Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Marina Knabbe

Erzeugung von Musiksequenzen mit LSTM-Netzwerken

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Marina Knabbe

Erzeugung von Musiksequenzen mit LSTM-Netzwerken

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Andreas Meisel
Zweitgutachter: Prof. Dr. Wolfgang Fohl

Eingereicht am: 24. Februar 2017

Marina Knabbe

Thema der Arbeit

Erzeugung von Musiksequenzen mit LSTM-Netzwerken

Stichworte

Musiksequenzen, LSTM, Neuronale Netze, Maschinenlernen, DeepLearning4J, Java

Kurzzusammenfassung

Das Ziel dieser Bachelorarbeit war es, ein LSTM-Netzwerk zur Erzeugung von Musiksequenzen anhand von MIDI-Dateien in der Java Bibliothek DeepLearning4J auf Implementierbarkeit zu prüfen. Hierfür wurden zwei Ansätze der Datenverwaltung verfolgt und mehrere Experimente in der Netzwerkparametrierung durchgeführt, welche die Erwartungen nicht erfüllen konnten, aber die Schwierigkeiten der Benutzung von DeepLearning4J für diese Aufgabe hervorheben.

Marina Knabbe

Title of the paper

Generation of music sequences with LSTM networks

Keywords

music sequences, LSTM, neural networks, machine learning, DeepLearning4J, Java

Abstract

The goal of this bachelor thesis was to test the implementability of an LSTM network for the generation of music sequences using MIDI files in the Java library DeepLearning4J. For this purpose, two approaches to data management were pursued and several experiments with the network parameterization were carried out, which could not meet the expectations but emphasize the difficulties of using DeepLearning4J for this task.

Inhaltsverzeichnis

Glossar **ix**

1. Einleitung **1**

2. Künstliche neuronale Netze **2**
2.1. Feedforward Netzwerke 2
2.1.1. Training durch Backpropagation 4
2.2. Recurrent Netzwerke 4
2.2.1. Training durch Backpropagation Through Time 5
2.2.2. Problem der verschwindenden und explodierenden Gradienten 6
2.2.3. Problem der Langzeit-Abhängigkeiten 6
2.3. Long Short-Term Memory Netze 7
2.3.1. Aufbau einer Speicherzelle 8
2.3.2. LSTM Varianten 10
2.4. Aktueller Forschungsstand 11
2.4.1. Allgemeine Forschung 11
2.4.2. Musikerzeugung 12

3. DeepLearning4J **13**
3.1. Netzwerke erstellen 13
3.1.1. Ein Netzwerk erstellen (3-Schritt-Methode) 14
3.1.2. Ein Netzwerk erstellen (Kurzversion) 16
3.1.3. Layertypen 17
3.2. Netzwerke trainieren 18
3.2.1. Ein Netzwerk trainieren 18
3.2.2. Daten erstellen 18

4. Musiksequenzen mit Hilfe von DL4J erzeugen **20**
4.1. Die Eingabe 20
4.1.1. Was sind MIDI-Dateien? 20
4.1.2. MIDI-Dateien lesen 21
4.1.3. DataSets erstellen 24
4.2. Das Netzwerk 27
4.3. Die Ausgabe 28
4.3.1. Auswerten der Netzwerkausgabe 28
4.3.2. MIDI Files schreiben 30
4.3.3. Ergebnisse 30

5. Fazit **32**

A. DL4J-Projekt in IntelliJ aufsetzen **33**

Tabellenverzeichnis

3.1. Übersicht einiger Netzwerkparameter 14

4.1. MIDI Nachrichtenformat 21
4.2. Ereignisschlüssel 23

Abbildungsverzeichnis

2.1. Feedforward Neuron . . . 3
2.2. Aufbau eines Feedforward Netzes . . . 3
2.3. RNN Neuron . . . 4
2.4. BPTT . . . 5
2.5. Vergleich RNN und LSTM . . . 7
2.6. LSTM Zellzustand . . . 8
2.7. LSTM: Forget Gate . . . 8
2.8. LSTM: Eingangsgate . . . 9
2.9. LSTM: Zellzustands Update . . . 9
2.10. LSTM: Ausgabe . . . 9
2.11. LSTM Variante: Gucklöcher . . . 10
2.12. LSTM Variante: Zusammengeführte Gates . . . 10

4.1. MIDI Nachricht . . . 21
4.2. Beispielausgabe 1 . . . 30
4.3. Beispielausgabe 2 . . . 31

A.1. „New Project“ Fenster . . . 33

Quellcodeverzeichnis

3.1. NeuralNetConfiguration.Builder Beispiel 14
3.2. Erstellen des ListBuilders 15
3.3. Erstellen der Netzwerk-Layer 15
3.4. Fertigstellen des ListBuilder 16
3.5. Ein Netz erzeugen 16
3.6. Netzwerk erstellen (Kurzversionbeispiel) 17
3.7. Ein Netz trainieren 18
3.8. Daten erstellen 18

4.1. MidiEvents Konstruktor 22
4.2. Ereignisobjekt: Listen von gültigen Daten 22
4.3. Ereignisschlüssel: Liste von gültigen Daten 23
4.4. Ereignisobjekt: Trainingsdaten 25
4.5. Ereignisschlüssel: Trainingsdaten 26
4.6. Ereignisobjekt: Größe der Ein- und Ausgangslayer 27
4.7. Ereignisschlüssel: Größe der Ein- und Ausgangslayer 27
4.8. Ereignisobjekt: Netzausgabe 28
4.9. Ereignisschlüssel: Netzausgabe 29

