübergehend ausgeschaltet. Der rekurrente Dropout bezieht auf die rekurrenten Einheiten, wird also anschaulich in einem abgerollten Netz in waagerechter Richtung angewendet. Zusätzlich kann hinter dem *LSTM*-Layer eine Dropout-Schicht eingefügt werden, die den Dropout auch auf den Output anwendet. Beim Kompilieren des Modells wird auch hier `RMSprop` als *Optimierer* gewählt. Die initiale *Lernrate* beträgt in allen Versionen der Merkmalscodierung 0.01. Auf einen `decay` wurde letztendlich verzichtet, da er zwar die starken Schwankungen der Verlustfunktion gegen Ende des Trainings verringerte, jedoch nie zu einem besseren Gesamtergebnis führte. Abbildung 5.4 veranschaulicht die Konfiguration der drei *LSTM*-Netze.

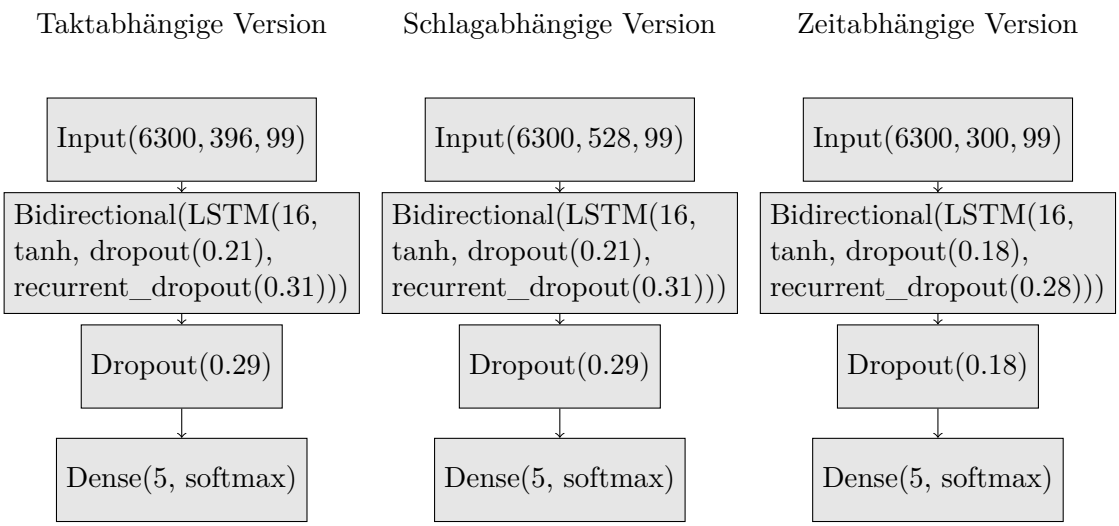



Abbildung 5.4.: Schematische Darstellung der drei LSTM-Netze

- ***LSTM*-Netz mit vorgeschalteten konvolutionalen Schichten**: Wie bereits in Abschnitt 3.1.5.8 erläutert, sind *2D-CNN*s für die Verarbeitung visueller Daten prädestiniert, da sie Muster im zweidimensionalen Raum lernen und an jeder Stelle im Bild wiedererkennen können. Dieses Prinzip wird in *1D-CNN*s auf zeitliche Muster angewendet. Sie bestehen entsprechend aus einer Reihe von *Conv1D*- und *MaxPooling-1D*-Layern und enden entweder mit einem *globalen MaxPooling* oder einer *Flatten*-Operation. Die Reihenfolge der Zeitschritte spielt nur innerhalb der Größe des Faltungsfensters eine Rolle. Insbesondere bei langen Sequenzen bietet es sich nun an, solch ein *CNN*-Netz (ohne die Abschluss-Operation) zur Vorverarbeitung der Daten einem rekurrenten Netz voranzuschalten. Dadurch werden die Sequenzen verkürzt. Das rekurrente Netz arbeitet schneller und berücksichtigt dennoch die Reihenfolge der erkannten Muster. Dieses Verfahren schlägt der Entwickler der *Keras*-Bibliothek François Chollet vor [4], es kommt aber bisher eher selten zum Einsatz und ist noch wenig erforscht. Da die Vereinigung der Stärken eines *konvolutionalen* und eines *LSTM*-Netzes vielversprechend klingt, wurden im Rahmen dieser Arbeit viele verschiedene Konfigurationen dieser Variante getestet. Dem *LSTM*-Netz wurden eine bis drei konvolutionale Schichten vorgeschaltet, jeweils mit zwischen acht und 64 auszugebenden Kanälen und Filterkerngröße drei und fünf. Bereits hier sei vorweggenommen, dass keine dieser Konfigurationen ein besseres Ergebnis erzielen konnte als ein reines *LSTM*-Netz. Lediglich die Laufzeit wurde um mehr als die Hälfte reduziert, von ca. 70 auf 30 Sekunden pro Epoche, wobei die absoluten Zahlen sehr von der Leistung des Rechners abhängen. Die in Anhang A.1 dokumentierten besten Ergebnisse der drei Versionen wurden mit einer initialen *Lernrate* von 0.001 und ohne *decay* erzielt. Abbildung 5.5 veranschaulicht die Konfiguration der drei konvolutionalen *LSTM*-Netze.

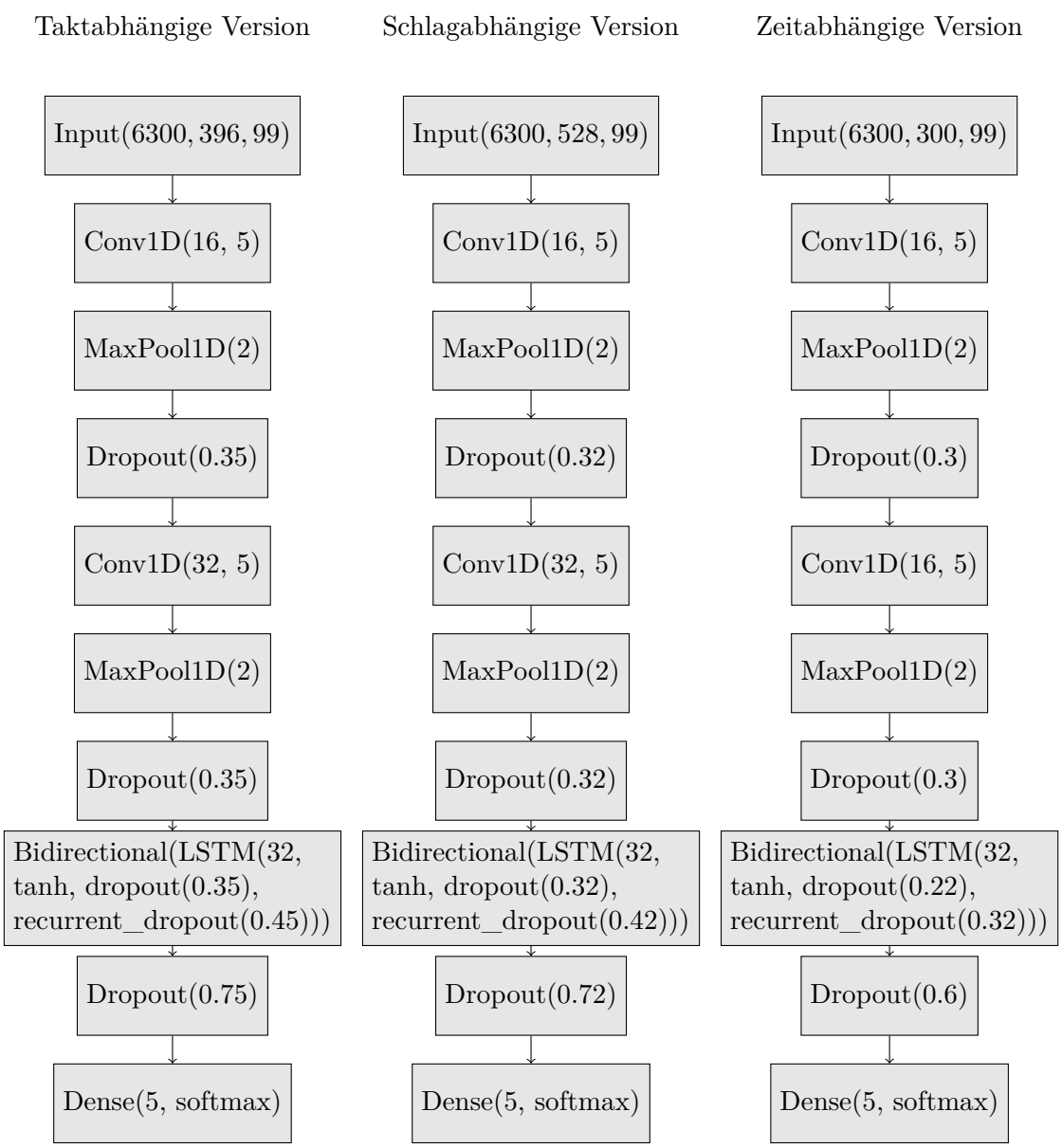



Abbildung 5.5.: Schematische Darstellung der drei konvolutionalen LSTM-Netze

## 5.2. Ermittlung der zur Evaluation benötigten Werte

Um die Qualität der verschiedenen Netze in Kombination mit den drei Versionen der Merkmalscodierung beurteilen zu können, ist es zunächst wichtig, den Verlauf der *Verlustfunktion* und der *Korrektklassifizierungsrate* jeweils auf den Trainings- und Validierungsdaten zu beobachten. Zu diesem Zweck werden die beiden Werte nach jeder Trainingsepoche ausgegeben und am Ende die entsprechenden Plots abhängig von der Epochenzahl gezeichnet. Im vorliegenden Fall ist der als wichtigstes Maß verwendete Wert der *Korrektklassifizierungsrate* ebenso aussagekräftig wie der *F1-Score*, da alle Klassen dieselbe Mächtigkeit besitzen. Andernfalls hätte dieser Wert nur eine geringe Aussagekraft bezüglich der Güte der Klassifikation. Zudem wird die *Konfusionsmatrix* erstellt, und es werden die Qualitätsmaße *Relevanz* (*precision*) und *Sensitivität* (*recall*) sowie der sich daraus ergebende *F1-Score* berechnet.

Interessant ist die Information, welche konkreten Stücke falsch klassifiziert werden, mit welcher Wahrscheinlichkeit diese der falschen Kategorie zugeordnet werden und wie hoch die Wahrscheinlichkeit für die tatsächliche Kategorie ist. Daher wird beim Einlesen der Validierungsmenge zusätzlich der Dateiname erfasst, der das jeweilige Stück eindeutig identifiziert. Dieser wird beim Abtrennen des Labels ebenfalls abgetrennt und in einem separaten Array „`titelfolge`“ gespeichert. Im Falle einer Nicht-Übereinstimmung werden die gewünschten Werte am Ende ausgegeben. Beträgt die berechnete Wahrscheinlichkeit für ein falsches Label 99%, so würde sich dies auch bei längerer Laufzeit kaum mehr ändern. Werden viele Stücke eines bestimmten Komponisten konsequent einer bestimmten anderen Epoche oder einem Komponisten zugeschrieben, so kann überprüft werden, ob auch Experten diese Werke häufig falsch einordnen.

## 5.3. Ergebnisse auf dem Validierungsdatensatz

Die Tabelle in Abbildung 5.6 beinhaltet die jeweils besten erzielten *Korrektklassifizierungsraten* aller neun möglichen Kombinationen von Netztyp und Merkmalscodierung auf der Validierungsmenge. Eine ausführlichere Dokumentation, die neben der Korrektklassifizierungsrate auch die Werte für die *Relevanz*, die *Sensitivität* und den *F1-Score* sowie die Plots der *Verlustfunktion* und der *Korrektklassifizierungsrate* enthält, findet sich in Anhang A.1.

| **accuracy** | Takt | Schlag | Zeit |
|---|---|---|---|
| Feedforward | 80.6% | **83.4%** | 75.4% |
| LSTM | **95.4%** | 94.3% | 93.7% |
| ConvLSTM | **93.1**% | 92.0% | 92.6% |

Abbildung 5.6.: Ergebnisse der Netzauswahlphase auf der Validierungsmenge

Wie zu erwarten, liegen die Leistungen des vollverbundenen Netzes wesentlich unter

denen der beiden rekurrenten Netze, und zwar unabhängig von der Art der Merkmalscodierung. Dennoch erreichen zumindest die takt- und schlagabhängige Variante Korrektklassifierungsraten von mehr als 80%. Unter Berücksichtigung der Tatsache, dass jeder erfasste Zeitpunkt hier als eigenständiges Merkmal betrachtet wird, unabhängig davon, welche Töne vorher und nachher erklingen, ist dies schon ein beachtliches Ergebnis. Dass die zeitabhängige Version gerade im vollverbundenen Netz deutlich abfällt, überrascht weniger, da die dadurch erfassten Klänge in der Regel nicht einmal die Taktschwerpunkte codieren und entscheidend vom festgelegten Tempo eines Stückes abhängen. Dasselbe Stück würde bei einem minimal abweichenden Tempo durch völlig andere Vektoren codiert. Die zeitabhängige Version kann daher nur für rekurrente Netze von Bedeutung sein.

Bedauerlich ist, dass die guten Leistungen des *LSTM* sich durch Vorschalten konvolutionaler Layer nicht steigern lassen. Die Laufzeit wird zwar um mehr als die Hälfte verringert, die Leistungen liegen jedoch knapp unter denen eines reinen *LSTM*-Netzes.

Da das beste Ergebnis mit einem *LSTM*-Netz und taktabhängiger Merkmalscodierung erzielt wird, soll im Folgenden diese Version auf einem neuen Datensatz näher untersucht und wenn möglich weiter optimiert werden. Dieser Datensatz enthält genügend Stücke, um neben Trainings- und Validierungsmenge auch eine Testmenge bereithalten zu können.

## 5.4. Einige Worte zur Laufzeit der Programme

An verschiedenen Stellen wurde bereits auf die Laufzeit der Programme hingewiesen, ohne konkrete Zahlen zu nennen. Der Grund dafür ist der, dass die Zeit in erheblichem Maße von der Rechnerleistung abhängt. Die in Abbildung 5.7 gegenübergestellten Laufzeiten *pro Epoche* wurden auf einer durchschnittlichen *Intel Core i5*-CPU erreicht. Die Anzahl der Epochen variiert je nach Netztyp zwischen 50 (*Feedforward-Netze*) und 800 (*LSTM*-Netze), so dass die Laufzeit des gesamten Trainings zwischen sechs Minuten und 23 Stunden liegt. Alle Zeiten beziehen sich auf die jeweils besten Konfigurationen des Netztyps und die bis hierher untersuchte erste Datenmenge. Im späteren Verlauf verlängern sich die Laufzeiten des *LSTM*-Netzes noch weiter, da die Zahl der Einheiten für das Training auf anderen Datenmengen in manchen Fällen vergrößert wird.

| **Laufzeiten/Epoche** | Takt | Schlag | Zeit |
|---|---|---|---|
| Feedforward | 10 s | 17 s | 7 s |
| LSTM | 76 s | 103 s | 52 s |
| ConvLSTM | 30 s | 40 s | 23 s |

Abbildung 5.7.: Trainingszeiten der Netze auf einer *Intel Core i5*-CPU in Sekunden pro Epoche

# 6. Feinabstimmung des LSTM-Netzes

Die Konfusionsmatrix des besten bisherigen Ergebnisses sowie *Relevanz*, *Sensitivität* und *F1-Score* sind in Abbildung 6.1 dargestellt, die zugehörigen Graphen der *Korrektklassifizierungsrate* und der *Verlustfunktion* in Abbildung 6.2. Das wichtigste Maß, der *F1-Score*, beträgt im Durchschnitt über die fünf Kategorien 0.95, die *Relevanz* 0.96 und die *Sensitivität* 0.95. Die Konfiguration des Netzes entspricht der Darstellung in Abbildung 5.4 auf der linken Seite.

| **LSTM, Takt, Set 1** | MuR | Ba | Kl | Ro | Rag | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 35 | 0 | 0 | 0 | 0 | 1.00 | 1.00 | 1.00 |
| Barock | 0 | 33 | 2 | 0 | 0 | 0.97 | 0.94 | 0.96 |
| Klassik | 0 | 1 | 33 | 0 | 1 | 0.92 | 0.94 | 0.93 |
| Romantik | 0 | 0 | 1 | 31 | 3 | 1.00 | 0.89 | 0.94 |
| Ragtime | 0 | 0 | 0 | 0 | 35 | 0.90 | 1.00 | 0.95 |

Abbildung 6.1.: Konfusionsmatrix, Relevanz, Sensitivität und F1-Score des besten Ergebnisses in der Netzauswahlphase

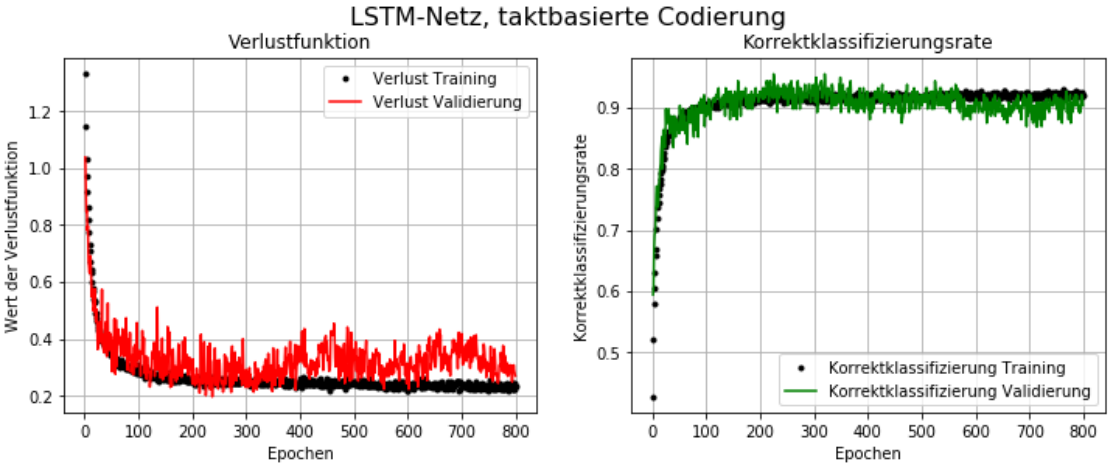



Abbildung 6.2.: Verlustfunktion und Korrektklassifizierungsrate des besten Ergebnisses in der Netzauswahlphase

Es gilt nun, diese Ergebnisse wenn möglich noch etwas zu steigern und im Anschluss sicherzustellen, dass das Netz auf bis zu diesem Zeitpunkt unbekannten Datensätzen ähnlich gute Vorhersagen trifft.

Zunächst wird eine neue Datenmenge zusammengestellt, die jetzt in drei Gruppen geteilt wird: Trainings-, Validierungs- und Testdaten. Im bisherigen Verlauf konnten nicht mehr als 140 Stücke pro Komponist verwendet werden, da von Frédéric Chopin nicht mehr Dateien zur Verfügung standen. Da nun zusätzlich Testdaten benötigt werden, wird Chopin ab hier durch Robert Schumann ersetzt, ebenfalls ein Komponist der Romantik. Die Stücke von Georg Friedrich Händel werden durch Stücke von Johann Sebastian Bach ersetzt, von dem ebenfalls mehr Kompositionen im *MIDI*-Format vorhanden sind. Die übrigen drei Epochen (Mittelalter/Renaissance, Klassik und das 20. Jahrhundert mit Ragtimes) werden demselben Pool entnommen wie während der Auswahl des besten Netzes. Die neue Trainingsmenge enthält damit pro Epoche 105 Stücke, insgesamt also 525 Stücke, die jeweils in alle 12 Tonarten transponiert werden. Die Validierungs- und die Testmenge enthalten je 30 Stücke pro Epoche, also jeweils 150 Stücke.

Werden die neuen Daten nun aufbereitet und in das unveränderte *LSTM*-Netz eingespeist, so ergibt dies auf der Validierungsmenge eine Korrektklassifizierungsrate von 90.0%. Dies liegt zwar unter dem Ergebnis auf der alten Datenmenge, jedoch erscheint es als erstes Ergebnis auf diesen Daten akzeptabel. Daher wird hier direkt die Testmenge hinzugezogen. Hier beträgt die Korrektklassifizierungsrate 90.7%. Da beide Mengen im Vergleich zu anderen Problemen des maschinellen Lernens recht klein sind, lässt sich dieser Unterschied durch die natürliche Streuung leicht erklären.