A.1. Backend Dependency CPU 34
A.2. DL4J Dependency 34
A.3. Versionsvariablen 34

Glossar

Aktivierungsfunktion
Zusammenhang zwischen Netzeingabe und Aktivitätslevel eines Neurons

Backpropagation
Fehlerrückführung, Verfahren zum Trainieren von KNNs

Backpropagation Through Time (BPTT)
zeitabhängige Fehlerrückführung, Verfahren zum Trainieren von KNNs

DataSet
Datenformat zum Trainieren von KNNs in DL4J

Deep Learning Netz
ein Netz mit mehr als einem Hidden Layer

DeepLearning4J (DL4J)
Java Bibliothek für KNNs

Feedforward Netzwerk
Klassifizierungsart von künstlichen neuronalen Netzen, bezeichnet vorwärtsgerichtete Netze

Git
Software zur Versionsverwaltung

Gradient
Veränderung aller Gewichte in Bezug auf Veränderung im Fehler

Gradient Descent
Algorithmus zum Finden eines lokalen Minimums

INDArrays
Datenformat, welches von DataSet benutzt wird

IntelliJ IDEA
Entwicklungsumgebung für Java

Künstliches Neuronales Netz (KNN)
Teil der künstlichen Intelligenz, findet ihre Anwendung im Bereich des Maschinen Lernens

Label
: Bezeichnung der Ausgangsdaten in DL4J

Layer
: Bestandteil eines KNNs, bezeichnet eine Schicht die aus Neuronen besteht

Long Short-Term Memory (LSTM) Netze
: Klassifizierungsart von RNNs

Maven
: Software für Projektmanagement

MIDI
: (steht für: Musical Instrument Digital Interface) Industriestandard zum Austausch musikalischer Steuerinformationen

Neuron
: Bestandteil eines KNNs, Neuronen bilden die Netzwerk Layer

Recurrent Neuronal Networks (RNN)
: Klassifizierungsart von künstlichen neuronalen Netzen, bezeichnet rückgekoppelte Netze

Restricted Boltzmann Maschinen (RBM)
: eine Art von KNN, nützlich zur Dimensionsreduktion

XOR
: Exklusives Oder, Logikelement

1. Einleitung

Diese Arbeit befasst sich mit der Erzeugung von Musiksequenzen mit LSTM-Netzwerken und ihrer Implementierung in der Java Bibliothek DeepLearning4J (DL4J). Anhand von MIDI-Beispieldateien soll das neuronale Netz in die Lage gebracht werden eigene Musiksequenzen zu erstellen.

Hierfür werden im zweiten Kapitel die Grundlagen der künstlichen neuronalen Netze erläutert. Untergliedert ist dieser Abschnitt in die Teilthemen Feedforward Netzwerke, Recurrent Netzwerke und Long Short-Term Memory Netze. Anschließend wird eine kleine Übersicht über den aktuellen Forschungsstand im Bereich der LSTM-Netzwerke gegeben.

Kapitel 3 befasst sich mit der Java Bibliothek DeepLearning4J und gibt einen Einblick in die Benutzung, indem es Beispiele zur Netzwerkerstellung und Trainierung aufzeigt. Wie ein neues DeepLearning4J Projekt in der Entwicklungsumgebung IntelliJ aufgesetzt werden kann, ist im Anhang A am Ende dieser Arbeit zu finden.

Im vierten Kapitel geht es um die eigentliche Implementierung des LSTM-Netzwerkes in DL4J. Dieser Bereich ist in die Abschnitte Eingabe, Netzwerk und Ausgabe aufgeteilt. Die Eingabe umfasst die Gebiete der MIDI-Dateien als Trainingsdaten und die Erstellung von DataSets. Die Ausgabe besteht aus der Auswertung der Netzwerkausgabe, dem Erstellen von MIDI-Dateien und die mit der Netzwerkimplementierung erreichten Ergebnisse.

Als Abschluss folgt das gezogene Fazit.

2. Künstliche neuronale Netze

Künstliche Intelligenz lässt sich mit einem Künstlichen Neuronalen Netz (KNN) realisieren. KNNs basieren auf dem Vorbild des biologischen neuronalen Netz des Gehirns. [Breitner 2014] schreibt dazu:

> „Die biologischen Vorgänge des menschlichen Denkens und Lernens (Aktivierung von Neuronen, chemische Veränderung von Synapsen usw.) werden, so gut wie möglich, mathematisch beschrieben und in Software oder Hardware modelliert."

KNNs bestehen also aus einem Satz von Algorithmen, welche Daten interpretieren. Diese Eingangsdaten sind numerisch und müssen meist durch Umwandlung der originalen Daten (z.B. Bilder, Text oder Musik) geschaffen werden. Sie sind dazu entwickelt anhand von Mustererkennung Daten eigenständig zu Gruppieren, vorgegebenen Klassen zu zuordnen oder den weiteren Verlauf vorherzusagen.

Das Training eines KNNs lässt sich in zwei Kategorien unterteilen: das überwachte Lernen und das unüberwachte Lernen. Beim überwachten lernen werden dem Netz Eingangs- und Ausgangsdaten gegeben anhand dessen das Netz den Zusammenhang erlernt und später in der Lage ist neue Daten entsprechend zu klassifizieren oder die nächste Ausgabe vorherzusagen. Beim unüberwachten Lernen erhält das Netz lediglich Eingangsdaten und lernt diese anhand von Ähnlichkeiten zu gruppieren.