Da hier neue Daten verarbeitet werden, ist zunächst unklar, ob das Ergebnis von 95.4% aus der Testauswahlphase erreicht oder gar übertroffen werden kann. Jedoch lässt der Verlauf der Kostenfunktion und der Korrektklassifizierungsrate in Abbildung 6.3 im Vergleich zu der in Abbildung 6.2 vermuten, dass es sich zumindest noch steigern lässt. Die Graphen des Trainings und der Validierung liegen in beiden Fällen etwas weiter auseinander als auf der Datenmenge während der Netzauswahl. Dies würde zunächst dafür sprechen, die Dropout-Rate zu erhöhen. Jedoch hätte dies auch einen noch langsameren Anstieg der Korrektklassifizierungsrate auf den Trainingsdaten zur Folge. Nur eine extrem lange Trainingszeit von einigen Tausend Epochen könnte mit dieser Konfiguration noch gute Ergebnisse hervorbringen. Die zweite Möglichkeit wäre, die Anzahl der Einheiten des rekurrenten Netzes zu erhöhen. Auch dies führt zu einer deutlich längeren Trainingszeit. Daher wird zunächst versucht, die Dropout-Rate sogar zu verringern. Zu erwarten ist in diesem Fall etwas mehr *Overfitting*, aber auch eine Korrektklassifizierung auf den Trainingsdaten, die höher liegt als das Wunschergebnis auf den Validierungsdaten.

Tatsächlich führt die Verringerung des Dropout zu einer Verbesserung der Korrektklassifizierung auf 93.3% auf den Validierungsdaten, allerdings auch zu einem größeren Abstand der Verlustfunktionen in Training und Validierung. Die Graphen in Abbildung 6.4 zeigen dies. Eine weitere Verringerung ist daher nicht anzuraten. Sinnvoll erscheint ein Training mit mehr Einheiten in der *LSTM*-Schicht und höherer Dropout-Rate.

Trotz zahlreicher getesteter Konfigurationen konnte jedoch auch mit mehr Einheiten kein insgesamt besseres Ergebnis mehr erzielt werden. Abbildung 6.5 zeigt exemplarisch einen solchen Trainingsverlauf, dessen Ergebnis auf den Validierungsdaten recht deut-

lich hinter dem bisher besten, mit nur 16 Einheiten erzielten Ergebnis zurückbleibt. Nur 90.7% der Stücke werden korrekt klassifiziert. Auf den Testdaten beträgt die Korrektklassifizierungsrate jedoch 95.3%. Dies zeigt insbesondere, dass die Validierungs- und Testmenge für reale Anwendungen hier zu klein sind, denn typischerweise sollte die Leistung auf der Testmenge bestenfalls gleich gut wie die auf der Validierungsmenge sein.

Die Tabellen in den Abbildungen 6.6 und 6.7 präsentieren die besten erzielten Ergebnisse auf dieser Datenmenge. Abbildung 6.8 zeigt auf der linken Seite noch einmal die unveränderte Konfiguration des Netzes aus der Auswahlphase, in der Mitte die erfolgreichste Version mit verringerter Dropout-Rate. Die rechte Seite zeigt die beschriebene Konfiguration mit mehr Einheiten.

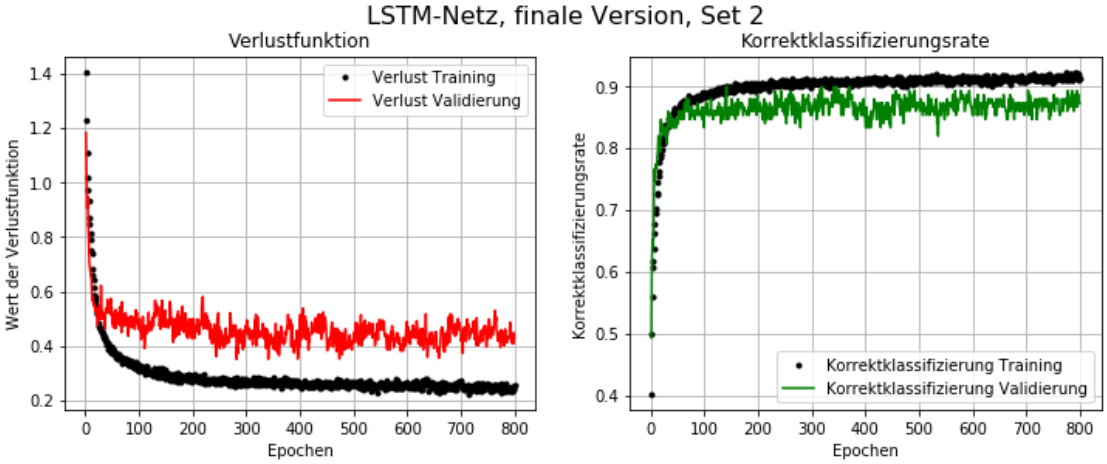



Abbildung 6.3.: Verlustfunktion und Korrektklassifizierungsrate des unveränderten LSTM-Netzes auf den neuen Daten. Korrektklassifizierungsrate 90.0%.

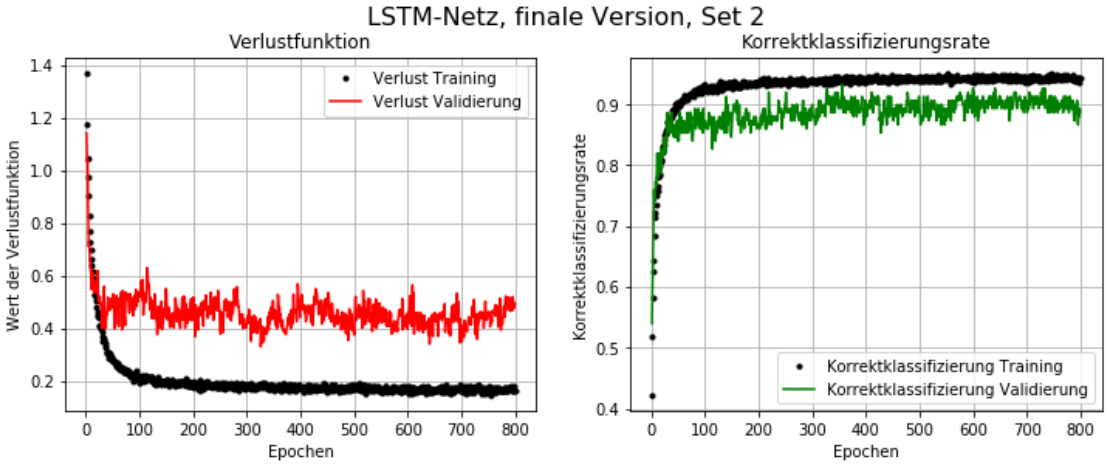



Abbildung 6.4.: Verlustfunktion und Korrektklassifizierungsrate des LSTM-Netzes mit verringertem Dropout auf den neuen Daten. Korrektklassifizierungsrate 93.3% Validierung, 92.7% Test.

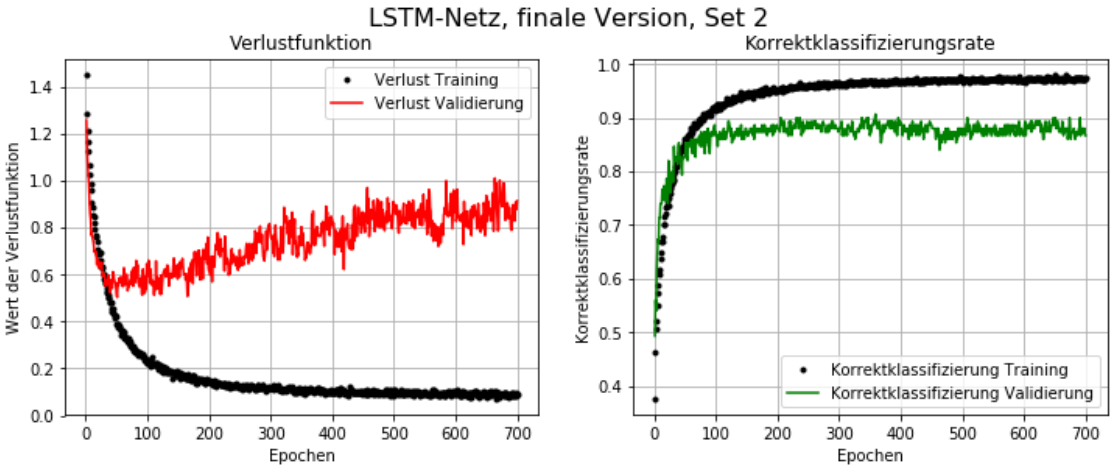



Abbildung 6.5.: Verlustfunktion und Korrektklassifizierungsrate des LSTM-Netzes mit mehr Einheiten auf den neuen Daten. Korrektklassifizierungsrate 90.7% Validierung, 95.3% Test.

| **LSTM, Takt, Set 2** | MuR | Ba | Kl | Ro | Rag | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 29 | 0 | 0 | 1 | 0 | 0.85 | 0.97 | 0.91 |
| Barock | 1 | 28 | 1 | 0 | 0 | 0.97 | 0.93 | 0.95 |
| Klassik | 2 | 0 | 28 | 0 | 0 | 0.97 | 0.93 | 0.95 |
| Romantik | 2 | 1 | 0 | 26 | 1 | 0.93 | 0.87 | 0.90 |
| Ragtime | 0 | 0 | 0 | 1 | 29 | 0.97 | 0.97 | 0.97 |

Abbildung 6.6.: Ergebnisse des LSTM-Netzes auf der Validierungsmenge des neuen Datensatzes. Korrektklassifizierungsrate 93.3%.

| **LSTM, Takt, Set 2** | MuR | Ba | Kl | Ro | Rag | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 27 | 2 | 0 | 1 | 0 | 0.96 | 0.90 | 0.93 |
| Barock | 1 | 25 | 1 | 2 | 1 | 0.86 | 0.83 | 0.85 |
| Klassik | 0 | 0 | 30 | 0 | 0 | 0.97 | 1.0 | 0.98 |
| Romantik | 0 | 2 | 0 | 28 | 0 | 0.88 | 0.93 | 0.90 |
| Ragtime | 0 | 0 | 0 | 1 | 29 | 0.97 | 0.97 | 0.97 |

Abbildung 6.7.: Ergebnisse des LSTM-Netzes auf der Testmenge des neuen Datensatzes. Korrektklassifizierungsrate 92.7%.

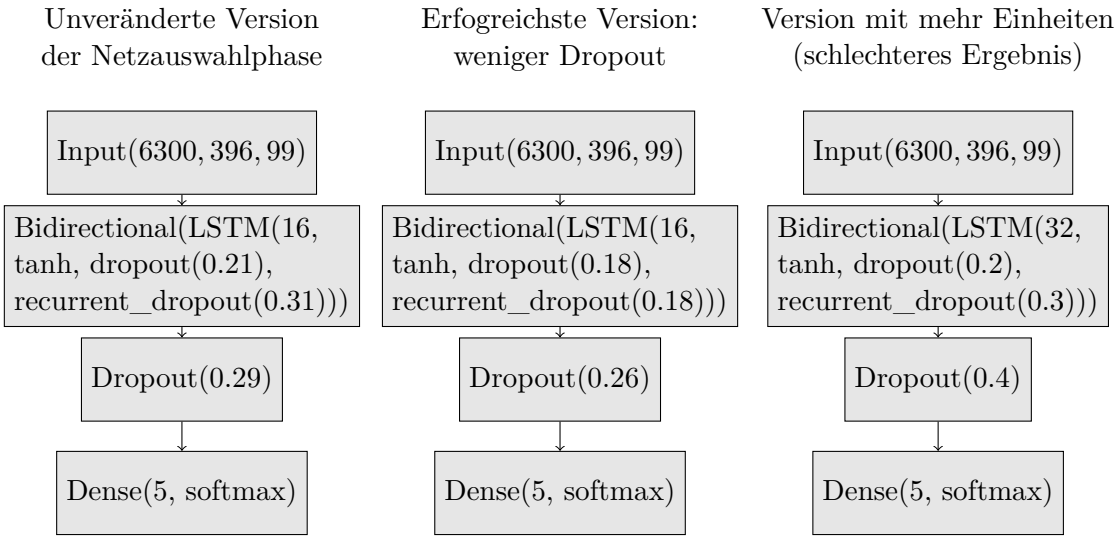



Abbildung 6.8.: Feinabstimmung des LSTM-Netzes

# 7. Leistungen des Netzes auf anderen Datenmengen

Nachdem nun ein Netz zur Verfügung steht, das auf zwei teilweise unterschiedlichen Datenmengen jeweils Korrektklassifizierungsraten von mehr als 90% auf Validierungs- und Testmenge erzielt, wird es hier mit drei schwierigeren Aufgaben konfrontiert.

## 7.1. Unterscheidung dreier Komponisten der Barockzeit

Die drei Hauptvertreter barocker Klaviermusik sind Johann Sebastian Bach, Georg Friedrich Händel und Domenico Scarlatti. Da alle drei im selben Jahr (1685) geboren und in den 1750er Jahren gestorben sind, sollten ihre Werke ähnliche stilistische Merkmale aufweisen. Experten und viele Laien können sie dennoch recht gut voneinander unterscheiden. Interessant ist die Frage, ob auch ein neuronales Netz dazu in der Lage ist. Da die Ähnlichkeit zwischen Kompositionen derselben Epoche größer ist als zwischen Stücken, deren Komposition mehrere Jahrhunderte auseinander liegt, ist ein etwas schlechteres Ergebnis zu erwarten. Da es aber auch nur drei Kategorien gibt, könnte eine Korrektklassifizierungsrate von 90% dennoch erreichbar sein. Die drei Kategorien sind mit jeweils 104 Beispielen in der Trainingsmenge und 25 Beispielen in der Validierungs- und in der Testmenge enthalten.

Die **erste Version** des Netzes verwendet eine sehr niedrige Dropout-Rate, um zunächst schauen zu können, ob die Korrektklassifizierung auf den Trainingsdaten die 95%-Marke überschreiten kann. Bei einer Dropout-Rate von 0.1 auf den Input in einem Zeitschritt, von 0.2 auf die rekurrente Einheit und ebenfalls 0.2 auf den Output wird dieses Ziel zwar nur knapp erreicht, jedoch beträgt auch die Korrektklassifizierungsrate auf den Validierungsdaten direkt 89.3%. Abbildung 7.1 zeigt die zugehörige Konfusionsmatrix. Da dies dem gesteckten Ziel von 90% schon recht nahe kommt, jedoch die Validierungsmenge mit nur 25 Stücken sehr klein ist, wird auch hier direkt die Testmenge hinzugezogen. Hier kann das Ergebnis leider nicht bestätigt werden, es werden nur 73.3% der Stücke dem richtigen Komponisten zugeordnet. Die Konfusionsmatrix in Abbildung 7.2 zeigt dies. Validierungs- und Testmenge wurden offenbar zufällig ungünstig zusammengestellt. Fasst man sie zu einer größeren Validierungsmenge zusammen, so beträgt die Korrektklassifizierungsrate insgesamt 81.3%. Diese Vorgehensweise erscheint hier legitim, da das Netz bisher nicht auf die Validierungsmenge hin optimiert wurde und die beiden Mengen einzeln offenbar nicht repräsentativ sind.

Abbildung 7.3 zeigt den Verlauf des ersten Trainings. Die Schwankungen der Verlustfunktion fallen extrem groß aus, ensprechend auch die der Korrektklassifizierung. Dies

| **LSTM, Takt, Set 3** | Bach | Händel | Scarlatti | prec. | recall | f1-score |
|---|---|---|---|---|---|---|
| Bach | 22 | 1 | 2 | 0.96 | 0.88 | 0.92 |
| Händel | 1 | 22 | 2 | 0.88 | 0.88 | 0.88 |
| Scarlatti | 0 | 2 | 23 | 0.85 | 0.92 | 0.88 |

Abbildung 7.1.: Erste Ergebnisse des LSTM-Netzes auf der Validierungsmenge der Komponisten der Barockzeit. Korrektklassifizierungsrate 89.3%.

| **LSTM, Takt, Set 3** | Bach | Händel | Scarlatti | prec. | recall | f1-score |
|---|---|---|---|---|---|---|
| Bach | 21 | 2 | 2 | 0.75 | 0.84 | 0.79 |
| Händel | 4 | 18 | 3 | 0.69 | 0.72 | 0.71 |
| Scarlatti | 3 | 6 | 16 | 0.76 | 0.64 | 0.70 |

Abbildung 7.2.: Erste Ergebnisse des LSTM-Netzes auf der Testmenge der Komponisten der Barockzeit. Korrektklassifizierungsrate 73.3%.

spricht zunächst für eine zu große Lernrate. Diese wird also in der **zweiten Version** heruntergesetzt, von 0.01 auf 0.005. Zudem wird der Dropout auf den Output leicht erhöht, von 0.2 auf 0.3. Der Verlauf dieses Trainings ist in Abbildung 7.4 zu sehen. Die Amplitude der Graphen ist kleiner, was auf die verringerte Lernrate zurückzuführen ist. Allerdings werden auf den Validierungsdaten nur noch 82.7% der Beispiele korrekt klassifiziert, auf der Testmenge jetzt 76%. Auf den Trainingsdaten werden immer noch über 95% erreicht. Dies spricht dafür, den Dropout noch einmal zu erhöhen. **Version 3** verwendet Raten von 0.15 auf den Dropout eines Zeitschrittes, 0.25 auf den rekurrenten Dropout und 0.32 auf den Output. Auch die Lernrate wird noch einmal herabgesetzt, auf 0.003. Abbildung 7.5 veranschaulicht das Training. Dies sieht schon recht gut aus, jedoch scheint eine weitere Optimierung in dieselbe Richtung noch möglich. Die **endgültige Version** besitzt schließlich *Dropout-Raten* von 0.17, 0.27 und 0.35 und eine *Lernrate* von 0.003. Die *Korrektklassifizierungsrate* beträgt 89.3% auf den Validierungsdaten und 77.3% auf den Testdaten, was insgesamt eine Rate von 83.3% ergibt. Die Abbildungen 7.7 und 7.8 enthalten die genauen Ergebnisse. Eine weitere Steigerung ließe sich eventuell mit noch größerer Epochenzahl erzielen. Abbildung 7.9 gibt einen Überblick über den Verlauf der Feinabstimmung dieses Netzes.

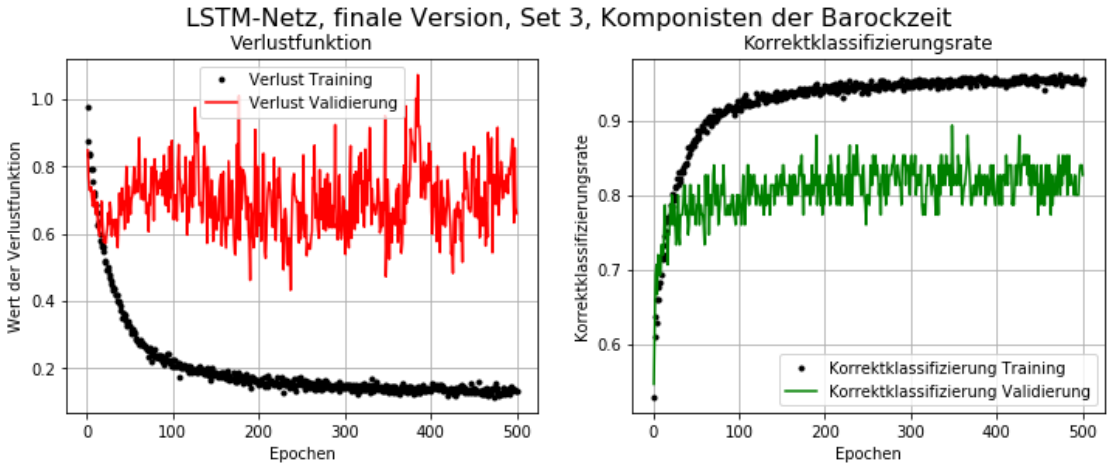



Abbildung 7.3.: Verlustfunktion und Korrektklassifizierungsrate des LSTM-Netzes zur Klassifikation der Komponisten des Barock - erste Version

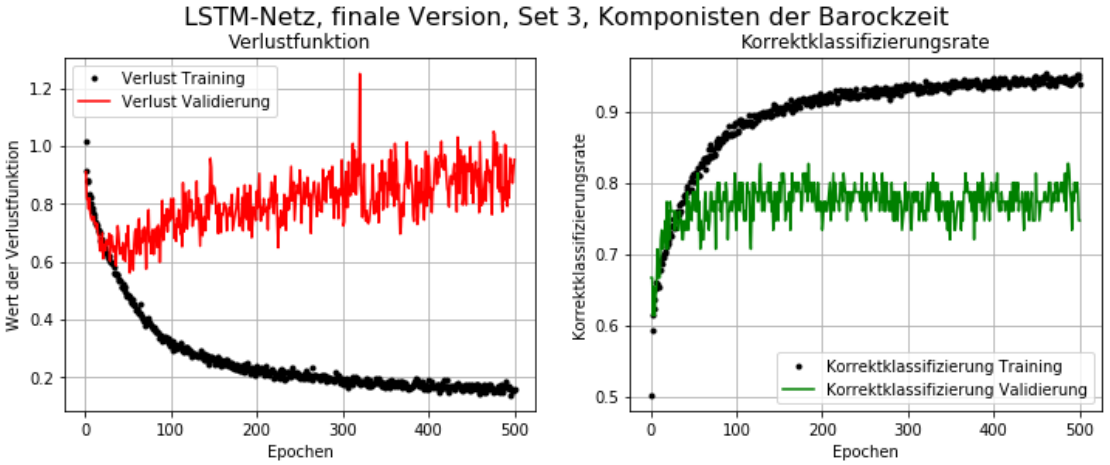



Abbildung 7.4.: Verlustfunktion und Korrektklassifizierungsrate des LSTM-Netzes zur Klassifikation der Komponisten des Barock - Version 2: kleinere Lernrate, etwas mehr Dropout

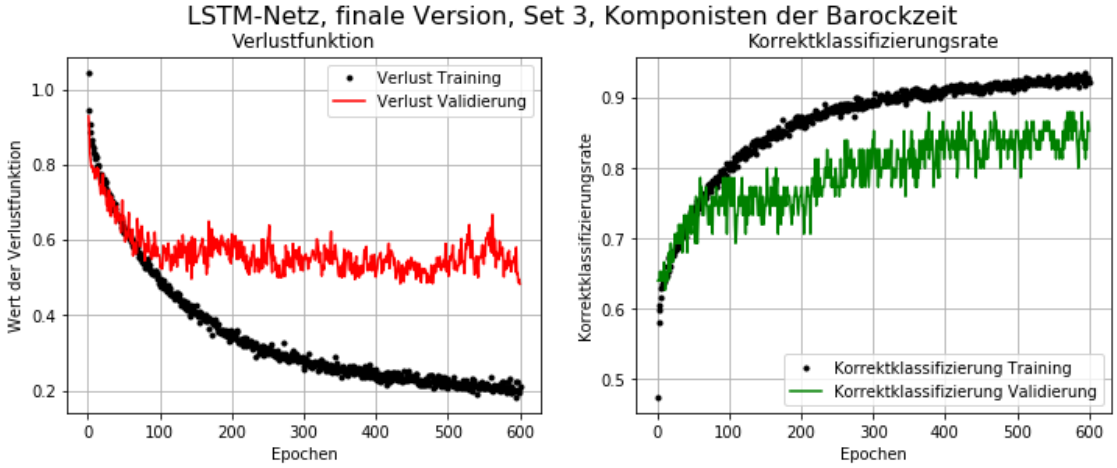



Abbildung 7.5.: Verlustfunktion und Korrektklassifizierungsrate des LSTM-Netzes zur Klassifikation der Komponisten des Barock - Version 3: noch kleinere Lernrate, noch mehr Dropout

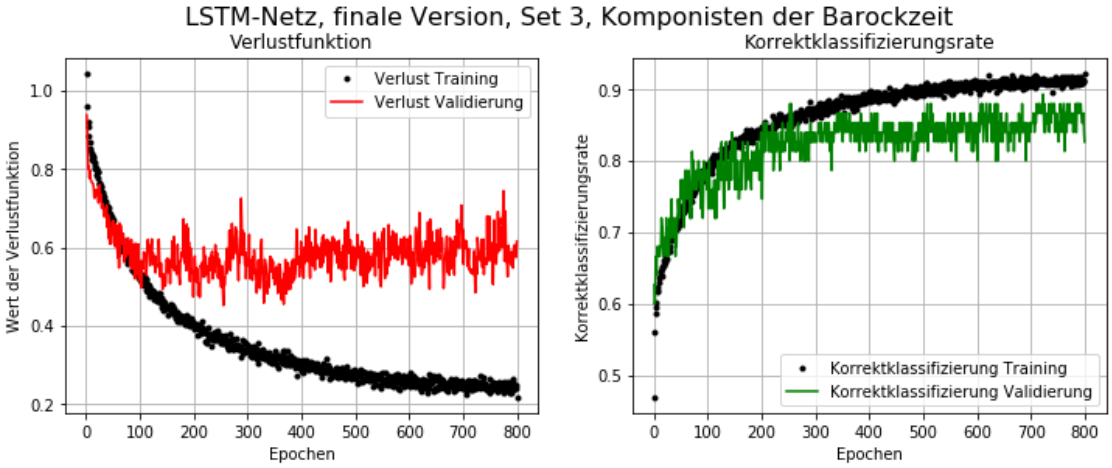



Abbildung 7.6.: Verlustfunktion und Korrektklassifizierungsrate des LSTM-Netzes zur Klassifikation der Komponisten des Barock - endgültige Version: gleiche Lernrate, mehr Dropout

| **LSTM, Takt, Set 3** | Bach | Händel | Scarlatti | prec. | recall | f1-score |
|---|---|---|---|---|---|---|
| Bach | 23 | 1 | 1 | 0.92 | 0.92 | 0.92 |
| Händel | 1 | 22 | 2 | 0.88 | 0.88 | 0.88 |
| Scarlatti | 1 | 2 | 22 | 0.88 | 0.88 | 0.88 |

Abbildung 7.7.: Beste erzielte Ergebnisse des LSTM-Netzes auf der Validierungsmenge der Barock-Komponisten. Korrektklassifizierungsrate 89.3%.

| **LSTM, Takt, Set 3** | Bach | Händel | Scarlatti | prec. | recall | f1-score |
|---|---|---|---|---|---|---|
| Bach | 20 | 3 | 2 | 0.77 | 0.80 | 0.78 |
| Händel | 3 | 20 | 2 | 0.74 | 0.80 | 0.77 |
| Scarlatti | 3 | 4 | 18 | 0.82 | 0.72 | 0.77 |

Abbildung 7.8.: Ergebnisse des LSTM-Netzes auf der Testmenge der Komponisten der Barockzeit. Korrektklassifizierungsrate 77.3%.

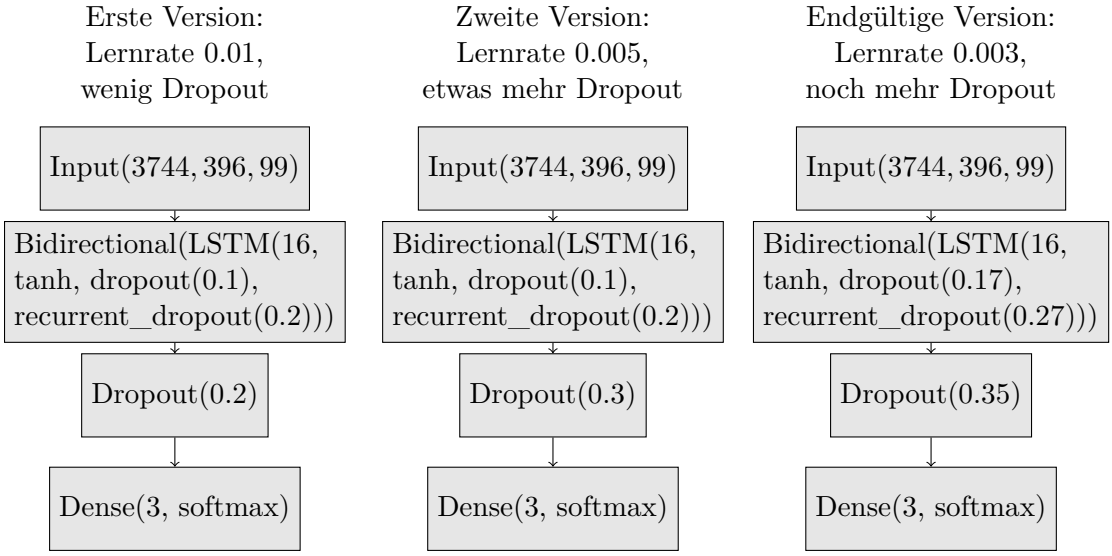



Abbildung 7.9.: Feinabstimmung des LSTM-Netzes auf die Komponisten des Barock

## 7.2. Klassifikation von Stücken diverser Komponisten

### 7.2.1. Klassifikation nach Epoche bei sechs Kategorien

Diese Aufgabe erscheint ähnlich wie die während der Netzauswahl und der Feinabstimmung. Jedoch gibt es nun eine Kategorie mehr, und alle Kategorien enthalten Stücke verschiedener Komponisten. Für die Kategorien *Mittelalter/Renaissance* und *Ragtime* war das auch bisher der Fall. Die Menge der *barocken* Stücke enthält nun zu gleichen Teilen Werke von Bach, Händel und Scarlatti, die Menge der *klassischen* Stücke Werke von Haydn, Mozart und Beethoven. Die *romantischen* Kompositionen stammen von Schubert, Schumann, Mendelssohn, Chopin, Liszt, Brahms, Grieg und Tschaikowski. Da die romantischen Stücke sowohl von Chopin als auch von Schumann ohnehin in *LSTM*-Netzen am häufigsten falsch klassifiziert wurden, ist auch hier mit Problemen zu rechnen. Eine neue Kategorie sind die Stücke des 20. Jahrhunderts, die nicht zu den Ragtimes gehören. Diese Unterscheidung erscheint durchaus sinnvoll, da der Ragtime zu den ersten Ausprägungen des Jazz gehört und wenig mit anderen Kompositionen nach 1900 gemeinsam hat. Die beiden einzigen Vertreter der neuen Epoche sind Alexander Scriabin und Béla Bartók. Beide wurden zwar Ende des 19. Jahrhunderts geboren, ihre Kompositionen sind aber zeitlich und stilistisch dem frühen 20. Jahrhundert zuzuordnen. Jede der sechs Kategorien ist mit 105 Beispielen in der Trainingsmenge und 30 Stücken in der Validierungs- und in der Testmenge enthalten. Ein besseres Ergebnis als 80% ist hier kaum zu erwarten, da zum einen die Zahl der Kategorien größer ist und zum anderen insbesondere die Romantik mit diversen Komponisten vertreten ist, die sich durchaus voneinander unterscheiden.

Abbildung 7.10 skizziert zunächst den Verlauf der im Folgenden beschriebenen Feinabstimmung. In der **ersten Version** entspricht die Konfiguration der des besten Netzes der Netzauswahlphase. Damit kann in 500 Trainings-Epochen eine Korrektklassifizierungsrate von 72.8% auf den Validierungsdaten erzielt werden. Abbildung 7.11 zeigt den recht großen Abstand zwischen den Graphen der Trainings- und Validierungsdaten. Ein höherer Dropout könnte hier Abhilfe schaffen. Jedoch werden nach den 500 Epochen auch in der Trainingsmenge nur knapp 80% der Stücke korrekt klassifiziert. Ein höherer Dropout würde diesen Wert noch einmal verringern, so dass auf der Validierungmenge in jedem Fall weniger als 80% zu erwarten wären.

Um also eine höhere Korrektklassifizierung auf Trainings- und Validierungsdaten zu ermöglichen, wird in der **zweiten Version** die Anzahl der Einheiten von 16 auf 32 erhöht und gleichzeitig der Dropout auf 0.3, 0.4 und 0.5. Dies hat zunächst zur Folge, dass die Korrektklassifizierungsrate auf den Trainingsdaten innerhalb der ersten 240 Epochen kontinuierlich bis etwa 75% ansteigt, auf den Validierungsdaten bis 73.3%. Dieser Aufwärtstrend wird aber plötzlich unterbrochen, es kommt zu einem sehr starken Abfall zwischen den Epochen 300 und 450. Abbildung 7.12 zeigt dieses Phänomen, das für eine zu große Lernrate spricht. Vermutlich wird ein (lokales oder globales) Minimum der Verlustfunktion immer wieder derart übersprungen, dass auf der gegenüberliegenden Seite ein höherer Wert erreicht wird.

Die Lernrate wird also in der **dritten Version** von 0.01 auf 0.003 verringert, unter Beibehaltung des Dropout. Abbildung 7.13 zeigt, dass die Ausschläge der Kurven hier deutlich geringer ausfallen und bis zum Ende ein Trend zur Steigerung sichtbar ist. Das Ergebnis von 73.9% Korrektklassifizierung auf den Validierungsdaten könnte also möglicherweise bei mehr Epochen und evtl. höherer Dropout-Rate noch verbessert werden. Eine **letzte Version** erzielt mit 600 Trainingsepochen und *Dropout-Raten* von 0.32, 0.42 und 0.52 das insgesamt beste Ergebnis mit 76.1% *Korrektklassifizierung* auf den Validierungsdaten und 79.4% auf den Testdaten. Da der entsprechende Wert auf den Trainingsdaten am Ende des Trainings mit ca. 80% nur minimal über der der Testmenge liegt, werden danach keine weiteren Versuche mehr durchgeführt. Der Abstand der Graphen zwischen Trainings- und Validierungsmenge ist in allen Abbildungen recht groß, jedoch zeigt das Ergebnis auf der Testmenge, dass die Validierungsmenge offenbar etwas ungünstig zusammengestellt wurde. Ohnehin sind Validierungs- und Testmenge für reale Anwendungen zu klein, um wirklich verlässliche Ergebnisse zu liefern. In den Abbildungen 7.15 und 7.16 sind die Ergebnisse der besten Konfiguration auf den Validierungs- und Testdaten zu sehen.

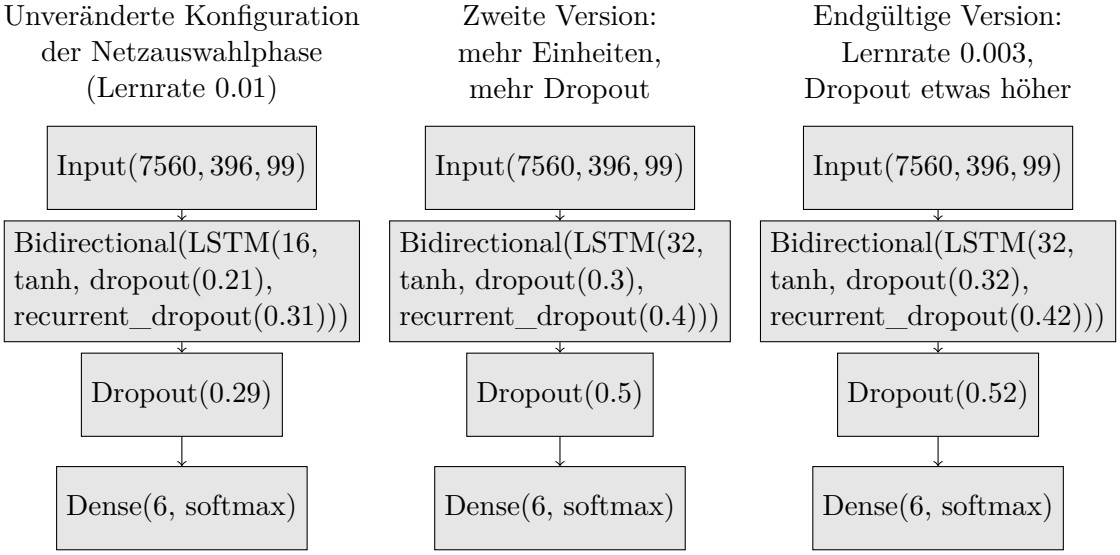



Abbildung 7.10.: Feinabstimmung des LSTM-Netzes auf die Klassifikation diverser Komponisten nach Epoche

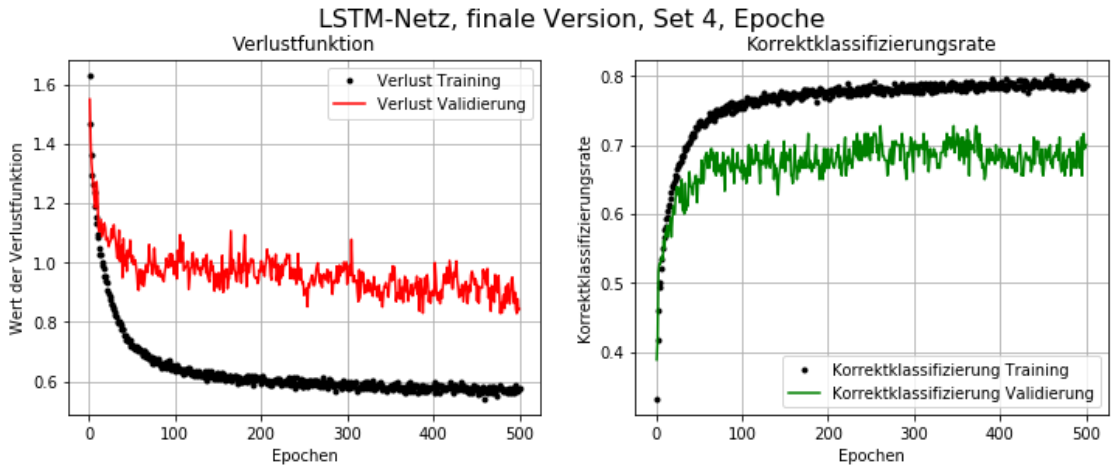



Abbildung 7.11.: Verlustfunktion und Korrektklassifizierungsrate des LSTM-Netzes auf dem Datenset mit diversen Komponisten aus sechs Epochen - erste Version: Korrektklassifizierungsrate 72.8%.

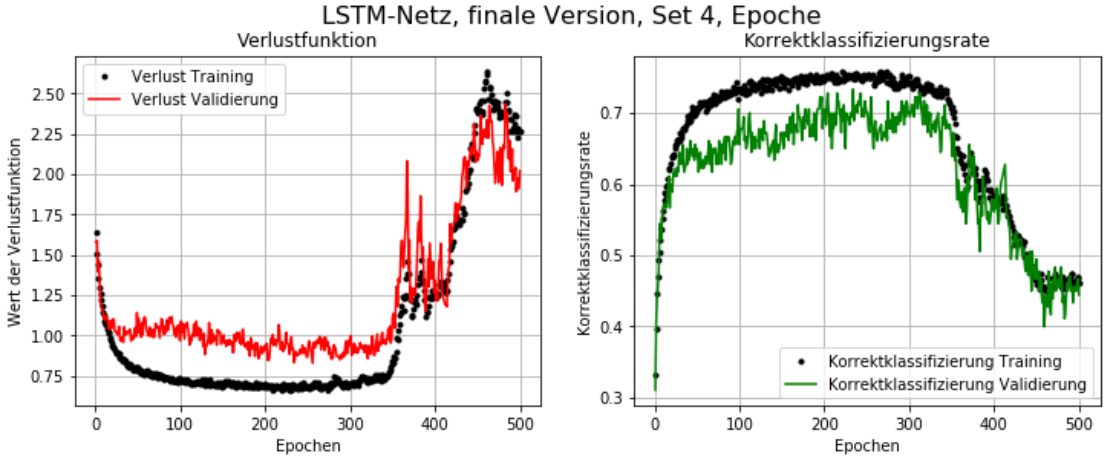



Abbildung 7.12.: Verlustfunktion und Korrektklassifizierungsrate des LSTM-Netzes auf dem Datenset mit diversen Komponisten aus sechs Epochen - zweite Version: mehr Einheiten und mehr Dropout, aber zu große Lernrate. Korrektklassifizierungsrate bis zu 73.3%.