Es gibt verschiedene Arten von Künstlichen Neuronalen Netzen und in den folgenden Abschnitten werden drei von ihnen erklärt.

2.1. Feedforward Netzwerke

Ein Feedforward Netzwerk besteht aus Layern und Neuronen. Die Neuronen sind für die Berechnungen der Ausgabe zuständig, während die Layer den Aufbau des Netzes bestimmen. Abbildung 2.1 zeigt einen möglichen Aufbau eines künstliches Neurons. Dieses Neuron besteht aus 1 bis x_m Eingängen (Inputs) mit Gewichten (Weights), einer Eingangsfunktion (Net input function), einer Aktivierungsfunktion (Activation function) und einem Ausgang (Outputs). Die zu verarbeiteten Daten werden an die Eingänge gelegt, durch die zugehörigen Gewichte

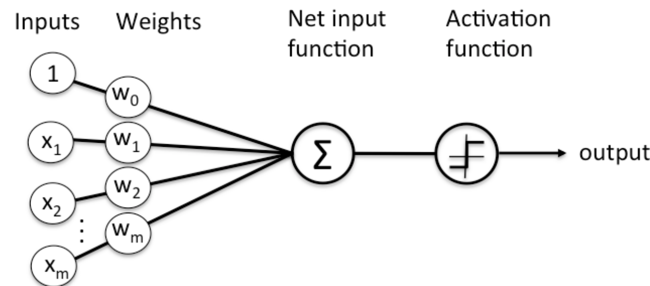

Abb. 2.1.: mögliches Aussehen eines Feedforward Neurons (Quelle: [4])

verstärkt oder abgeschwächt und anschließend durch die Eingangsfunktion aufsummiert. Die entstandene Summe wird dann an die Aktivierungsfunktion übergeben, welche das Ergebnis dieses Neurons festlegt.

Ein Layer besteht aus einer Reihe von Neuronen beliebiger Anzahl. Ein künstliches neuronales Netz setze sich aus einem Input Layer, einem Output Layer und beliebig vielen Hidden Layern zusammen. Hat ein Netz mehr als ein Hidden Layer so wird es auch als Deep Learning Netz bezeichnet. Abbildung 2.2 zeigt ein Feedforward Netzwerk. Bei diesem Netz besitzt das

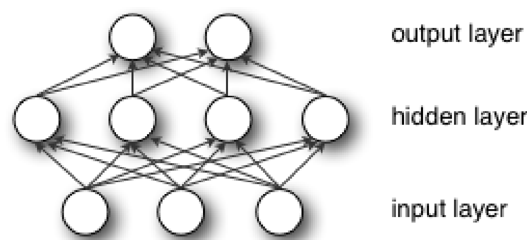

Abb. 2.2.: Aufbau eines Feedforward Netzes (Quelle: [4])

Input Layer drei Neuronen, das Hidden Layer hat vier und das Output Layer hat zwei Neuronen. Die Ergebnisse des Input und Hidden Layers dienen dem nachfolgenden Layer als Eingang. Die Eingänge des Input Layers und der Ausgang des Output Layers sind hier nicht dargestellt, da der Fokus auf der inneren Verknüpfung liegen soll. Jedes Neuron hat hier so viel Ausgänge wie die Anzahl der Neuronen im folgenden Layer und ist somit vollständig verknüpft. Dies muss nicht immer der Fall sein, doch auf diesen Sonderfall soll hier nicht weiter eingegangen werden.

2.1.1. Training durch Backpropagation

Ein neuronales Netz kann anhand von Trainingsdaten eine Funktion erlernen, indem es die Gewichte verändert. Um sinnvolle Ergebnisse zu erhalten müssen die Gewichte solange angepasst werden, bis der Fehler zwischen Netzausgabe und tatsächlichen Ausgabewert am kleinsten ist. Dies wird mit Hilfe der Backpropagation gemacht, indem Rückwärts vom Fehler über die Ausgänge, die Gewichte und die Eingänge der verschiedenen Layer ein Zusammenhang zwischen Fehlergröße und einzelnen Gewichtseinstellungen hergestellt wird. Für die Bestimmung der benötigten Gewichte benutzt man Optimierungsfunktionen. Eine weitverbreitete Optimierungsfunktion heißt Gradient Descent. Sie beschreibt das Verhältnis des Fehlers zu einem einzelnen Gewicht und wie sich der Fehler verändert, wenn das Gewicht angepasst wird.

Das Ziel ist möglichst schnell den Punkt zu erreichen an dem der Fehler am kleinsten ist. Um dies zu erreichen wiederholt das Netz so oft wie nötig die folgenden Schritte: Ergebnis anhand der aktuellen Gewichte bestimmen, Fehler messen, Gewichte aktualisieren.