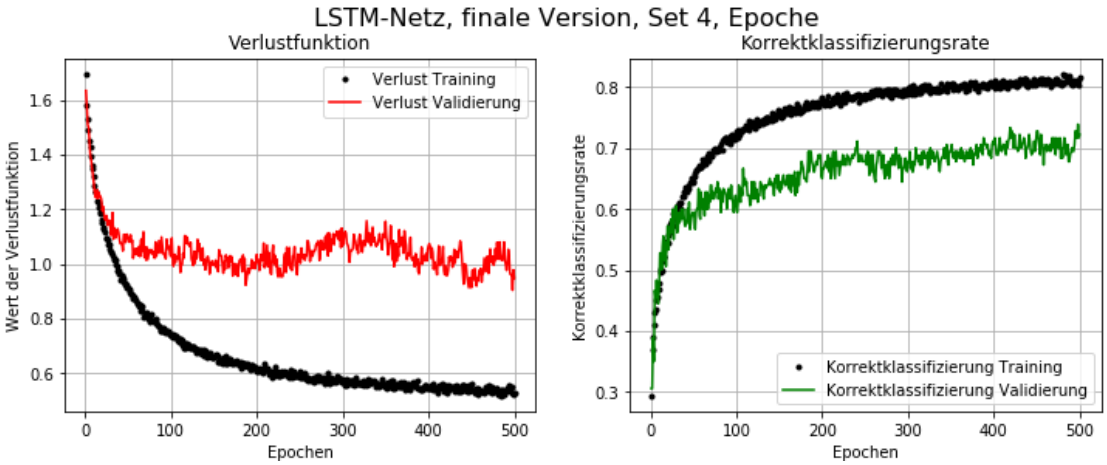



Abbildung 7.13.: Verlustfunktion und Korrektklassifizierungsrate des LSTM-Netzes auf dem Datenset mit diversen Komponisten aus sechs Epochen - dritte Version: kleinere Lernrate, sonst unveränderte Konfiguration. Korrektklassifizierung 73.9% auf Validierungsmenge, 78.9% auf Testmenge.

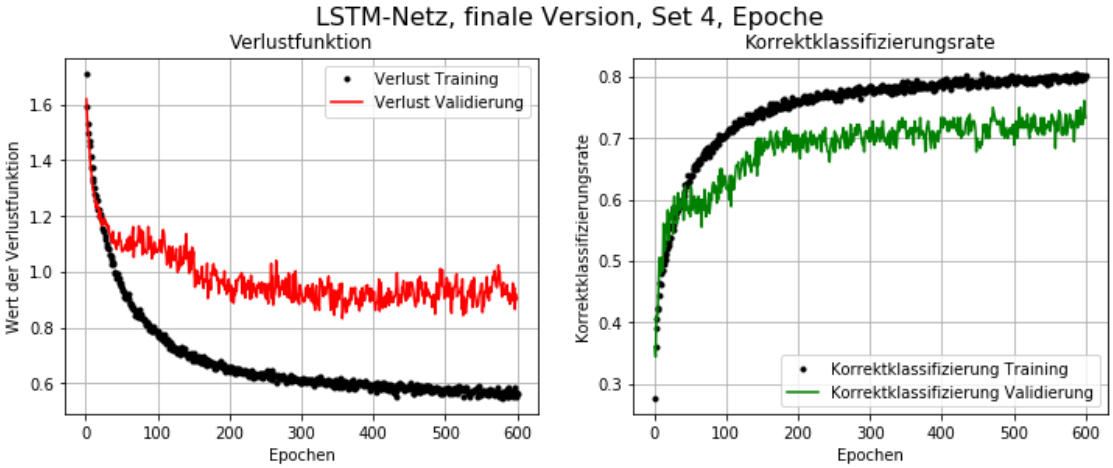



Abbildung 7.14.: Verlustfunktion und Korrektklassifizierungsrate des LSTM-Netzes auf dem Datenset mit diversen Komponisten aus sechs Epochen - vierte Version: mehr Epochen und etwas höherer Dropout. Korrektklassifizierung 76.1% auf Validierungsmenge, 79.4% auf Testmenge.

| **LSTM, Set 4** | MuR | Ba | Kl | Ro | Rag | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|---|
| MuR | 29 | 1 | 0 | 0 | 0 | 0 | 0.85 | 0.97 | 0.91 |
| Barock | 1 | 22 | 5 | 1 | 0 | 1 | 0.73 | 0.73 | 0.73 |
| Klassik | 0 | 4 | 23 | 3 | 0 | 0 | 0.68 | 0.77 | 0.72 |
| Romantik | 2 | 1 | 2 | 16 | 3 | 6 | 0.67 | 0.53 | 0.59 |
| Ragtime | 0 | 0 | 0 | 3 | 27 | 0 | 0.87 | 0.90 | 0.89 |
| 20. Jh | 2 | 2 | 4 | 1 | 1 | 20 | 0.74 | 0.67 | 0.70 |

Abbildung 7.15.: Beste erzielte Ergebnisse des LSTM-Netzes zur Klassifikation diverser Komponisten nach Epoche auf den Validierungsdaten. Korrektklassifizierungsrate 76.1%.

| **LSTM, Set 4** | MuR | Ba | Kl | Ro | Rag | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|---|
| MuR | 29 | 0 | 0 | 0 | 0 | 1 | 0.85 | 0.97 | 0.91 |
| Barock | 2 | 25 | 1 | 1 | 0 | 1 | 0.81 | 0.83 | 0.82 |
| Klassik | 1 | 2 | 23 | 2 | 0 | 2 | 0.82 | 0.77 | 0.79 |
| Romantik | 0 | 2 | 2 | 18 | 3 | 5 | 0.72 | 0.60 | 0.65 |
| Ragtime | 0 | 0 | 0 | 1 | 29 | 0 | 0.85 | 0.97 | 0.91 |
| 20. Jh | 2 | 2 | 2 | 3 | 2 | 19 | 0.68 | 0.63 | 0.66 |

Abbildung 7.16.: Beste erzielte Ergebnisse des LSTM-Netzes zur Klassifikation diverser Komponisten nach Epoche auf den Testdaten. Korrektklassifizierungsrate 79.4%.

### 7.2.2. Klassifikation nach Komponist bei 15 Kategorien

Eine Möglichkeit, auf dieser Bachelorarbeit aufzubauen, wäre die Entwicklung eines neuronalen Netzes, das Stücke aller bekannten Komponisten mit einer Korrektklassifizierungsrate von 95% richtig zuordnet. Dazu wäre zum einen eine wesentlich größere Trainingsmenge notwendig, zum anderen aber auch mindestens ein leistungsstarker Rechner mit *NVIDIA*-Grafikkarte, besser noch ein Rechnerverbund. Die einzelnen Stücke könnten in Abschnitte von jeweils 32 Takten segmentiert werden, sodass die Anzahl der Trainingsbeispiele noch einmal vervielfacht würde. Sehr hilfreich wäre zudem der Aufbau einer Datenbank, die nur auf ihre Richtigkeit überprüfte *MIDI*-Dateien enthielte. Nur dann könnten die 32 Takte tatsächlich musikalisch sinnvolle Einheiten umfassen.

Um eine Idee davon zu bekommen, wie weit das hier entwickelte Netz von diesem Ziel entfernt ist, wird es nun an einer Datenmenge überprüft, die Stücke aus 15 Kategorien enthält. Die Stücke aus dem Mittelalter und die Ragtimes bilden nach wie vor eigene Kategorien, alle weiteren Kategorien enthalten nur jeweils Stücke eines einzelnen Komponisten. Die Auswahl wurde derart vorgenommen, dass aus jeder Kategorie mindestens 55 Stücke verfügbar sind, von denen 40 zum Training und 15 zur Validierung dienen. Eine Testmenge gibt es hier nicht. Zum einen sind beide Mengen ohnehin zu klein, um ein in der Praxis taugliches Ergebnis liefern zu können, zum anderen interessiert hier lediglich die Größenordnung der Korrektklassifizierung. Aus der Zeit des Barock sind wieder Bach, Händel und Scarlatti enthalten, aus der Klassik nun neben Haydn, Mozart und Beethoven auch Muzio Clementi. Die Epoche der Romantik wird nur noch durch Schubert, Schumann, Chopin und Brahms vertreten, da von den übrigen Komponisten aus Datenset 4 zu wenige Stücke zur Verfügung stehen. Aus dem frühen 20. Jahrhundert stammen die Werke wieder von Bartók und Scriabin. Bei nur 40 Stücken pro Komponist in der Trainingsmenge erscheint es von Vornherein schwierig, hier zusätzlich viele unterschiedliche Gattungen zu berücksichtigen. Daher wurde bei den Komponisten, von denen wesentlich mehr als 55 Stücke zur Verfügung stehen, bewusst eine Auswahl getroffen. Dies betrifft folgende Kategorien:

- **Johann Sebastian Bach**: nur Präludien (keine Fugen, keine Suiten)
- **Wolfgang Amadeus Mozart** und **Ludwig van Beethoven**: überwiegend Sonaten (keine Variationen)
- **Franz Schubert**: vorwiegend Impromptus, Moments Musicaux und Sonaten (keine Walzer)
- **Robert Schumann**: nur Albumblätter, Novelletten, Bunte Blätter, Papillons und Waldszenen (keine Walzer, keine Stücke aus dem Album für die Jugend oder aus den Kinderszenen, ...)
- **Frédéric Chopin**: nur Mazurken, Nocturnes und Polonaisen (keine Etüden, keine Préludes)
- **Béla Bartók**: keine Klavierstücke für Kinder

In allen anderen Kategorien wurden die Stücke rein zufällig aus den vorhandenen ausgewählt.

Anvisiert ist ein Prozentsatz der Korrektklassifizierung, der zumindest signifikant über dem Erwartungswert bei zufälligem Raten (6.7%) liegen sollte. Da auch hier alle Kategorien die gleiche Mächtigkeit besitzen, genügt die Korrektklassifizierungsrate als Metrik. Zu erwarten sind folgende Probleme:

- Komponisten derselben Epoche lassen sich schwieriger voneinander unterscheiden. Für die Zeit des Barock wurde dies bereits gezeigt, für die Klassik und die Romantik gilt es voraussichtlich in verstärktem Maße.
- Auf Datenset 4 wurde bereits ersichtlich, dass die Stücke aus den Epochen der Romantik und des frühen 20. Jahrhunderts häufig der jeweils anderen zugeordnet werden. Da hier vier Komponisten der Romantik und zwei des 20. Jahrhunderts vertreten sind, werden diese voraussichtlich die schlechtesten Ergebnisse erzielen.

Da zunächst unklar ist, welche Änderungen die völlig neue Struktur der Datenmenge am Netz notwendig macht, wird die Konfiguration des besten Netzes der Auswahlphase übernommen. Das bidirektionale *LSTM*-Netz hat 16 Einheiten, der Dropout auf den Input eines Zeitschrittes beträgt 0.21, der auf die rekurrente Einheit 0.31, der Dropout auf den Output 0.29, die Lernrate 0.01.

Das erste Ergebnis ist bereits unerwartet gut. 61.3% der Stücke werden korrekt klassifiziert. Abbildung 7.17 zeigt den Verlauf der Verlustfunktion und der Korrektklassifizierungsrate im Laufe des Trainings. Die Ergebnisse sind in Abbildung 7.18 zu sehen.

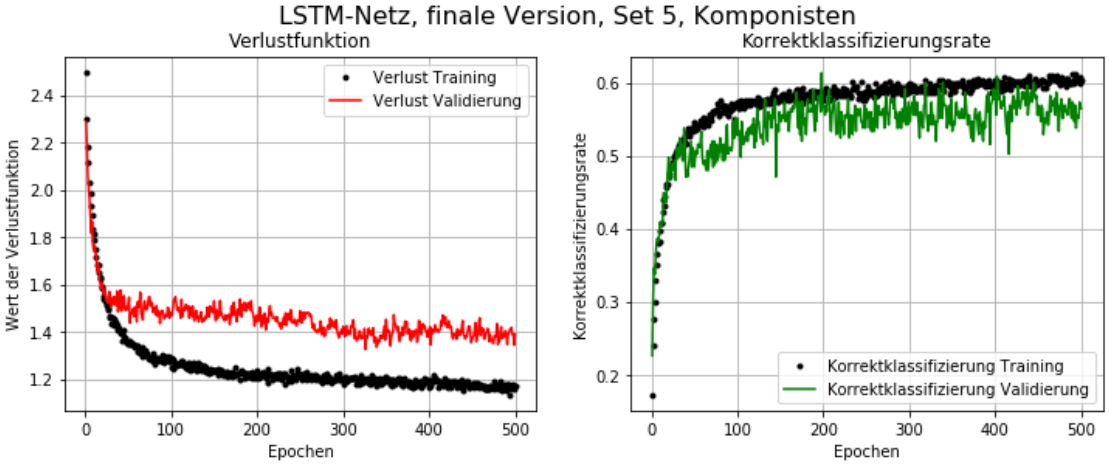



Abbildung 7.17.: Verlustfunktion und Korrektklassifizierungsrate des LSTM-Netzes auf dem Datenset mit 15 Kategorien: Korrektklassifizierung 61.3%.

| **Komp** | MR | Ba | Hä | Sc | Ha | Cl | Mo | Be | Sb | Sm | Ch | Br | Ra | Scr | Bar |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MR | **13** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Ba | 0 | **16** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Hä | 0 | 1 | **11** | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Sc | 0 | 0 | 0 | **10** | 3 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| Ha | 0 | 0 | 2 | 1 | **7** | 1 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Cl | 0 | 1 | 0 | 5 | 1 | **3** | 1 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| Mo | 0 | 0 | 2 | 0 | 3 | 1 | **8** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Be | 0 | 0 | 1 | 0 | 1 | 0 | 2 | **7** | 1 | 2 | 0 | 0 | 0 | 0 | 1 |
| Sb | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | **8** | 1 | 1 | 1 | 1 | 1 | 0 |
| Sm | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | **10** | 0 | 0 | 0 | 1 | 0 |
| Ch | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 1 | **10** | 1 | 0 | 0 | 0 |
| Br | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | **10** | 1 | 0 | 0 |
| Rag | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | **14** | 0 | 0 |
| Scr | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 1 | 3 | 0 | **6** | 1 |
| Bar | 3 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 1 | 0 | 1 | 1 | **5** |

| | prec. | recall | f1-score |
|---|---|---|---|
| MA/Ren. | 0.76 | 0.87 | 0.81 |
| Bach | 0.89 | 1.0 | 0.94 |
| Händel | 0.61 | 0.73 | 0.67 |
| Scarlatti | 0.56 | 0.67 | 0.61 |
| Haydn | 0.44 | 0.50 | 0.47 |
| Clementi | 0.60 | 0.20 | 0.30 |
| Mozart | 0.50 | 0.53 | 0.52 |
| Beethoven | 0.54 | 0.47 | 0.50 |
| Schubert | 0.47 | 0.53 | 0.50 |
| Schumann | 0.53 | 0.67 | 0.59 |
| Chopin | 0.59 | 0.67 | 0.62 |
| Brahms | 0.67 | 0.67 | 0.67 |
| Ragtime | 0.82 | 0.93 | 0.87 |
| Scriabin | 0.67 | 0.40 | 0.50 |
| Bartók | 0.50 | 0.33 | 0.40 |

Abbildung 7.18.: Beste erzielte Ergebnisse des LSTM-Netzes zur Klassifikation nach Komponist bei 15 Kategorien. Korrektklassifizierungsrate 61.3%.

Dass das erste der beiden erwarteten Probleme tatsächlich eingetreten ist, wird in der Konfusionsmatrix ersichtlich, die sich ergibt, wenn die Komponisten ihren Epochen zugeordnet werden. In Abbildung 7.18 oben ist dies bereits durch doppelte Linien angedeutet. Werden die Werte innerhalb dieser größeren Einheiten addiert, so ergibt sich die Matrix in Abbildung 7.19. Die *Korrektklassifizierungsrate* steigt auf 76%.

Um das zweite Problem herauszurechnen, muss die Kategorie des frühen 20. Jahrhunderts entfernt werden. In der Tat sind die Werte hier am schlechtesten, und auch die *Relevanz* (*precision*) derjenigen Kategorien wird verringert, in die diese Stücke falsch einsortiert werden. Die *Relevanz* ohne das 20. Jahrhundert ist leicht zu berechnen. Die *Sensitivität* kann nur in einem Intervall „[*von ... bis ...*]“ angegeben werden, da ihr Wert davon abhängt, ob die fälschlicherweise ins 20. Jahrhundert eingeordneten Werke ohne diese Kategorie richtig klassifiziert worden wären. Es ergeben sich die in Abbildung 7.20 eingetragenen Qualitätsmaße.

| **LSTM, Set 5** | MuR | Ba | Kl | Ro | Rag | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|---|
| MuR | 13 | 0 | 1 | 0 | 0 | 1 | 0.76 | 0.87 | 0.81 |
| Barock | 0 | 40 | 4 | 1 | 0 | 1 | 0.74 | 0.87 | 0.80 |
| Klassik | 0 | 12 | 40 | 5 | 0 | 2 | 0.80 | 0.68 | 0.74 |
| Romantik | 1 | 1 | 3 | 51 | 2 | 2 | 0.75 | 0.85 | 0.80 |
| Ragtime | 0 | 0 | 0 | 1 | 14 | 0 | 0.82 | 0.93 | 0.87 |
| 20. Jh | 3 | 1 | 2 | 10 | 1 | 13 | 0.68 | 0.43 | 0.53 |

Abbildung 7.19.: Auf sechs Epochen umgerechnetes Ergebnis des LSTM-Netzes zur Klassifikation nach 15 Komponisten. Korrektklassifizierungsrate 76.0%. Durchschnittlicher F1-Score ebenfalls 0.76.

| **LSTM, Set 5** | prec. | recall | f1-score |
|---|---|---|---|
| MuR | 0.93 | 0.87 bis 0.93 | 0.81 bis 0.93 |
| Barock | 0.75 | 0.87 bis 0.89 | 0.80 bis 0.81 |
| Klassik | 0.83 | 0.68 bis 0.71 | 0.74 bis 0.76 |
| Romantik | 0.88 | 0.85 bis 0.88 | 0.80 bis 0.88 |
| Ragtime | 0.86 | 0.93 | 0.87 |

Abbildung 7.20.: Auf fünf Epochen (ohne 20. Jahrhundert) umgerechnetes Ergebnis des LSTM-Netzes zur Klassifikation nach 15 Komponisten. Korrektklassifizierungsrate 81.0 bis 84.1%. F1-Score 0.80 bis 0.85.

In Anbetracht der Tatsache, dass jede der 15 Kategorien mit nur 40 Beispielen in der Trainingsmenge vertreten war, ist dies doch ein unerwartet gutes Ergebnis. Mehrere Versuche, es durch Verringern der Dropout-Rate und der Lernrate noch zu verbessern, scheiterten. Möglicherweise wären leichte Verbesserungen noch durch mehr Einheiten im Netz zu erreichen. In jedem Fall motivieren das Ergebnis und die daraus gewonnenen Erkenntnisse zu dem Versuch, am Ende doch noch ein Netz zu erstellen, das bei mehreren Komponisten pro Epoche einen *F1-Score* von annähernd 0.9 erzielt.

### 7.2.3. Finales Netz: Klassifikation nach Epoche bei diversen Komponisten pro Epoche

Letztes Ziel ist es nun, ein Netz zu konstruieren, das Stücke diverser Komponisten in die richtige Epoche einordnet. Hierzu fließen alle gewonnenen Erkenntnisse ein, bezüglich der Frage, welche Komponisten und Gattungen sich für solch ein eher einfaches *LSTM*-Netz eignen. Mit den Kategorien *Mittelalter/Renaissance* und *Ragtime* gab es ohnehin bisher keine Probleme. Im Zeitalter des *Barock* wurden vorwiegend polyphone Werke falsch eingeordnet, daher fallen sie hier heraus. In der *Klassik* wurden die meisten Stücke von Clementi dem Barock zugeordnet, in der *Romantik* erzielten die Werke von Schubert die schlechtesten Ergebnisse. Die Kategorie des *frühen 20. Jahrhunderts* (ohne die Ragtimes) ließ sich insgesamt sehr schlecht klassifizieren, was insbesondere auch den Wert der *Relevanz* der anderen Kategorien verschlechterte. Die neue Datenmenge umfasst daher nur noch die ursprünglichen fünf Kategorien und enthält 165 Werke aus dem Mittelalter und der Renaissance, ebenfalls 165 Ragtimes und je 55 Stücke von Bach, Händel, Scarlatti, Haydn, Mozart, Beethoven, Schumann, Chopin und Brahms. Pro 55 Stücke werden 40 zum Training und 15 zur Validierung verwendet. Eine Testmenge wird nicht angelegt.

Auch dieses Netz wird zunächst so konfiguriert wie das beste Netz der Auswahlphase. Damit erfüllt es mit 89.3% korrekt klassifizierter Beispiele aus der Validierungsmenge bereits die Erwartungen. Die Plots in Abbildung 7.21 zeigen allerdings, dass hier sogar mehr Beispiele richtig klassifiziert werden als auf den Trainingsdaten. Um einen entsprechend hohen Wert auf den Trainingsdaten zu erhalten, so dass die Möglichkeit besteht, das Ergebnis noch einmal zu verbessern, kann der Dropout etwas erniedrigt werden, auf 0.19 (ein Zeitschritt), 0.29 (rekurrent) und 0.27 (Output). Abbildung 7.22 zeigt den Verlauf des Trainings.

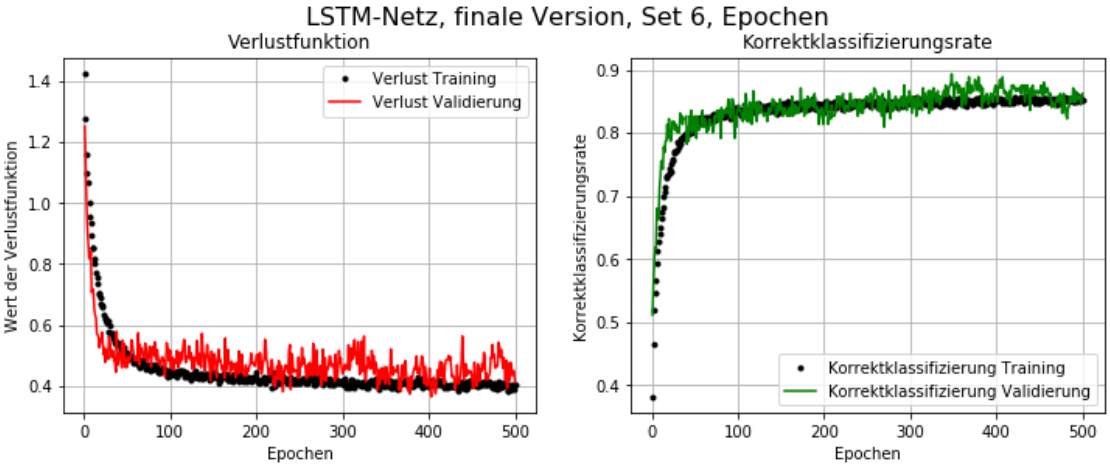



Abbildung 7.21.: Verlustfunktion und Korrektklassifizierungsrate des finalen LSTM-Netzes - erste Version: Korrektklassifizierung 89.3%.

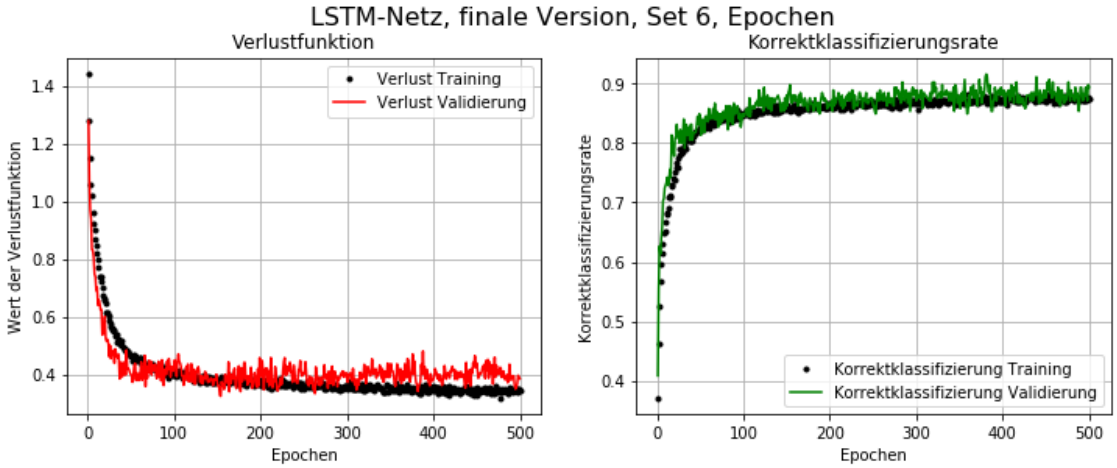



Abbildung 7.22.: Verlustfunktion und Korrektklassifizierungsrate des finalen LSTM-Netzes - zweite Version mit etwas weniger Dropout: Korrektklassifizierung 91.6%.

In der Tat verbessert sich die Korrektklassifizierungsrate auf 91.6%, jedoch scheint der Dropout noch immer zu hoch zu sein, denn im Training werden nicht mehr als 88% der Beispiele korrekt klassifiziert. Der Dropout wird also noch einmal erniedrigt, auf schließlich 0.16, 0.26 und 0.24. Dies führt zwar auf den Trainingsdaten erstmals zu einer Korrektklassifizierung von knapp über 90%, auf den Validierungsdaten werden allerdings nur noch 88.4% erreicht. Abbildung 7.23 zeigt den Trainingsverlauf.

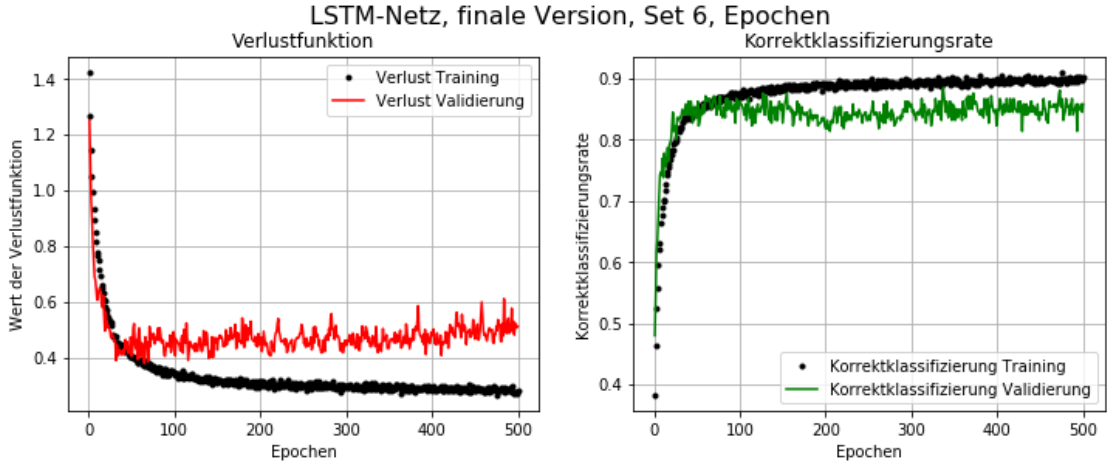



Abbildung 7.23.: Verlustfunktion und Korrektklassifizierungsrate des finalen LSTM-Netzes - dritte Version: noch weniger Dropout, schlechteres Ergebnis. Korrektklassifizierungsrate 88.4%.

Ein besseres Ergebnis als die mit der zweiten Version erzielten 91.6% Korrektklassifi-

zierung ist nur bei mehr Einheiten denkbar. Daher wird die Zahl der Einheiten auf 32 verdoppelt, die *Dropout-Raten* werden auf 0.2 (ein Zeitschritt), 0.3 (rekurrent) und 0.4 (Output) gesetzt. Der Trainingsverlauf in Abbildung 7.24 zeigt, dass die die *Korrektklassifizierung* im Training hier sogar auf mehr als 94% steigt, in der Validierungsmenge werden bis zu 92.9% der Beispiele der richtigen Kategorie zugeordnet. In Abbildung 7.25 ist die zugehörige Konfusionsmatrix zu sehen.

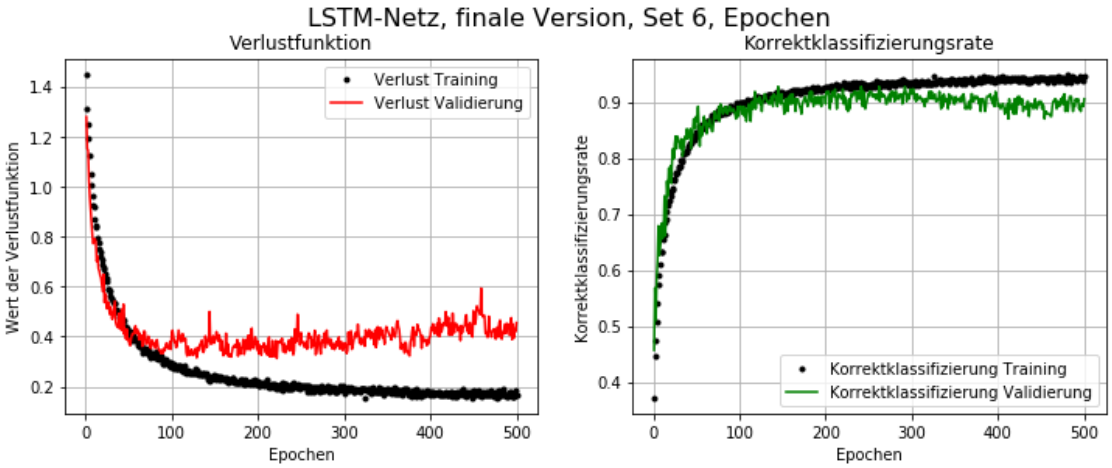



Abbildung 7.24.: Verlustfunktion und Korrektklassifizierungsrate des finalen LSTM-Netzes - endgültige Version: mehr Einheiten, Dropout wieder leicht erhöht. Korrektklassifizierungsrate 92.9%.

| **LSTM, Takt, Set 6** | MuR | Ba | Kl | Ro | Rag | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 45 | 0 | 0 | 0 | 0 | 0.98 | 1.00 | 0.99 |
| Barock | 0 | 40 | 5 | 0 | 0 | 0.91 | 0.89 | 0.90 |
| Klassik | 1 | 1 | 41 | 2 | 0 | 0.89 | 0.91 | 0.90 |
| Romantik | 0 | 3 | 0 | 38 | 4 | 0.95 | 0.84 | 0.89 |
| Ragtime | 0 | 0 | 0 | 0 | 45 | 0.92 | 1.00 | 0.96 |

Abbildung 7.25.: Ergebnisse des finalen LSTM-Netzes zur Klassifikation nach fünf Kategorien mit jeweils mehreren Komponisten pro Epoche. Korrektklassifizierungsrate 92.9%.

# 8. Einige Schlussfolgerungen bezüglich Ähnlichkeit verschiedener Komponisten und Epochen

In den vergangenen drei Kapiteln wurde die Klassifikation von Stücken zahlreicher Komponisten aus insgesamt sechs Epochen untersucht, in Kapitel 5 mit verschiedenen Typen neuronaler Netze, danach nur noch mit einem *LSTM*-Netz. Daraus lassen sich schon einige Schlussfolgerungen ziehen, die allerdings mit etwas Vorsicht zu betrachten sind. Zum einen sind die *MIDI*-Dateien von unterschiedlicher Qualität, und nicht alle Stücke konnten dahingehend untersucht werden. Zum anderen ist die Datenmenge im Vergleich zu anderen Problemen des maschinellen Lernens recht klein. Gäbe es eine Datenbank, die einen Großteil der klassischen Klaviermusik als qualitativ hochwertige *MIDI*-Dateien enthielte, so könnten sicherlich allgemeingültigere Aussagen getroffen werden. Dennoch sollen hier einige Auffälligkeiten diskutiert werden.

## 8.1. Schlussfolgerungen aus der Netzauswahlphase

In der Netzauswahlphase wurden neun verschiedene Netztypen auf jeweils derselben Datenmenge getestet. Enthalten waren diverse Komponisten aus der Zeit des Mittelalters und der Renaissance sowie Ragtimes verschiedener Komponisten. Die Epoche des Barock wurde durch Georg Friedrich Händel vertreten, die Wiener Klassik durch Joseph Haydn und die Romantik durch Frédéric Chopin. Mit fast allen Netztypen wurden die früheste und späteste Epoche am besten erkannt, obwohl gerade diese Kategorien durch mehrere Komponisten vertreten waren. Das *LSTM*-Netz mit taktabhängiger Codierung klassifizierte in seiner erfolgreichsten Konfiguration in der Validierungsmenge alle 35 Stücke dieser Epochen korrekt, die Kategorie *Mittelalter/Renaissance* erreichte sogar einen *F1-Score* von 1.0. Eine mögliche Erklärung dafür ist die Tatsache, dass aufeinanderfolgende Epochen sich offensichtlich ähnlicher sind als weiter auseinander liegende, und dass die erste und letzte Epoche nur jeweils einen „Nachbarn“ besitzen. Eine zweite mögliche Erklärung ist die, dass die *MIDI*-Dateien dieser Gruppen von jeweils einer einzigen Website heruntergeladen wurden und alle Stichproben von sehr hoher Qualität sind.

Die größten Probleme in den beiden insgesamt besten Netzen (takt- und schlagabhängige Version des *LSTM*-Netzes) bereitete die Zuordnung der romantischen Stücke. Abzulesen ist dies an der *Sensitivität*, die nur dort unter der 90%-Marke lag. Da die falsch klassifizierten romantischen Werke am häufigsten der Gruppe der Ragtimes zugeordnet wurden, liegt dort die Relevanz entsprechend niedriger. Aus musiktheoretischer Sicht

ist das Problem nachvollziehbar: Jeder Takt wird in nur 12 Zeitintervalle gegliedert, es werden also höchstens 12 verschiedene Klänge pro Takt erfasst. Gerade in der Romantik gibt es aber häufig erheblich mehr Töne pro Takt, die sehr schnell aufeinanderfolgen. Der Versuch, mit einer feineren Granularität zu codieren, brachte allerdings keine besseren Ergebnisse, weil dadurch das Problem der Überanpassung vergrößert wurde. Um dem entgegenzuwirken, müsste die Anzahl der Sequenzen in der Trainingsmenge vervielfacht werden. Dies konnte im Rahmen der vorliegenden Thesis nicht durchgeführt werden.

Die Tabelle in Abbildung 8.1 vergleicht die *F1-Scores* aller Netztypen und Epochen.

| **F1-Scores** | MA/Ren. | Barock | Klassik | Romantik | Ragtime | Durchschn. |
|---|---|---|---|---|---|---|
| FF-Takt | 0.83 | 0.71 | 0.69 | 0.85 | 0.94 | 0.81 |
| FF-Schlag | 0.89 | 0.82 | 0.74 | 0.77 | 0.92 | 0.83 |
| FF-Zeit | 0.79 | 0.78 | 0.68 | 0.75 | 0.76 | 0.75 |
| LSTM-Takt | **1.00** | **0.96** | 0.93 | **0.94** | 0.95 | **0.95** |
| LSTM-Schlag | 0.97 | 0.93 | **0.97** | 0.90 | 0.94 | 0.94 |
| LSTM-Zeit | 0.99 | 0.91 | 0.91 | 0.93 | 0.96 | 0.94 |
| CLSTM-Takt | 0.94 | 0.86 | 0.94 | 0.92 | **1.00** | 0.93 |
| CLSTM-Schlag | 0.94 | 0.88 | 0.89 | 0.90 | 0.99 | 0.92 |
| CLSTM-Zeit | 0.94 | 0.86 | 0.91 | **0.94** | 0.97 | 0.93 |

Abbildung 8.1.: Vergleich der F1-Scores aller Netztypen und Epochen. („FF“ steht für „Feedforward Netz“.)

## 8.2. Schlussfolgerungen aus den Leistungen des LSTM-Netzes auf verschiedenen Datenmengen

### Datenmenge 2: Fünf Epochen

Hierbei handelt es sich um die Datenmenge, die zur Feinabstimmung des *LSTM*-Netzes diente. Anstelle von Händel wurde die Barockzeit hier durch Johann Sebastian Bach vertreten, die Romantik durch Robert Schumann. Das Ergebnis von 95.4% konnte nicht ganz bestätigt werden, was offenbar auf die beiden ausgetauschten Komponisten zurückzuführen ist. Die *F1-Scores* der Kategorien *Barock* und *Romantik* weichen von den in der Netzauswahlphase erzielten Werten am stärksten nach unten ab. Von Bach wurden insbesondere Sätze aus der *Englischen Suite* und die *Fugen* schlecht erkannt.

### Datenmenge 3: Komponisten der Barockzeit

Auf der Datenmenge mit Stücken dreier Komponisten der Barockzeit fielen die Ergebnisse schlechter aus als erwartet. Sicherlich ähneln sich Kompositionen aus derselben Zeit mehr als solche, deren Entstehungszeit weiter auseinander liegt. Dennoch sind gerade

diese drei Komponisten für Experten recht gut zu unterscheiden. Aus diesem Grund lohnt sich auch hier ein Blick auf die genauen Stücke, deren Label falsch zugeordnet wurde. Es fällt auf, dass dies im Falle von Bach ausschließlich *polyphone* Werke und Sätze aus der *Englischen Suite* sind. Alle *Präludien* wurden korrekt klassifiziert. Die *MIDI*-Dateien der falsch klassifizierten Stücke sind zum großen Teil von guter Qualität, jedoch enthalten die Sätze der Suite sehr viele ausgeschriebene *Verzierungen. Triller* können in *MIDI* etwa nicht durch „*tr*“ gekennzeichnet werden, statt dessen werden sie als sehr kleine Notenwerte codiert. Dies kann aufgrund der recht groben Granularität der Merkmalscodierung nicht korrekt erfasst werden. Die Falschklassifizierung der *Fugen* ist vermutlich auf die Polyphonie zurückzuführen.

#### Datenmenge 4: Sechs Epochen, die jeweils Stücke mehrerer Komponisten enthalten

Die vierte Datenmenge enthält Stücke vieler verschiedener Komponisten, die ihrer Epoche zuzuordnen waren. Das frühe 20. Jahrhundert kam als neue Epoche hinzu. Hier fällt zunächst ins Auge, dass die Kategorien der Romantik und des 20. Jahrhunderts ähnlich schlechte Werte aufweisen, was offenbar daran liegt, dass die Stücke der Romantik häufig ins 20. Jahrhundert eingeordnet wurden und die des 20. Jahrhunderts in alle anderen. Fasst man die Werte der Validierungs- und Testmenge aus den Abbildungen 7.15 und 7.16 zusammen, so ergibt sich die Konfusionsmatrix in Abbildung 8.2. Die größten Probleme insgesamt brachten die Stücke von Béla Bartók mit sich. 15 von 22 dieser Werke wurden falsch klassifiziert, wobei alle Epochen enthalten waren. Auch hier liegt die Ursache nicht in fehlerhaft codierten *MIDI*-Dateien, sondern eher darin, dass die Stücke von Bartók sehr „heterogen“ sind. Der sogenannte „*Mikrokosmos*“ dient als Klavierschule für Kinder und enthält sowohl Anklänge aus ungarischer Volksmusik als auch polyphone Stücke. Ebenso finden sich bei Bartók noch Einflüsse aus der Romantik und dem Impressionismus, aber auch sehr dissonante Stücke, in denen das Klavier eher als Schlagzeug denn als Melodieinstrument verwendet wird. Bei nur wenigen Trainingsbeispielen sind daher die Probleme bei der Klassifikation im Nachhinein nicht verwunderlich. Die falsch klassifizierten Stücke der Kategorien des Barock, der Klassik und der Ragtimes wurden dagegen überwiegend ihren „Nachbarepochen“ zugeordnet.

| **LSTM, Set 4** | MuR | Ba | Kl | Ro | Rag | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|---|
| MuR | 58 | 1 | 0 | 0 | 0 | 1 | 0.85 | 0.97 | 0.91 |
| Barock | 3 | 47 | 6 | 2 | 0 | 2 | 0.77 | 0.78 | 0.77 |
| Klassik | 1 | 6 | 46 | 5 | 0 | 2 | 0.74 | 0.77 | 0.75 |
| Romantik | 2 | 3 | 4 | 34 | 6 | 11 | 0.69 | 0.57 | 0.62 |
| Ragtime | 0 | 0 | 0 | 4 | 56 | 0 | 0.86 | 0.93 | 0.89 |
| 20. Jh | 4 | 4 | 6 | 4 | 3 | 37 | 0.71 | 0.65 | 0.68 |

Abbildung 8.2.: Zusammengefasste Konfusionsmatrix und F1-Scores der Klassifikation mit sechs Kategorien. Korrektklassifizierungsrate 77.8%.