2.2. Recurrent Netzwerke

Recurrent Neuronal Networks (RNN) betrachten im Gegensatz zu Feedforward Netzwerken nicht nur die aktuellen Eingangsdaten sondern auch die vorhergegangenen. Sie besitzen daher zwei Eingangsgrößen, nämlich die gerade angelegten und die zurückgeleiteten aus dem vorherigen Zeitschritt. Abbildung 2.3 zeigt links eine vereinfachte Darstellung eines

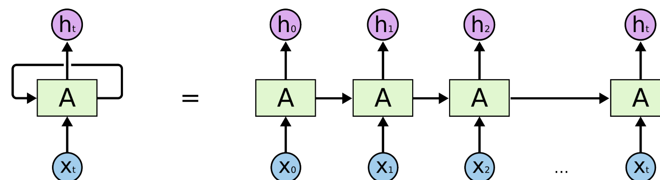

Abb. 2.3.: vereinfachte Darstellung eines RNN Neurons (Quelle: [3])

RNN Neurons mit Rückführung, aber ohne dargestellte Gewichte oder Aktivierungsfunktion. Rechts ist das Ganze als zeitlicher Verlauf dargestellt. Im ersten Zeitschritt wird x_0 an den Eingang gelegt und h_0 als Ergebnis berechnet. Außerdem führt ein Pfeil zum Neuron im zweiten Zeitschritt und dient dort als zweiter Eingang. Das Ergebnis, das ein Neuron liefert ist also immer vom vorherigen abhängig. Man bezeichnet dies auch als Gedächtnis des Netzes. Einem Netz ein Gedächtnis zu geben macht immer dann Sinn, wenn die Eingangsdaten eine Sequenz bilden und nicht komplett unabhängig von einander sind. Im Gegensatz zu den Feedforward

Netzen können Recurrent Netzwerke Sequenzen erfassen und sie zur Erzeugung ihrer Ausgaben nutzen. Dies ist zum Beispiel bei der automatischen Textgenerierung hilfreich, wo ein folgender Buchstabe immer vom vorherigen abhängt und nicht willkürlich gewählt werden kann. Ein RNN ist in der Lage gezielt auf ein q ein u folgen zu lassen um sinnvolle Wörter zu bilden, ein Feedforward Netzwerk kann das nicht.

2.2.1. Training durch Backpropagation Through Time

Da bei Recurrent Netzen das Ergebnis und somit der Fehler nicht nur vom aktuellen Zeitschritt abhängt, muss auch die Backpropagation erweitert werden um sinnvoll arbeiten zu können. Backpropagation Through Time (BPTT) ergänzt die normale Backpropagation um den Faktor Zeit, so dass ein Einfluss auf den Fehler von einem Gewicht aus früheren Schritten ermittelt werden kann. Abbildung 2.4 soll diesen Vorgang verdeutlichen. Sie zeigt ein Recurrent Netz

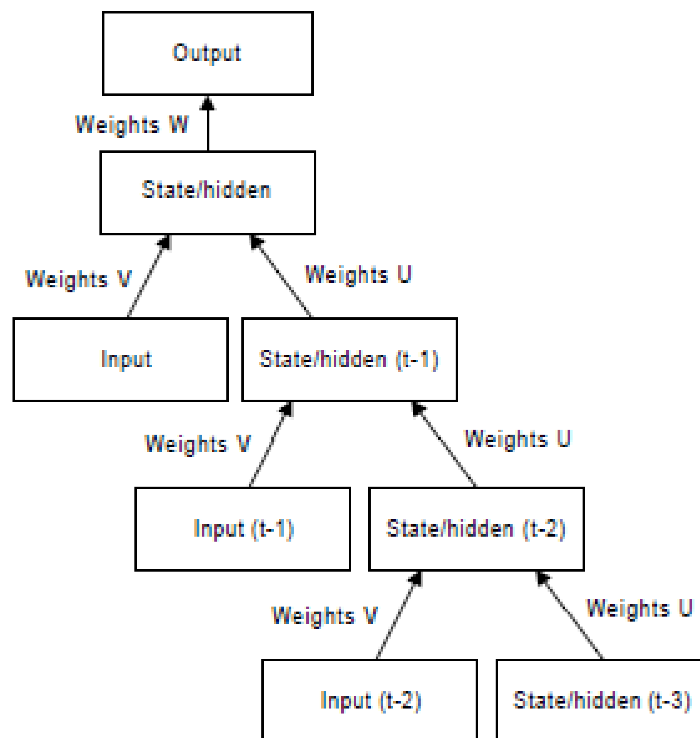

Abb. 2.4.: entrolltes RNN für BPTT (Quelle: [2])

das um drei Zeitschritte entrollt wurde, indem Komponenten dupliziert wurden. Dadurch lösen sich die Rückführungen auf und das Netzwerk verhält sich wie ein Feedforward Netz. Der

Einfluss jedes Gewichts kann nun anteilig berechnet und anschließend summiert werden, so dass ein einzelner Wert je Gewicht für die Anpassung ermittelt wird.

Dieses Verfahren benötigt natürlich mehr Speicher, da alle vorherigen Zustände und Daten für eine bestimme Anzahl an Zeitschritten gespeichert werden müssen.

2.2.2. Problem der verschwindenden und explodierenden Gradienten

Der Gradient stellt die Veränderung aller Gewichte in Bezug auf Veränderung im Fehler dar. Wenn der Gradient unbekannt ist, ist eine Veränderung an den Gewichten zur Verkleinerung des Fehlers nicht möglich und das Netz ist nicht in der Lage zu lernen. Zu unbekannten Gradienten kann es kommen, da Informationen die durch ein Deep Netz fließen vielfach multipliziert werden. Multipliziert man einen Betrag regelmäßig mit einem Wert knapp größer 1 kann das Ergebnis unmessbar groß werde und in diesem Fall spricht man von einem explodierenden Gradienten. Umgekehrt führt eine wiederholte Multiplikation eines Betrages mit einem Wert kleiner als 1 zu einem sehr kleinem Ergebnis. Der Wert kann so klein werden, dass er von einem Netz nicht mehr gelernt werden kann. Hier spricht man von einem verschwindenden Gradienten.