### Datenmenge 5: Klassifikation nach Komponist, 15 Kategorien

Die Klassifikation nach 15 verschiedenen Kategorien, von denen 13 nur Stücke jeweils eines einzigen Komponisten enthielten, brachte ein besseres Ergebnis als erwartet. Zudem wurde hier sehr konkret deutlich, welche Komponisten sich besonders ähnlich und welche ohnehin schwer zu klassifizieren sind. Große Ähnlichkeit besteht grundsätzlich zwischen Komponisten derselben Epoche. Dies war so zu erwarten. Die Komponisten, die zudem die größten Probleme bereiteten, waren wiederum die des 20. Jahrhunderts, und außerdem Muzio Clementi, der der Klassik zuzuordnen ist. Die Stücke Clementis wurden jedoch häufiger Scarlatti zugeordnet als korrekt klassifiziert. In der Tat besteht hier eine gewisse Ähnlichkeit. Im Falle des 20. Jahrhunderts können die Fehler auch darin begründet sein, dass es nur noch wenige Regeln gab, an die sich Komponisten zu halten hatten. In Kompositionen dieser Zeit gibt es zusammengesetzte Taktarten, beliebig viele kleinere Notenwerte pro Schlag, beliebige Akkordfolgen, beliebige Dissonanzen usw. Es ist schwierig, hier Gemeinsamkeiten zu finden. Insbesondere ist es schwierig, solche Stücke von denen aus der Romantik zu unterscheiden, da auch dort ein Schlag in beliebig viele kleine Notenwerte aufgeteilt werden kann. Da die Granularität der Quantisierung mit nur 12 Zeiteinheiten pro Takt verhältnismäßig grob ist, kann hier nur sehr ungenau codiert werden, so dass auch in Stücken der Romantik Dissonanzen codiert werden, wo tatsächlich keine sind. Eine feinere Granularität wurde jedoch ohne besseres Ergebnis getestet. Das ohnehin große Problem des *Overfitting* verstärkte sich dadurch noch einmal. Zudem verlängerte sich die Trainingszeit auf mehrere Tage, weswegen nicht viele solcher Versuche möglich waren. Hier könnte nur ein Rechner mit höherer Rechen- und Speicherkapazität Abhilfe schaffen.

### Datenmenge 6: Klassifikation nach Epoche, 5 Kategorien

Hier wurden alle gewonnenen Erkenntnisse eingebracht, um einen Klassifizierer zu erhalten, der Stücke verschiedener Komponisten mit annähernd 90% Trefferwahrscheinlichkeit in die richtige Epoche einordnet. Die Tatsache, dass diese Erwartung sogar übertroffen wurde, zeigt, dass die Schlussfolgerungen bezüglich der Ähnlichkeit und Erkennbarkeit der verschiedenen Komponisten prinzipiell richtig sind. Zusammengefasst betrifft dies folgende Punkte:

- Komponisten derselben Epoche sind sich so ähnlich, dass bei einer Klassifikation nach Komponist bei mindestens 15 Kategorien mit den vorhandenen Kapazitäten keine wesentlich bessere Korrektklassifizierungsrate als 60% zu erzielen ist.
- Polyphone Werke werden oft falsch klassifiziert. Insbesondere betrifft dies die Fugen von Johann Sebastian Bach.
- Stücke von Muzio Clementi werden oft fälschlicherweise Domenico Scarlatti zugeordnet.
- Stücke von Béla Bartók können sehr schlecht identifiziert werden. Nur etwa ein

Drittel der Stücke wird korrekt klassifiziert, die übrigen in beliebige andere Epochen eingeordnet.

- Die Stücke von Alexander Scriabin werden häufig der Romantik zugeordnet und umgekehrt.

Aus diesen Gründen klassifiziert das letzte neuronale Netz wieder nur die ursprünglichen fünf Epochen, von denen jede aber nun Stücke mehrerer Komponisten enthält. Von Johann Sebastian Bach sind ausschließlich Präludien in der Datenmenge enthalten, und auch die übrigen erkannten Fehlerquellen wurden nach Möglichkeit ausgeschaltet. Das Training auf dieser Menge lieferte einen Klassifizierer, der 92.9% der Stücke aus der Validierungsmenge korrekt ihrer Epoche zuordnete.

# 9. Fazit und Ausblick

Ziel dieser Bachelorarbeit war die Erstellung und Konfiguration eines neuronalen Netzes, das klassische Musikstücke mit akzeptabler Zuverlässigkeit ihrer Epoche oder ihrem Komponisten zuordnet. Die erste Herausforderung war die Aufbereitung der Daten zur Eingabe ins Netz. Hier wurden drei verschiedene Versionen implementiert, die auf den unterschiedlichen Netztypen unterschiedlich gut abschnitten. Mit allen Versionen konnten jedoch auf reinen *LSTM*-Netzen und solchen mit vorgeschalteten konvolutionalen Schichten Ergebnisse von mehr als 90% Korrektklassifizierungsrate auf den Validierungsdaten erzielt werden. Als bestes Netz kristallisierte sich schließlich ein bidirektionales *LSTM*-Netz heraus, insbesondere in Kombination mit einer taktbasierten Merkmalscodierung. Hier konnte in der Netzauswahlphase eine Korrektklassifizierungsrate von 95.4% auf den Valdierungsdaten erzielt werden.

Dieses Netz wurde ausgewählt, in Phase zwei anhand einer modifizierten Datenmenge neu trainiert und mit Hilfe einer Testmenge überprüft. Hier konnte das gute Ergebnis nicht ganz bestätigt werden, nur 93% der Stücke wurden korrekt klassifiziert. Der Unterschied ist darauf zurückzuführen, dass insbesondere polyphone Werke von Johann Sebastian Bach und schlecht codierte *MIDI*-Dateien von Kompositionen Robert Schumanns schlechter zugeordnet werden konnten als die Werke von Händel und Chopin aus der ersten Datenmenge.

In Phase drei wurde schließlich das fertige Netz an anderen Aufgabenstellungen getestet, wobei die Konfiguration jeweils angepasst wurde. Die Werke der Barockzeit konnten in 83.3% der Fälle korrekt ihren Komponisten zugeordnet werden, was bei nur drei Kategorien nicht ganz den Erwartungen entsprach. Wieder lag das Problem bei den polyphonen Werken Johann Sebastian Bachs, jedoch wurden auch viele Stücke von Händel und Scarlatti falsch klassifiziert. Offenbar sind Komponisten derselben Epoche schwieriger zu unterscheiden als Komponisten verschiedener Epochen.

Eine weitere Datenmenge umfasste Stücke vieler verschiedener Komponisten, die jeweils einer von sechs Epochen zuzuordnen waren. Dies gelang bei 79.4% der Stücke einer Testmenge. Die romantischen Stücke und die des frühen 20. Jahrhunderts wurden am häufigsten falsch klassifiziert.

In der ursprünglich letzten Aufgabe wurde das Netz darauf trainiert, bei 15 Kategorien nach Komponist zu klassifizieren. Hier fiel das Ergebnis mit 61.3% Korrektklassifizierung deutlich besser aus als erwartet, insbesondere, da weniger Trainingsbeispiele pro Kategorie zur Verfügung standen als in den übrigen Datenmengen. Es fiel auf, dass die meisten Falschklassifizierungen auf eine von zwei Ursachen zurückzuführen waren, nämlich zum einen die Verwechslungen innerhalb der einzelnen Epochen und zum anderen die schlechte Identifizierbarkeit der Werke des frühen 20. Jahrhunderts.

Um nun ein finales Netz zu erhalten, das Werke verschiedener Komponisten mit möglichst gutem Ergebnis ihren Epochen zuordnet, wurden die erkannten Fehlerquellen beseitigt und ein Netz mit neuer Trainings- und Validierungsmenge erstellt, welches die ursprünglichen fünf Kategorien unterscheidet, diesmal jedoch mit Stücken mindestens drei verschiedener Komponisten in jeder Kategorie. Dies gelang mit 92.9% korrekt klassifizierten Beispielen in der Validierungsmenge.

Die erzielten Ergebnisse zeigen, dass sich bereits mit der Rechen- und Speicherkapazität eines durchschnittlichen Rechners ohne Zugriff auf eine hochwertige, zum maschinellen Lernen geeignete Grafikkarte ein neuronales Netz trainieren lässt, das Stücke verschiedener Komponisten mit hoher Trefferwahrscheinlichkeit ihrer Epoche zuordnet. Ausgespart werden muss dabei allerdings die Epoche des 20. Jahrhunderts, und die meisten Fehler treten bei Stücken aus dem 19. Jahrhundert auf. Um auch solche Werke richtig einzuordnen, wären mehr Merkmale und deutlich mehr Trainingsbeispiele notwendig, die zudem auf ihre Richtigkeit hin überprüft werden müssten. Es gibt nur wenige romantische und moderne Klavierwerke im *MIDI*-Format, die den Ansprüchen genügen. Ein großer Teil des Aufwands würde daher darauf entfallen, entsprechende Dateien selbst zu editieren. Die Menge der Trainingsbeispiele könnte zudem vervielfacht werden, indem jedes Stück in 32-taktige Segmente geteilt würde. Dann könnte der enorme Rechenaufwand allerdings nur noch mit Zugriff auf eine Grafikkarte bewältigt werden. Dies würde ein Training mit kleinerer Lernrate und über mehrere Tausend Epochen ermöglichen.

Die Tatsache, dass auch mit der zeitbasierten Merkmalscodierung sehr gute Ergebnisse zu erzielen sind, zeigt zudem, dass auch Audiodateien zur Eingabe in solch ein Netz aufbereitet werden können, die ja keine Informationen zur Taktart enthalten. Die hier erzielten Ergebnisse liegen zwar um wenige Prozentpunkte unter denen mit musikalisch sinnvoller Quantisierung, jedoch dürfte das bei deutlich feinerer Granulariät weniger ins Gewicht fallen.

In jedem Fall konnte gezeigt werden, dass der reine Notentext eines Stückes, ohne jeden Ausdruck, Angaben zur Lautstärke oder zum Tempo, einem *LSTM*-Netz genügt, um mit hoher Trefferwahrscheinlichkeit die Epoche der Komposition vorherzusagen. Bei ausreichend Trainingsdaten sollte in den meisten Fällen auch die Vorhersage des Komponisten gut möglich sein.

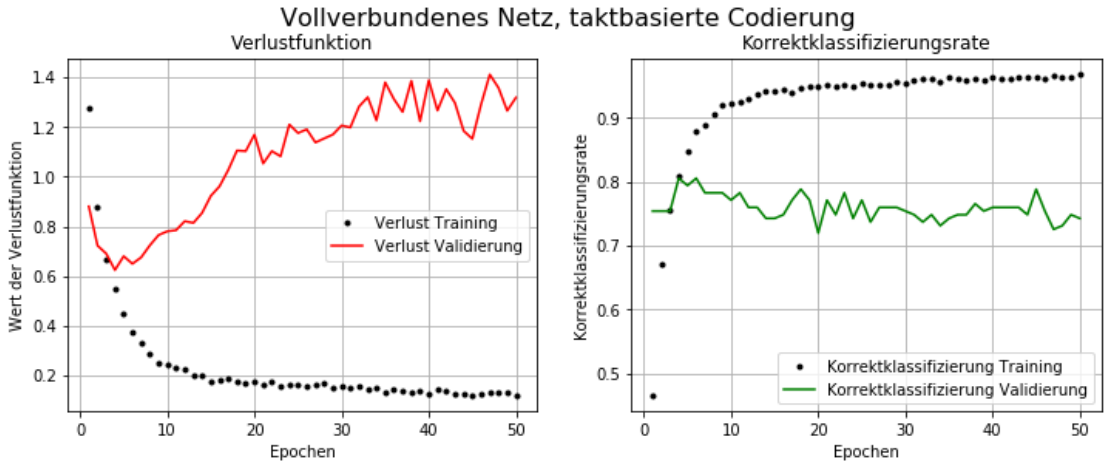
# A. Ergebnisse der am besten konfigurierten Netze jedes Typs

## A.1. Phase 1: Netzauswahl, Datenset 1

### Vollverbundenes Netz, taktabhängige Version

Korrektklassifizierungsrate: 80.6%
Relevanz: 0.81
Sensitivität: 0.81
F1-Score: 0.81

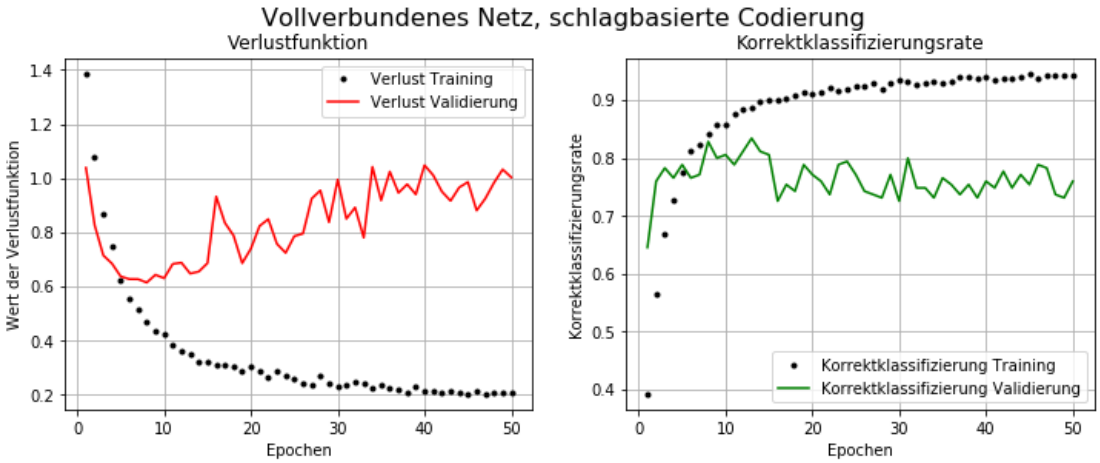
| **Feedforward, Takt** | MuR | Ba | Kl | Ro | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 29 | 2 | 4 | 0 | 0 | 0.83 | 0.83 | 0.83 |
| Barock | 2 | 25 | 7 | 1 | 0 | 0.71 | 0.71 | 0.71 |
| Klassik | 3 | 5 | 25 | 1 | 1 | 0.68 | 0.71 | 0.69 |
| Romantik | 1 | 3 | 1 | 28 | 2 | 0.90 | 0.80 | 0.85 |
| Ragtime | 0 | 0 | 0 | 1 | 34 | 0.92 | 0.97 | 0.94 |



### Vollverbundenes Netz, schlagabhängige Version

Korrektklassifizierungsrate: 83.4%
Relevanz: 0.83
Sensitivität: 0.83
F1-Score: 0.83

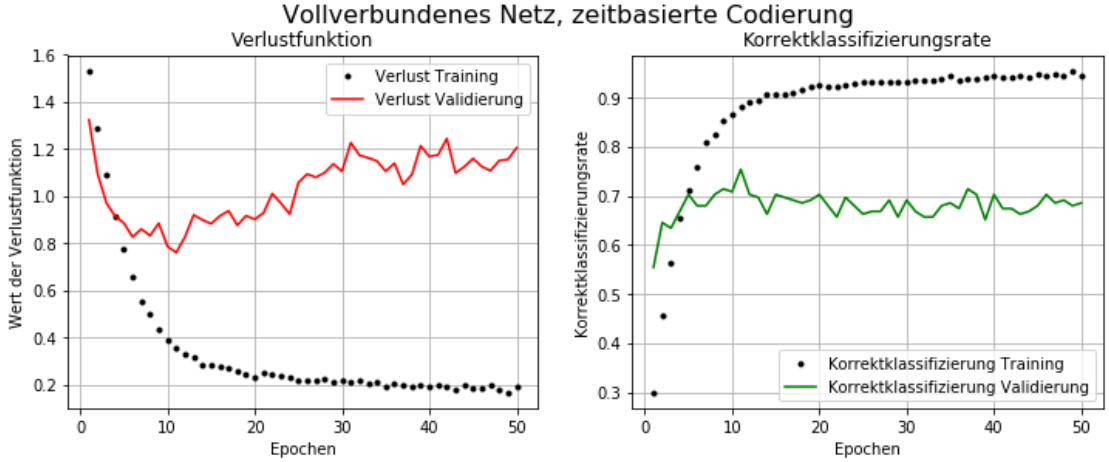
| **Feedforward, Schlag** | MuR | Ba | Kl | Ro | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 33 | 0 | 2 | 0 | 0 | 0.85 | 0.94 | 0.89 |
| Barock | 3 | 30 | 1 | 0 | 1 | 0.79 | 0.86 | 0.82 |
| Klassik | 1 | 6 | 23 | 5 | 0 | 0.85 | 0.66 | 0.74 |
| Romantik | 2 | 2 | 1 | 25 | 5 | 0.83 | 0.71 | 0.77 |
| Ragtime | 0 | 0 | 0 | 0 | 35 | 0.85 | 1.00 | 0.92 |




### Vollverbundenes Netz, zeitabhängige Version

Korrektklassifizierungsrate: 75.4%
Relevanz: 0.76
Sensitivität: 0.75
F1-Score: 0.75

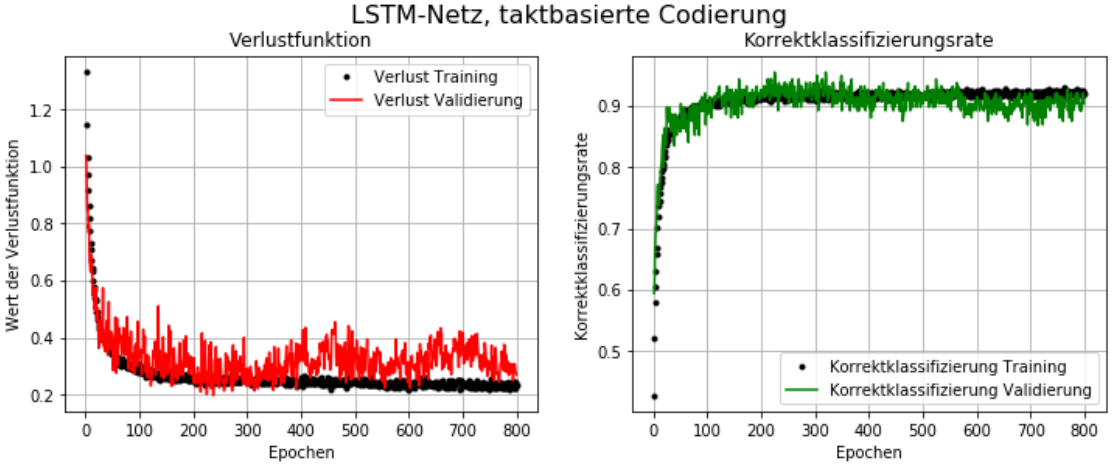
| **Feedforward, Zeit** | MuR | Ba | Kl | Ro | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 32 | 2 | 1 | 0 | 0 | 0.70 | 0.91 | 0.79 |
| Barock | 3 | 28 | 2 | 1 | 1 | 0.76 | 0.80 | 0.78 |
| Klassik | 7 | 6 | 20 | 1 | 1 | 0.83 | 0.57 | 0.68 |
| Romantik | 2 | 0 | 1 | 25 | 7 | 0.78 | 0.71 | 0.75 |
| Ragtime | 2 | 1 | 0 | 5 | 27 | 0.75 | 0.77 | 0.76 |