Das Problem der explodierenden Gradienten lässt sich durch eine sinnvolle Obergrenze beheben. Bei den verschwindenden Gradienten sieht eine Lösung wesentlich schwieriger aus und dieses Thema ist noch immer Gegenstand der Forschung.

2.2.3. Problem der Langzeit-Abhängigkeiten

Wie bereits erwähnt sind RNNs in der Lage Sequenzen zu erkennen und mit Abhängigkeiten zu arbeiten, doch diese Fähigkeit ist leider begrenzt. Besteht nur eine kleine zeitliche Lücke zwischen den von einander abhängigen Daten, ist ein RNN in der Lage diesen Zusammenhang zu erkennen und die richtigen Schlüsse zu ziehen. Wird der zeitliche Abstand zwischen Eingabe der Daten und dem Zeitpunkt an dem sie für ein Ergebnis benötigt werden jedoch sehr groß kann ein RNN diesen Zusammenhang nicht mehr herstellen. Als Beispiel gibt [Olah 2015] in seinem Artikel ein Sprach-Model an, welches das nächste Wort abhängig vom Vorherigen vorhersagt. Ein RNN ist in der Lage im Satz „Die Wolken sind im Himmel.“ das letzte Wort vorauszusagen, da der Abstand von Himmel und Wolken sehr klein ist. Im Text „Ich bin in Frankreich aufgewachsen. … Ich spreche fließend französisch.“ kann der Abstand zum letzten Wort aber sehr groß sein und die vorherigen Wörter lassen lediglich den Schluss zu, dass eine Sprache folgen muss. Denn Kontext, dass es sich sehr wahrscheinlich um französisch handelt,

erhält man nur durch den ersten Satz. Ein RNN kann sich aber keinen ganzen Text merken und somit hier den Zusammenhang von Frankreich und französisch nicht lernen.

Um das Problem der Langzeit-Abhängigkeiten zu lösen, benutzt man Long Short-Term Memory Netze.

2.3. Long Short-Term Memory Netze

Long Short-Term Memory (LSTM) Netze sind eine besondere Art von Recurrent Netzwerken, die mit Langzeit-Abhängigkeiten arbeiten können. Sie wurden so entworfen, dass sie speziell dieses Problem lösen, denn Informationen über einen langen Zeitraum zu speichern ist ihr Standardverhalten und nicht etwas was mühsam erlernt werden muss. Sie bestehen aus Speicherzellen, in die Informationen geschrieben und wieder herausgelesen werden können. Mit Hilfe von sogenannten Gates, die geöffnet oder geschlossen werden, entscheidet eine Zelle was gespeichert wird und wann ein Auslesen, Reinschreiben und Löschen erlaubt ist. Diese Gates sind analog und durch eine Sigmoid-Funktion implementiert, so dass sich ein Bereich von 0 bis 1 ergibt. (Analog hat den Vorteil gegenüber digital dass es differenzierbar ist und somit für die Backpropagation geeignet.) Genau wie die Eingänge bei den Feedforward und

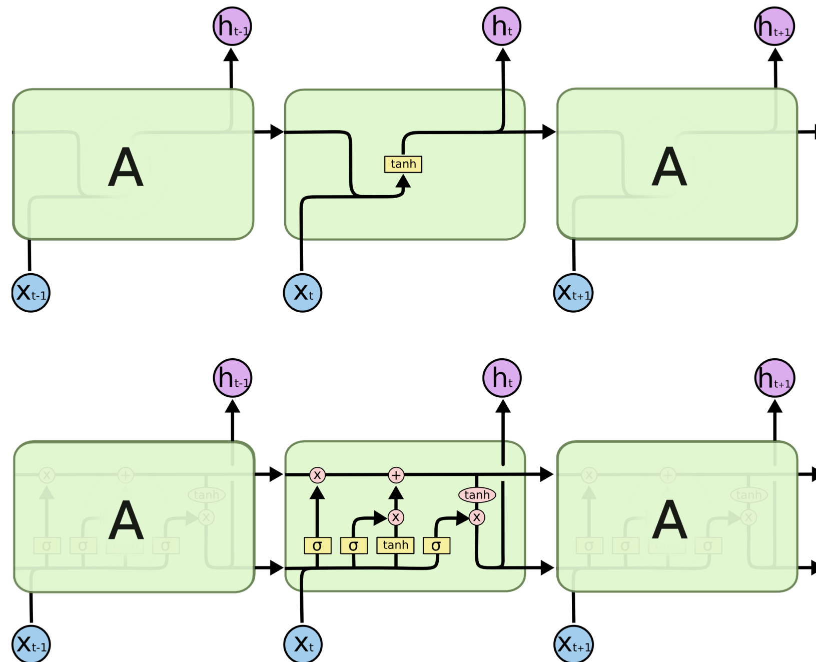

Abb. 2.5.: Vergleich RNN und LSTM (Quelle: [3])

Recurrent Netzen besitzen die Gates Gewichte. Diese Gewichte werden ebenfalls während des Lernprozesses angepasst, so dass die Zelle lernt wann Daten eingelassen, ausgelesen oder gelöscht werden müssen.

Abbildung 2.5 zeigt zum Vergleich oben ein simples Recurrent Netz und unten ein LSTM Netz. Beide sind über drei Zeitschritte dargestellt, wobei der zweite Schritt jeweils ihr Innenleben wiedergibt. Während beim RNN eine simple Struktur mit nur einer Funktion (gelber Kasten in der Abbildung) für das Ergebnis verantwortlich ist, benutzt ein LSTM vier Funktionen. Wie diese Funktionen mit einander agieren und zu einem Ergebnis kommen wird im nächsten Abschnitt Schrittweise erklärt.