## LSTM-Netz, taktabhängige Version

Korrektklassifizierungsrate: 95.4%
Relevanz: 0.96
Sensitivität: 0.95
F1-Score: 0.95

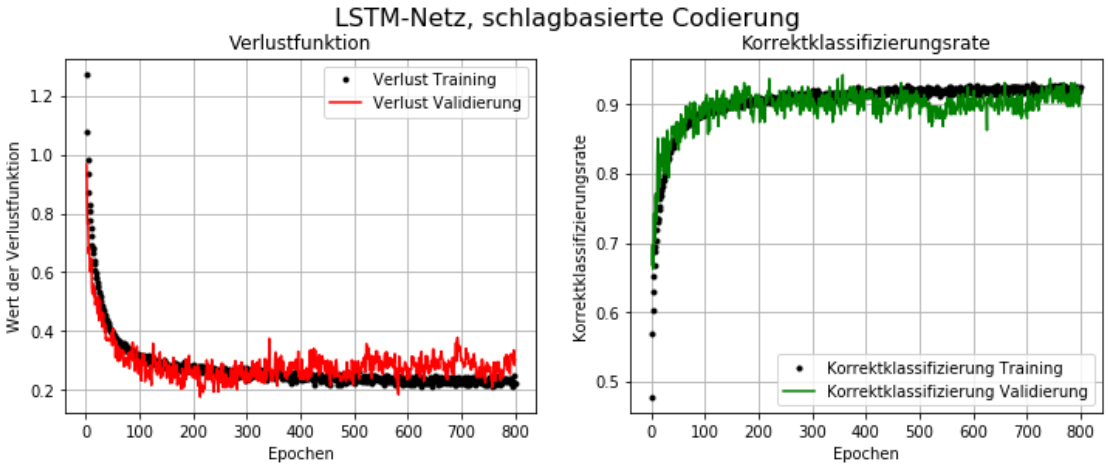
| **LSTM, Takt, Set 1** | MuR | Ba | Kl | Ro | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 35 | 0 | 0 | 0 | 0 | 1.00 | 1.00 | 1.00 |
| Barock | 0 | 33 | 2 | 0 | 0 | 0.97 | 0.94 | 0.96 |
| Klassik | 0 | 1 | 33 | 0 | 1 | 0.92 | 0.94 | 0.93 |
| Romantik | 0 | 0 | 1 | 31 | 3 | 1.00 | 0.89 | 0.94 |
| Ragtime | 0 | 0 | 0 | 0 | 35 | 0.90 | 1.00 | 0.95 |



## LSTM-Netz, schlagabhängige Version

Korrektklassifizierungsrate: 94.3%
Relevanz: 0.94
Sensitivität: 0.94
F1-Score: 0.94

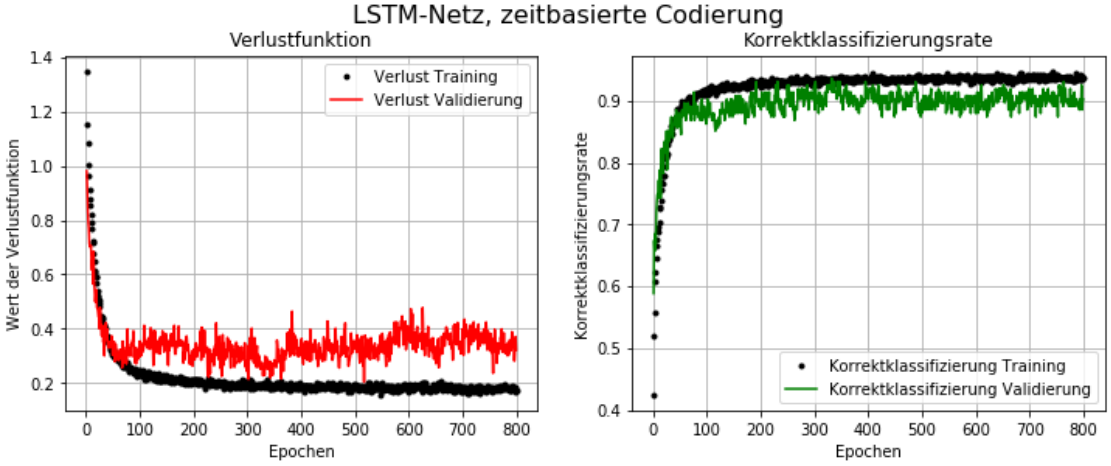
| **LSTM, Schlag, Set 1** | MuR | Ba | Kl | Ro | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 34 | 1 | 0 | 0 | 0 | 0.97 | 0.97 | 0.97 |
| Barock | 1 | 33 | 1 | 0 | 0 | 0.92 | 0.94 | 0.93 |
| Klassik | 0 | 0 | 34 | 1 | 0 | 0.97 | 0.97 | 0.97 |
| Romantik | 0 | 2 | 0 | 30 | 3 | 0.94 | 0.86 | 0.90 |
| Ragtime | 0 | 0 | 0 | 1 | 34 | 0.92 | 0.97 | 0.94 |




## LSTM-Netz, zeitabhängige Version

Korrektklassifizierungsrate: 93.7%
Relevanz: 0.94
Sensitivität: 0.94
F1-Score: 0.94

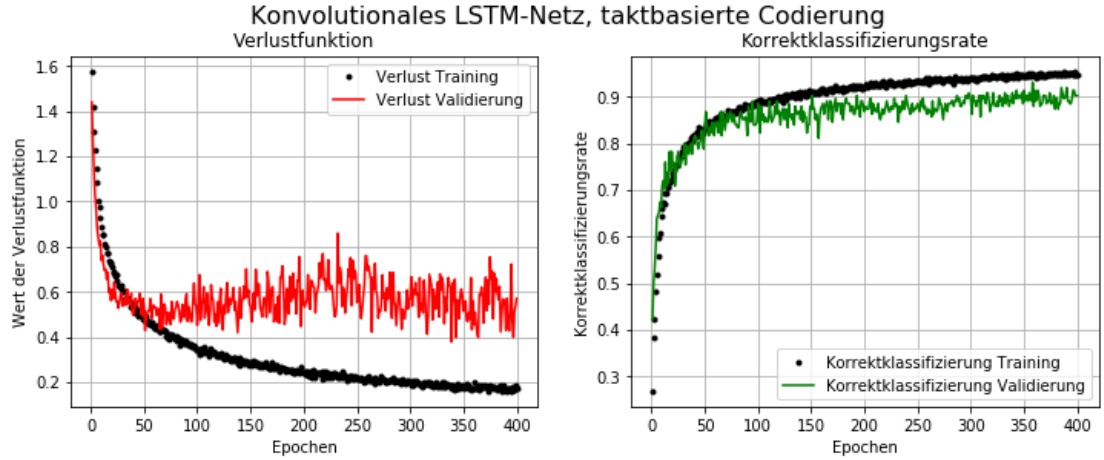
| **LSTM, Zeit, Set 1** | MuR | Ba | Kl | Ro | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 34 | 0 | 1 | 0 | 0 | 1.00 | 0.97 | 0.99 |
| Barock | 0 | 34 | 0 | 1 | 0 | 0.85 | 0.97 | 0.91 |
| Klassik | 0 | 5 | 30 | 0 | 0 | 0.97 | 0.86 | 0.91 |
| Romantik | 0 | 1 | 0 | 31 | 3 | 0.97 | 0.89 | 0.93 |
| Ragtime | 0 | 0 | 0 | 0 | 35 | 0.92 | 1.00 | 0.96 |




## Konvolutionales LSTM-Netz, taktabhängige Version

Korrektklassifizierungsrate: 93.1%
Relevanz: 0.94
Sensitivität: 0.93
F1-Score: 0.93

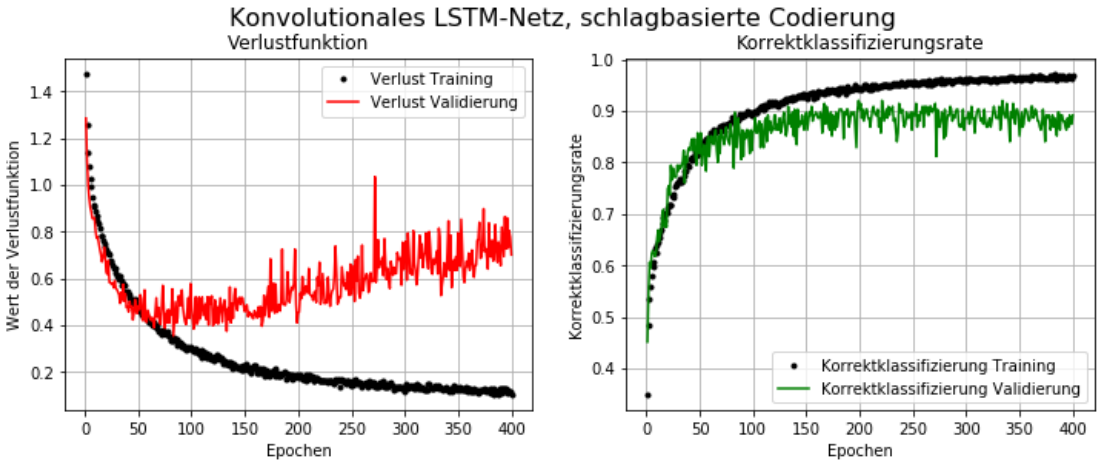
| **CLSTM, Takt** | MuR | Ba | Kl | Ro | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 34 | 1 | 0 | 0 | 0 | 0.92 | 0.97 | 0.94 |
| Barock | 2 | 33 | 0 | 0 | 0 | 0.79 | 0.94 | 0.86 |
| Klassik | 0 | 4 | 31 | 0 | 0 | 1.00 | 0.89 | 0.94 |
| Romantik | 1 | 4 | 0 | 30 | 0 | 1.00 | 0.86 | 0.92 |
| Ragtime | 0 | 0 | 0 | 0 | 35 | 1.00 | 1.00 | 1.00 |



### Konvolutionales LSTM-Netz, schlagabhängige Version

Korrektklassifizierungsrate: 92.0%
Relevanz: 0.92
Sensitivität: 0.92
F1-Score: 0.92

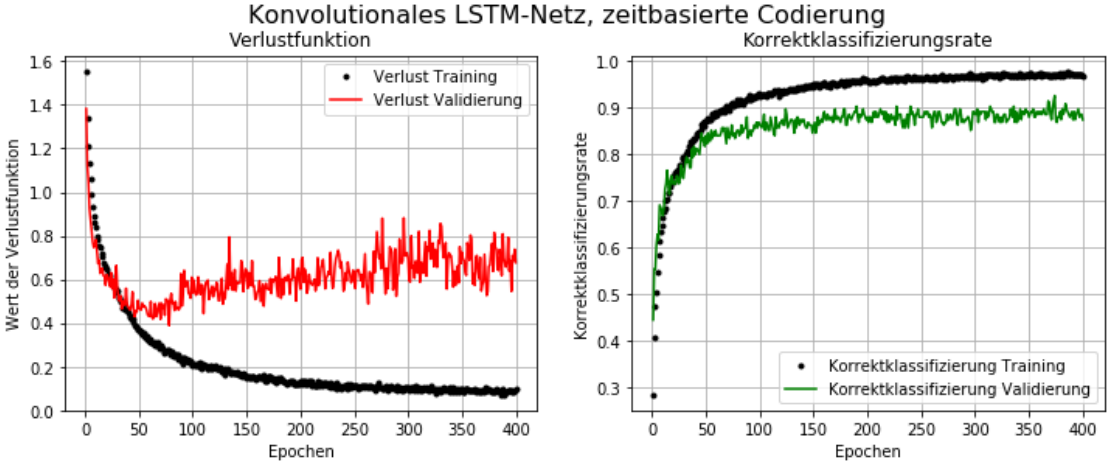
| **CLSTM, Schlag** | MuR | Ba | Kl | Ro | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 34 | 1 | 0 | 0 | 0 | 0.92 | 0.97 | 0.94 |
| Barock | 1 | 30 | 3 | 1 | 0 | 0.91 | 0.86 | 0.88 |
| Klassik | 0 | 2 | 32 | 1 | 0 | 0.86 | 0.91 | 0.89 |
| Romantik | 2 | 0 | 2 | 30 | 1 | 0.94 | 0.86 | 0.90 |
| Ragtime | 0 | 0 | 0 | 0 | 35 | 0.97 | 1.00 | 0.99 |




### Konvolutionales LSTM-Netz, zeitabhängige Version

Korrektklassifizierungsrate: 92.6%
Relevanz: 0.93
Sensitivität: 0.93
F1-Score: 0.93

| **CLSTM, Zeit** | MuR | Ba | Kl | Ro | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 32 | 2 | 0 | 1 | 0 | 0.97 | 0.91 | 0.94 |
| Barock | 1 | 31 | 3 | 0 | 0 | 0.84 | 0.89 | 0.86 |
| Klassik | 0 | 3 | 32 | 0 | 0 | 0.91 | 0.91 | 0.91 |
| Romantik | 0 | 1 | 0 | 32 | 2 | 0.97 | 0.91 | 0.94 |
| Ragtime | 0 | 0 | 0 | 0 | 35 | 0.95 | 1.00 | 0.97 |

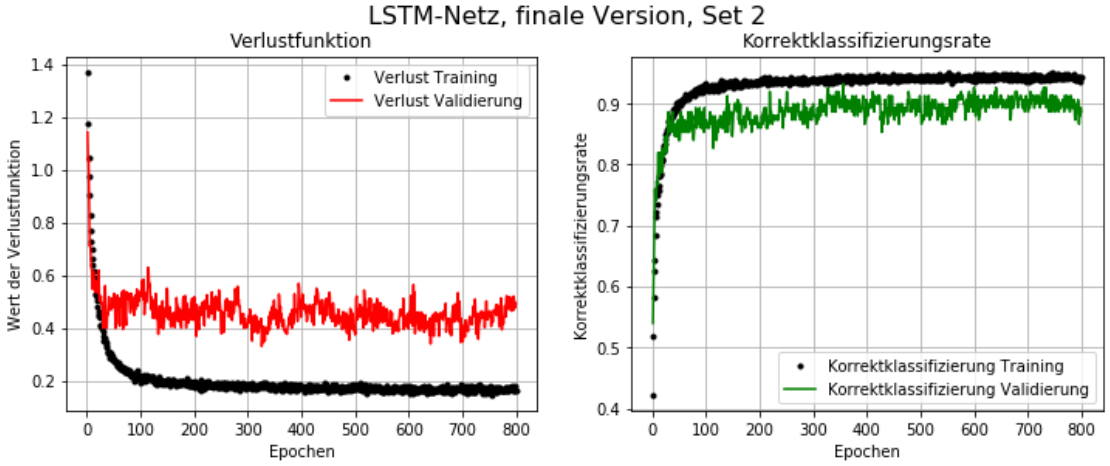



## A.2. Phase 2: Feinabstimmung des LSTM-Netzes, Datenset 2

**Validierungsmenge:**

Korrektklassifizierungsrate: 93.3%
Relevanz: 0.94
Sensitivität: 0.93
F1-Score: 0.93

| **LSTM, Takt, Set 2** | MuR | Ba | Kl | Ro | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 29 | 0 | 0 | 1 | 0 | 0.85 | 0.97 | 0.91 |
| Barock | 1 | 28 | 1 | 0 | 0 | 0.97 | 0.93 | 0.95 |
| Klassik | 2 | 0 | 28 | 0 | 0 | 0.97 | 0.93 | 0.95 |
| Romantik | 2 | 1 | 0 | 26 | 1 | 0.93 | 0.87 | 0.90 |
| Ragtime | 0 | 0 | 0 | 1 | 29 | 0.97 | 0.97 | 0.97 |



#### Testmenge:

Korrektklassifizierungsrate: 92.7%
Relevanz: 0.93
Sensitivität: 0.93
F1-Score: 0.93

| LSTM, Takt, Set 2 | MuR | Ba | Kl | Ro | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 27 | 2 | 0 | 1 | 0 | 0.96 | 0.90 | 0.93 |
| Barock | 1 | 25 | 1 | 2 | 1 | 0.86 | 0.83 | 0.85 |
| Klassik | 0 | 0 | 30 | 0 | 0 | 0.97 | 1.0 | 0.98 |
| Romantik | 0 | 2 | 0 | 28 | 0 | 0.88 | 0.93 | 0.90 |
| Ragtime | 0 | 0 | 0 | 1 | 29 | 0.97 | 0.97 | 0.97 |

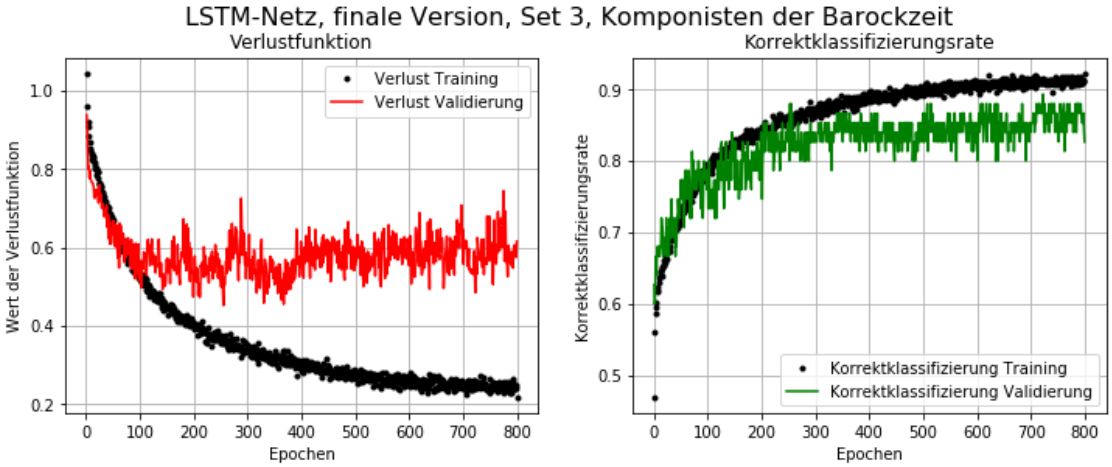
## A.3. Phase 3: Überprüfung des Netzes an anderen Datenmengen

### Komponisten der Barockzeit, Datenset 3

#### Validierungsmenge:

Korrektklassifizierungsrate: 89.3%
Relevanz: 0.89
Sensitivität: 0.89
F1-Score: 0.89

| LSTM, Takt, Set 3 | Bach | Händel | Scarlatti | prec. | recall | f1-score |
|---|---|---|---|---|---|---|
| Bach | 23 | 1 | 1 | 0.92 | 0.92 | 0.92 |
| Händel | 1 | 22 | 2 | 0.88 | 0.88 | 0.88 |
| Scarlatti | 1 | 2 | 22 | 0.88 | 0.88 | 0.88 |



### Testmenge:

Korrektklassifizierungsrate: 77.3%
Relevanz: 0.78
Sensitivität: 0.77
F1-Score: 0.77

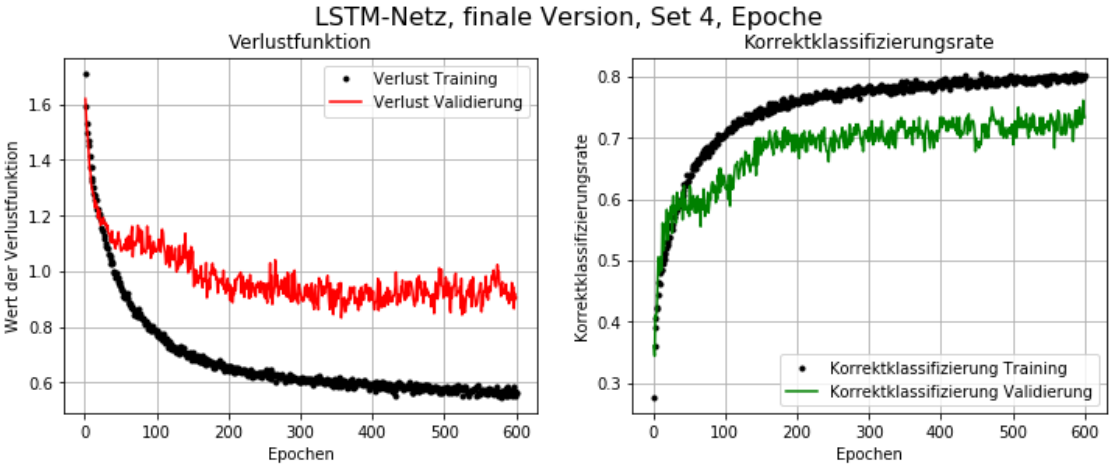
| **LSTM, Takt, Set 3** | Bach | Händel | Scarlatti | prec. | recall | f1-score |
|---|---|---|---|---|---|---|
| Bach | 20 | 3 | 2 | 0.77 | 0.80 | 0.78 |
| Händel | 3 | 20 | 2 | 0.74 | 0.80 | 0.77 |
| Scarlatti | 3 | 4 | 18 | 0.82 | 0.72 | 0.77 |