2.3.1. Aufbau einer Speicherzelle

Zellzustand

Der Zellzustand ist der eigentliche Speicherort oder das Gedächtnis des LSTM. Abbildung 2.6 zeigt den Verlauf durch eine Speicherzelle. Links wird der Zellzustand vom vorherigen Zeitschritt übernommen und rechts an den nächsten weitergegeben. In der Mitte sind zwei Operation, die den Zustand während dieses Zeitschrittes verändern können. Welche Aufgabe sie haben folgt im Abschnitt Zellzustands Update.

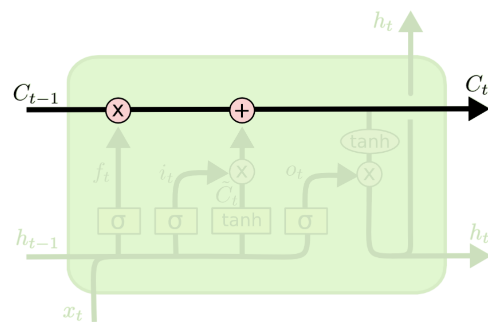

Abb. 2.6.: LSTM Zellzustand (Quelle: [3])

Forget Gate

Das Forget Gate entscheidet mit Hilfe der Sigmoid-Funktion welche Informationen gelöscht werden. Es sieht sich den alten Ausgang h_{t-1} und den neuen Eingang x_t an und gibt für jede Information im Zellzustand C_{t-1} einen Wert zwischen 0 und 1 an. Eine 1 bedeutet behalte es und eine 0 vergiss bzw. lösche es. Der Grund für das Vorhandensein einer Vergissfunktion in einem Baustein, der die Aufgabe hat sich Sachen zu merken, liegt darin dass es manchmal sinnvoll sein kann Dinge zu vergessen. Zum Beispiel kann mit ihrer Hilfe

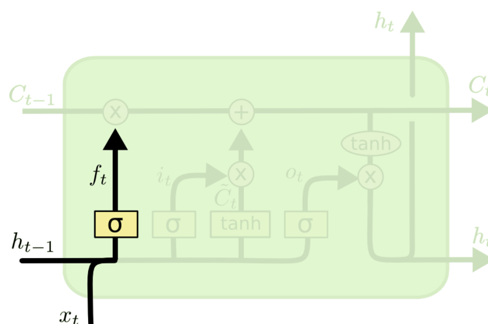

Abb. 2.7.: Forget Gate (Quelle: [3])

die Speicherzelle zurückgesetzt werden, wenn bekannt ist dass die folgenden Daten in keinem Zusammenhang zu den vorherigen stehen.

Eingang

Die Entscheidung, welche Daten gespeichert werden sollen, besteht aus zwei Teilen. Das Eingangsgate ist ebenfalls eine Sigmoid-Funktion und liefert ein Ergebnis zwischen 0 und 1. Sie entscheidet welche Daten zum Zellzustand wie stark durchgelassen werden. Außerdem bereitet eine tanh-Funktion die Daten so auf, dass sie im Zellzustand gespeichert werden können.

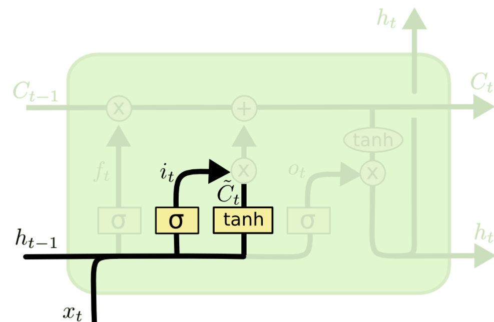

Abb. 2.8.: Eingangsgate (Quelle: [3])

Zellzustands Update

Nachdem das Forget Gate und das Eingangsgate entschieden haben was mit den Daten passieren soll, wird der Zellzustand aktualisiert. Dafür wird der alte Zellzustand C_{t-1} mit dem Ergebnis f_t des Forget Gates multipliziert und somit alles gelöscht, das vergessen werden soll. Anschließend werden die vom Eingangsgate skalierten und von der tanh-Funktion vorbereiteten Daten zum Zellzustand addiert.

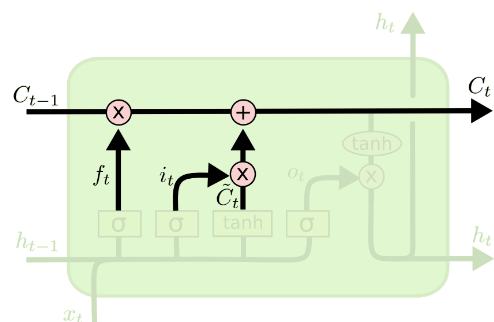

Abb. 2.9.: Zellzustands Update (Quelle: [3])

Ausgabe

Die Ausgabe erfolgt mit Hilfe eines Ausgabegates, welches ebenfalls eine Sigmoid-Funktion ist. Der Zellzustand wird durch eine tanh-Funktion geleitet und anschließend mit dem Ergebnis der Sigmoid-Funktion multipliziert. Die tanh-Funktion wandelt die Werte in einen Bereich von -1 bis 1 um, welches der typische Wertebereich von KNN-Ausgängen ist. Durch die Multiplikation kontrolliert das Ausgangsgate, ob und wie stark das Ergebnis ausgegeben wird.

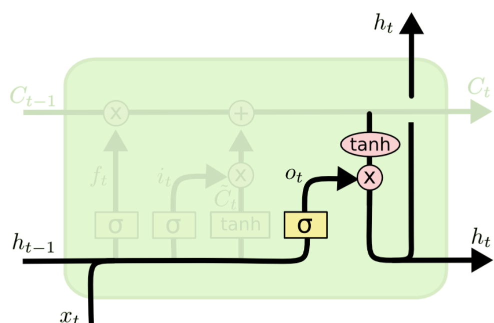

Abb. 2.10.: Ausgabe (Quelle: [3])

Zusammenfassung

Eine Speicherstelle besteht aus einem Zellzustand, der als Gedächtnis fungiert und drei Gates, die den Zellzustand beschützen und kontrollieren. Jedes Gate arbeitet mit einer Sigmoid-Funktion, die einen Wertebereich zwischen 0 und 1 ausgibt und damit die Itensität der Aktion bestimmt. Das Forget Gate ist für das Löschen zuständig, das Eingangsgate übernimmt die Aktion das Neu-Merkens, indem es neue Informationen in den Zellzustand speichert und das Ausgangsgate bestimmt die Informationen, die ausgegeben werden.

2.3.2. LSTM Varianten

Nicht alle LSTM sind so aufgebaut wie bisher beschrieben. Es gibt viele durch kleine Veränderungen leicht abweichende Versionen. Da eine komplette Auflistung den Umfang dieser Arbeit sprengen würden, werden hier nur zwei Varianten vorgestellt um einen Eindruck zu vermitteln, welche Möglichkeiten es gibt.

Gucklöcher

In dieser Variante werden den Gates eine Guckloch-Verbindung hinzugefügt. Dies ermöglicht es den Gates einen Einblick in den aktuellen Zellzustand zu nehmen und die dadurch gewonnenen Informationen in ihre Entscheidung einfließen zu lassen. Abbildung 2.11 zeigt diese Verbindungen für alle drei Gates, jedoch ist dies nicht zwingend notwendig. Es besteht die Möglichkeit auch nur einem oder zwei Gates diese Verbindung zu geben.

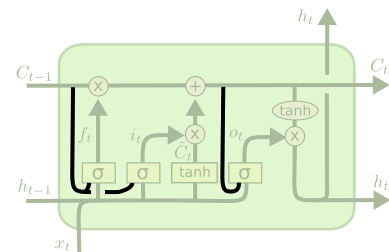

Abb. 2.11.: Variante Gucklöcher (Quelle: [3])

Zusammengeführte Gates

Eine andere Variante ist in Abbildung 2.12 dargestellt und schließt das Forget Gate und das Eingangsgate zu einem Gate zusammen. Diese Veränderung hat die Auswirkung, dass die Entscheidung was gelöscht und was neu gespeichert wird nur noch gemeinsam getroffen werden kann. Somit kann nur etwas vergessen werden, wenn es durch neue Informationen ersetzt wird und im Umkehrschluss können neue Informationen nur gespeichert werden, wenn andere Informationen aus dem Zellzustand gelöscht werden.

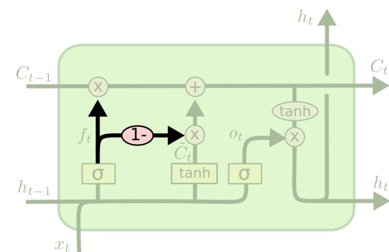

Abb. 2.12.: Variante Zusammengeführte Gates (Quelle: [3])

2.4. Aktueller Forschungsstand

Long Short-Term Memory Netzwerke wurden von Sepp Hochreiter und Jürgen Schmidhuber Mitte der 90er Jahre vorgestellt und haben seitdem immer mehr an Bedeutung im Bereich des maschinellen Lernens gewonnen. In diesem Abschnitt sollen kurz ein paar Beispiele aus dem aktuellen Forschungsstand gegeben werden, bezüglich allgemeiner Verwendungen und speziell der Musikerzeugung mit LSTM-Netzwerken.

2.4.1. Allgemeine Forschung

Analyse von LSTM-Netz-Varianten

[Greff u. a. 2015] führten eine Studie durch, in der sie acht LSTM-Varianten miteinander verglichen. Die Varianten wichen je um nur einen Aspekt vom Standart-LSTM-Netzwerk ab, um ihre Auswirkung sichtbar zu machen. Getestet wurden die Netze in drei repräsentativen Aufgaben: Spracherkennung, Handschrifterkennung und Musikmodellierung. Ihre Ergebnisse zeigten, dass keine der Varianten das Standart-LSTM-Netz signifikant verbesserte und dass das Forget Gate und die Ausgangsaktivierungsfunktion die kritischten Komponenten eines Netzes sind.

LSTM-Netzwerk-basierte Merkmalextraktion

Die menschliche Aktivitätserkennung findet in mobilen Anwendungen ihren Nutzen. Fitness und Gesundheitsprogramme auf Smartphones basieren meistens auf dem Ansatz der gleitenden Fensterprozedur und erstellen Aktivitätshypothesen anhand von aufwendig handgemachten Merkmalen. [Chen u. a. 2016] stellen in ihrer Arbeit eine LSTM-Netzwerk-basierte Merkmalextraktion vor, welche getestet an Datensätzen eine sehr hohe Genauigkeit aufweist und somit als praktikabler Ansatz gilt.