## Klassifikation diverser Komponisten nach Epoche, Datenset 4

### Validierungsmenge:

Korrektklassifizierungsrate: 76.1%
Relevanz: 0.76
Sensitivität: 0.76
F1-Score: 0.76

| **LSTM, Set 4** | MuR | Ba | Kl | Ro | Rag | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|---|
| MuR | 29 | 1 | 0 | 0 | 0 | 0 | 0.85 | 0.97 | 0.91 |
| Barock | 1 | 22 | 5 | 1 | 0 | 1 | 0.73 | 0.73 | 0.73 |
| Klassik | 0 | 4 | 23 | 3 | 0 | 0 | 0.68 | 0.77 | 0.72 |
| Romantik | 2 | 1 | 2 | 16 | 3 | 6 | 0.67 | 0.53 | 0.59 |
| Ragtime | 0 | 0 | 0 | 3 | 27 | 0 | 0.87 | 0.90 | 0.89 |
| 20. Jh | 2 | 2 | 4 | 1 | 1 | 20 | 0.74 | 0.67 | 0.70 |



### Testmenge:

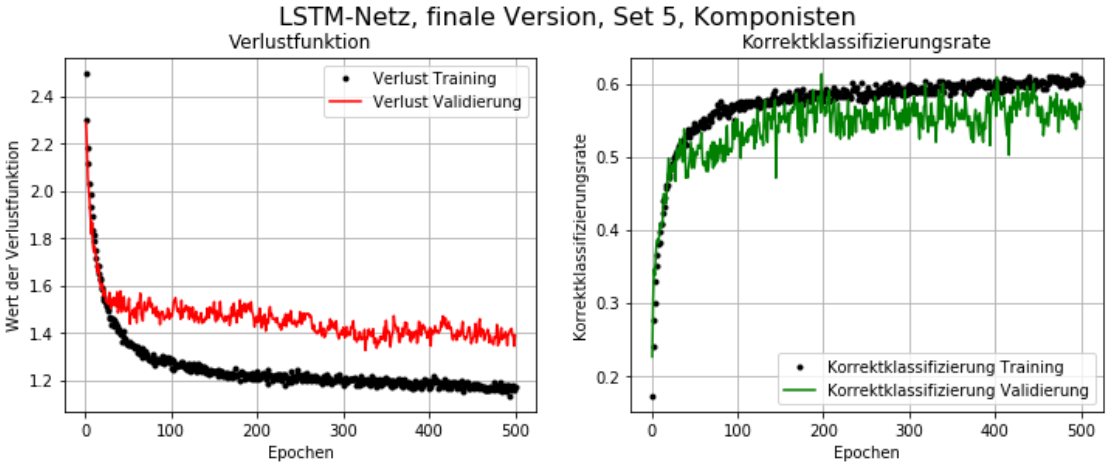
Korrektklassifizierungsrate: 79.4%
Relevanz: 0.79
Sensitivität: 0.79
F1-Score: 0.79

| **LSTM, Set 4** | MuR | Ba | Kl | Ro | Rag | 20.Jh | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|---|
| MuR | 29 | 0 | 0 | 0 | 0 | 1 | 0.85 | 0.97 | 0.91 |
| Barock | 2 | 25 | 1 | 1 | 0 | 1 | 0.81 | 0.83 | 0.82 |
| Klassik | 1 | 2 | 23 | 2 | 0 | 2 | 0.82 | 0.77 | 0.79 |
| Romantik | 0 | 2 | 2 | 18 | 3 | 5 | 0.72 | 0.60 | 0.65 |
| Ragtime | 0 | 0 | 0 | 1 | 29 | 0 | 0.85 | 0.97 | 0.91 |
| 20. Jh | 2 | 2 | 2 | 3 | 2 | 19 | 0.68 | 0.63 | 0.66 |

## Klassifikation nach Komponist, 15 Kategorien, Datenset 5

### Validierungsmenge:

Korrektklassifizierungsrate: 61.3%
Relevanz: 0.61
Sensitivität: 0.61
F1-Score: 0.60



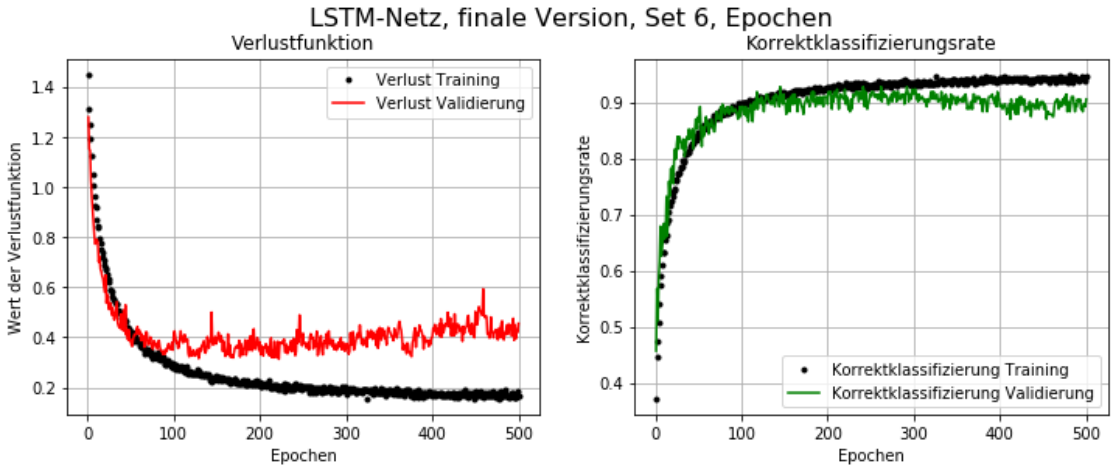
| **Komp** | MR | Ba | Hä | Sc | Ha | Cl | Mo | Be | Sb | Sm | Ch | Br | Ra | Scr | Bar |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MR | **13** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Ba | 0 | **16** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Hä | 0 | 1 | **11** | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Sc | 0 | 0 | 0 | **10** | 3 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| Ha | 0 | 0 | 2 | 1 | **7** | 1 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Cl | 0 | 1 | 0 | 5 | 1 | **3** | 1 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| Mo | 0 | 0 | 2 | 0 | 3 | 1 | **8** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Be | 0 | 0 | 1 | 0 | 1 | 0 | 2 | **7** | 1 | 2 | 0 | 0 | 0 | 0 | 1 |
| Sb | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | **8** | 1 | 1 | 1 | 1 | 1 | 0 |
| Sm | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | **10** | 0 | 0 | 0 | 1 | 0 |
| Ch | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 1 | **10** | 1 | 0 | 0 | 0 |
| Br | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | **10** | 1 | 0 | 0 |
| Rag | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | **14** | 0 | 0 |
| Scr | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 1 | 3 | 0 | **6** | 1 |
| Bar | 3 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 1 | 0 | 1 | 1 | **5** |

| | prec. | recall | f1-score |
|---|---|---|---|
| MA/Ren. | 0.76 | 0.87 | 0.81 |
| Bach | 0.89 | 1.0 | 0.94 |
| Händel | 0.61 | 0.73 | 0.67 |
| Scarlatti | 0.56 | 0.67 | 0.61 |
| Haydn | 0.44 | 0.50 | 0.47 |
| Clementi | 0.60 | 0.20 | 0.30 |
| Mozart | 0.50 | 0.53 | 0.52 |
| Beethoven | 0.54 | 0.47 | 0.50 |
| Schubert | 0.47 | 0.53 | 0.50 |
| Schumann | 0.53 | 0.67 | 0.59 |
| Chopin | 0.59 | 0.67 | 0.62 |
| Brahms | 0.67 | 0.67 | 0.67 |
| Ragtime | 0.82 | 0.93 | 0.87 |
| Scriabin | 0.67 | 0.40 | 0.50 |
| Bartók | 0.50 | 0.33 | 0.40 |

### Finales LSTM-Netz, Klassifikation nach fünf Epochen, Datenset 6

Korrektklassifizierungsrate: 92.9%
Relevanz: 0.93
Sensitivität: 0.93
F1-Score: 0.93

| **LSTM, Takt, Set 6** | MuR | Ba | Kl | Ro | Rag | prec. | recall | f1-score |
|---|---|---|---|---|---|---|---|---|
| Mittelalter/Renaissance | 45 | 0 | 0 | 0 | 0 | 0.98 | 1.00 | 0.99 |
| Barock | 0 | 40 | 5 | 0 | 0 | 0.91 | 0.89 | 0.90 |
| Klassik | 1 | 1 | 41 | 2 | 0 | 0.89 | 0.91 | 0.90 |
| Romantik | 0 | 3 | 0 | 38 | 4 | 0.95 | 0.84 | 0.89 |
| Ragtime | 0 | 0 | 0 | 0 | 45 | 0.92 | 1.00 | 0.96 |



# B. Inhalt der beigefügten CD-ROM

- vorliegende Bachelor-Arbeit im pdf-Format:
  `bachelorarbeit-schaefer-silke.pdf`
- im Ordner `implementierung` 14 Jupyter-Notebooks:
  - aus Phase 1 (Netzauswahl):
    1. `music-classification-01-nn-bar.ipynb`: vollverbundenes Feed-Forward-Netzwerk mit taktabhängiger Codierung
    2. `music-classification-02-nn-beat.ipynb`: vollverbundenes Feed-Forward-Netzwerk mit schlagabhängiger Codierung
    3. `music-classification-03-nn-time.ipynb`: vollverbundenes Feed-Forward-Netzwerk mit zeitabhängiger Codierung
    4. `music-classification-04-lstm-bar.ipynb`: *LSTM*-Netz mit taktabhängiger Codierung
    5. `music-classification-05-lstm-beat.ipynb`: *LSTM*-Netz mit schlagabhängiger Codierung
    6. `music-classification-06-lstm-time.ipynb`: *LSTM*-Netz mit zeitabhängiger Codierung
    7. `music-classification-07-clstm-bar.ipynb`: konvolutionales *LSTM*-Netz mit taktabhängiger Codierung
    8. `music-classification-08-clstm-beat.ipynb`: konvolutionales *LSTM*-Netz mit schlagabhängier Codierung
    9. `music-classification-09-lstm-time.ipynb`: konvolutionales *LSTM*-Netz mit zeitabhängiger Codierung
  - aus Phase 2 (Feinabstimmung):
    10. `music-classification-10-lstm-final-set2.ipynb`: *LSTM*-Netz, das auf der zweiten Datenmenge trainiert wurde (Kategorien: Mittelalter/Renaissance, Bach, Haydn, Schumann, Ragtime)
  - aus Phase 3 (Leistungen des Netzes auf anderen Datenmengen):
    11. `music-classification-11-lstm-final-set3.ipynb`: *LSTM*-Netz, das auf der dritten Datenmenge trainiert wurde (Kategorien: Bach, Händel, Scarlatti)

12. `music-classification-12-lstm-final-set4-era.ipynb`: *LSTM*-Netz, das auf der vierten Datenmenge trainiert wurde und nach Epoche klassifiziert (Kategorien: Mittelalter/Renaissance, Barock, Wiener Klassik, Romantik, Ragtime, 20. Jh.)
13. `music-classification-13-lstm-final-set5.ipynb`: *LSTM*-Netz, das auf der fünften Datenmenge trainiert wurde und nach Komponist klassifiziert (15 Kategorien)
14. `music-classification-14-lstm-final-set6.ipynb`: *LSTM*-Netz, das auf der sechsten Datenmenge trainiert wurde und nach Epoche klassifiziert (5 Kategorien)

- ebenfalls im Ordner `implementierung` folgende Ordner mit *MIDI*-Dateien:
  - `data1-training` und `data1-validation`:
    Set 1, enthält Stücke aus fünf verschiedenen Epochen.
  - `data2-training`, `data2-validation` und `data2-test`:
    Set 2, enthält ebenfalls Stücke aus fünf Epochen.
  - `data3-training`, `data3-validation` und `data3-test`:
    Set 3, enthält Stücke von Bach, Händel und Scarlatti.
  - `data4-training`, `data4-validation` und `data4-test`:
    Set 4, enthält Stücke aus sechs verschiedenen Epochen, die jeweils durch mehrere Komponisten repräsentiert werden.
  - `data5-training` und `data5-validation`:
    Set 5, enthält Stücke aus 15 Kategorien: Mittelalter/Renaissance, Bach, Händel, Scarlatti, Haydn, Clementi, Mozart, Beethoven, Brahms, Chopin, Schubert, Schumann, Ragtime, Bartók, Scriabin.
  - `data6-training` und `data6-validation`:
    Set 6, enthält Stücke aus fünf verschiedenen Epochen, die jeweils durch mehrere Komponisten repräsentiert werden.
- im Ordner `auswertung` zu jedem der 14 Jupyter-Notebooks mit der Bezeichnung `music-classification-<xy>`:
  - eine Datei `results-<xy>-best_weights.hdf5`, die die Gewichte des erfolgreichsten Modells enthält. Diese können in ein passendes Modell geladen werden, um den Klassifizierer an neuen Daten zu testen. Eine solche Datei wird auch in jedem Programmlauf in das aktuelle Arbeitsverzeichnis geschrieben bzw. die alte Datei überschrieben.
  - eine Datei `results-<xy>-plots.png`, die die Plots der Verlustfunktion und der Korrektklassifizierungsrate enthält.

# Abbildungsverzeichnis

# Literatur


[1] MIDI Manufacturers Association u. a. „The complete MIDI 1.0 detailed specification“. In: *Los Angeles, CA, The MIDI Manufacturers Association* (1996).

[2] Gabriele Kern-Isberner Christoph Beierle und Gabriele Kern-Isberner. *Methoden wissensbasierter Systeme*. Springer, 2003.

[3] Yoshua Bengio, Patrice Simard und Paolo Frasconi. „Learning long-term dependencies with gradient descent is difficult“. In: *IEEE transactions on neural networks* 5.2 (1994), S. 157–166.

[4] François Chollet. *Deep learning with python*. Manning Publications Co., 2017.

[5] Douglas Eck und Juergen Schmidhuber. „Finding temporal structure in music: Blues improvisation with LSTM recurrent networks“. In: *Neural Networks for Signal Processing, 2002. Proceedings of the 2002 12th IEEE Workshop on*. IEEE. 2002, S. 747–756.

[6] Michael Freitag u. a. „auDeep: Unsupervised learning of representations from audio with deep recurrent neural networks“. In: *The Journal of Machine Learning Research* 18.1 (2017), S. 6340–6344.

[7] Yarin Gal und Zoubin Ghahramani. „A theoretically grounded application of dropout in recurrent neural networks“. In: *Advances in neural information processing systems*. 2016, S. 1019–1027.

[8] Aurélien Géron. *Praxiseinstieg Machine Learning mit Scikit-Learn & TensorFlow. Konzepte, Tools und Techniken für intelligente Systeme*. O'Reilly, 2018. ISBN: 978-3-96009-061-8.

[9] Felix A Gers, Jürgen Schmidhuber und Fred Cummins. „Learning to forget: Continual prediction with LSTM“. In: *9th International Conference on Artificial Neural Networks: ICANN '99* (1999), S. 850–855.

[10] Xavier Glorot, Antoine Bordes und Yoshua Bengio. „Deep Sparse Rectifier Neural Networks“. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Hrsg. von Geoffrey Gordon, David Dunson und Miroslav Dudík. Bd. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Nov. 2011, S. 315–323.

[11] Donald Olding Hebb. *The Organizations of Behavior: a Neuropsychological Theory*. Lawrence Erlbaum, 1963.

[12] Sepp Hochreiter und Jürgen Schmidhuber. „Long short-term memory“. In: *Neural computation* 9.8 (1997), S. 1735–1780.

[13] Rafal Jozefowicz, Wojciech Zaremba und Ilya Sutskever. „An empirical exploration of recurrent network architectures“. In: *International Conference on Machine Learning*. 2015, S. 2342–2350.

[14] Feynman Liang u. a. „Automatic stylistic composition of bach chorales with deep LSTM“. In: *Proceedings of the 18th International Society for Music Information Retrieval Conference (ISMIR-17), Suzhou, China*. 2017.

[15] Warren S McCulloch und Walter Pitts. „A logical calculus of the ideas immanent in nervous activity“. In: *The bulletin of mathematical biophysics* 5.4 (1943), S. 115–133.

[16] Tom M Mitchell u. a. „Machine learning. 1997“. In: *Burr Ridge, IL: McGraw Hill* 45.37 (1997), S. 870–877.

[17] Andrew Ng. *Machine Learning*. 2011. URL: `https://www.coursera.org/learn/machine-learning` (besucht am 02. 11. 2018).

[18] Christopher Olah. *Understanding lstm networks*. 2015. URL: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/` (besucht am 03. 11. 2018).

[19] David E Rumelhart, Geoffrey E Hinton und Ronald J Williams. „Learning representations by back-propagating errors“. In: *nature* 323.6088 (1986), S. 533.

[20] Stuart Russell und Peter Norvig. *Künstliche Intelligenz*. Pearson, 2012.

[21] Arthur L Samuel. „Some studies in machine learning using the game of checkers“. In: *IBM Journal of research and development* 3.3 (1959), S. 210–229.

[22] Jürgen Schmidhuber. „Deep learning in neural networks: An overview“. In: *Neural networks* 61 (2015), S. 85–117.

[23] John R. Searle. „Minds, brains, and programs“. In: *Behavioral and Brain Sciences* 3.3 (1980), S. 417–424. DOI: `10.1017/S0140525X00005756`.

[24] Hagen Soltau, Hank Liao und Hasim Sak. „Neural speech recognizer: Acoustic-to-word LSTM model for large vocabulary speech recognition“. In: *arXiv preprint arXiv:1610.09975* (2016).

[25] Nitish Srivastava u. a. „Dropout: a simple way to prevent neural networks from overfitting“. In: *The Journal of Machine Learning Research* 15.1 (2014), S. 1929–1958.

[26] Alan M Turing. „Computing machinery and intelligence“. In: *Parsing the Turing Test*. Springer, 2009, S. 23–65.

[27] George Tzanetakis und Perry Cook. „Musical genre classification of audio signals“. In: *IEEE Transactions on speech and audio processing* 10.5 (2002), S. 293–302.

[28] Paul Werbos. „Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences“. In: *Ph. D. dissertation, Harvard University* (1974).

[29] SHI Xingjian u. a. „Convolutional LSTM network: A machine learning approach for precipitation nowcasting“. In: *Advances in neural information processing systems*. 2015, S. 802–810.

[30] Rob Young. *Arbeiten mit MIDI-Files*. GC Carstensen Verlag, München, 2005. ISBN: 3-910098-34-7.

[31] Rob Young. *The MIDI Files*. Prentice Hall Europe, 1996. ISBN: 0-13-262403-6.

[32] Wieland Ziegenrücker. *Allgemeine Musiklehre*. Breitkopf & Härtel, 1997. ISBN: 978-3-7651-0309-4.

## Erklärung

| | |
|---|---|
| Name: | Silke Schäfer |
| Matrikel-Nr: | 8718040 |
| Fach: | Informatik |
| Modul: | Bachelorarbeit |

Ich erkläre, dass ich die vorliegende Abschlussarbeit mit dem Thema

**Klassifikation klassischer Musik mit Hilfe von neuronalen Netzen**

selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich, inhaltlich oder sinngemäß entnommenen Stellen als solche den wissenschaftlichen Anforderungen entsprechend kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate überprüft und ausschließlich für Prüfungszwecke gespeichert wird.

**Datum:** _____ **Unterschrift:** _____