Sprachgenerierung für gesprochene Dialogsysteme

Die Sprachgenerierung für gesprochene Dialogsysteme erfolgt zur Zeit hauptsächlich über regelbasierte Ansätze und obwohl diese Systeme robust und zuverlässig arbeiten, ist ein Gespräch auf Grund von häufigen Wiederholungen identischer Ausgaben oft eintönig. [Wen u. a. 2015] benutzen für ihren Generator ein LSTM-Netzwerk, mit dem Ziel eine größere Varianz in der Satzbildung zu bekommen. Menschliche Richter stuften dieses System höher in Informativität und Natürlichkeit ein und bevorzugten es gegenüber anderen Systemen.

2.4.2. Musikerzeugung

Musiksequenzmodellierung mit LSTM-RBM-Netzwerk

Das Modell von [Lyu u. a. 2015] verbindet die Fähigkeiten von LSTM-Netzwerken und Restricted Boltzmann Maschinen (RBM). Während das Kurzzeitgedächtnis für das Erstellen einer Melodie zuständig ist, sorgt das Langzeitgedächtnis des LSTM-Netzwerkes für ein stimmiges Gesamtergebnis. Die Restricted Boltzmann Maschine ist für die hochdimensionale Datenmodellierung verantwortlich. Dieser Ansatz verbessert die Leistungsfähigkeit von polyphoner Musiksequenzmodellierung erheblich.

Googles Magenta

[Magenta] ist ein Deep Learning Musikprojekt von Google mit dem Ziel unwiderstehliche Musik zu generieren. Es ist in Python geschrieben, benutzt die Open Source Software Tensorflow und arbeitet mit MIDI-Dateien. Seit Juni 2016 steht es als Open Source Projekt zur Verfügung und enthält zur Zeit die Implementation eines RNN und zweier LSTM-Netzwerke. Magenta kann zu diesem Zeitpunkt einen einzelnen String von Noten generieren. ([Brinkkemper 2016])

BachBot

[BachBot] ist ein Open Source Forschungsprojekt der Universität von Cambridge, dessen Ziel es ist Musikstücke zu erzeugen, die von Bachs Werken nicht zu unterscheiden sind. Die Webseite bietet einen Test an, bei dem man sich zwei Musiksequenzen anhören und dann raten kann, bei welchen es sich um ein Original von Bach handelt. Dieses Projekt ist in Python geschrieben, benutzt LSTM-Netzwerke und arbeitet mit wav-Dateien.

GRUV

[GRUV] ist eine Forschungsarbeit der Universität Stanford von [Vitelli u. Nayebi 2015], welche LSTM-Netzwerke und Gated Recurrent Unit (GRU) Netzwerke zur algorithmischen Musikgenerierung vergleicht. Das Fazit der Autoren lautet, dass die Ausgabe des LSTM-Netzwerkes signifikant musikalisch plausibler ist als die Ausgabe des GRU-Netzes. Dieses Projekt ist ebenfalls in Python verfasst und arbeitet mit waveform-Dateien.

3. DeepLearning4J

DeepLearning4J (DL4J) ist eine Open-Source Deep Learning Bibliothek für die Java Virtual Machine. [DL4J] stellt mehrere Beispielprogramme zur Verfügung, wie z.B. Datenklassifizierung mit Feedforward Netzwerk, XOR-Netzwerk mit manueller Erstellung von einem simplen DataSet oder zufällige Texterzeugung im Shakespeare Schreibstil.

Die Entwickler empfehlen eine Benutzung der Tool-Kombination IntelliJ IDEA, Apache Maven und Git für ein komfortables Arbeiten und falls benötigt, eine erleichterte Hilfestellung via Chat.

Dieses Kapitel ist unterteilt in die zwei Bereiche Netzwerke erstellen und Netzwerke trainieren. Im ersten Abschnitt werden Implementierungsmöglichkeiten aufgezeigt und ein kurzer Überblick zu den zur Verfügung stehenden Layertypen gegeben. Der zweite Abschnitt befasst sich mit dem Trainieren von Netzwerken und dem besonderen Datenformat, in welches die Trainingsdaten gebracht werden müssen.

Wie ein neues DL4J Projekt in IntelliJ aufgesetzt werden kann, wird im Anhang A erläutert.

3.1. Netzwerke erstellen

Ein Neuronales Netz wird in DL4J durch drei Komponenten erstellt. Die Komponenten sind der NeuralNetConfiguration.Builder, der ListBuilder und die MultiLayerConfiguration. Nachfolgend werden zwei Beispiele gegeben wie die Implementation aussehen kann. Die 3-Schritt-Methode zeigt die Implementation jeder Komponente einzeln und die Kurzversion fasst die drei Schritte zusammen, was Codezeilen spart, aber für Neulinge vermutlich etwas schwerer verständlich ist.

An zu merken ist noch, dass es sich bei dem gezeigten Code nicht um dasselbe Netzwerk handelt, sondern zwei verschiedene Netzwerke gezeigt werden. Da es hier nicht um einen Vergleich der beiden Methoden geht, sondern nur gezeigt werden soll in welcher Form eine Implementation möglich ist. Beide Codeauszüge stammen von den Netzwerk-Beispielen, welche [DL4J] zum Download zur Verfügung stellt.

3.1.1. Ein Netzwerk erstellen (3-Schritt-Methode)

Schritt 1: NeuralNetConfiguration.Builder

Mit Hilfe des NeuralNetConfiguration.Builder kann man die Netzparameter festlegen. Der Quellcode 3.1 enthält hierzu einen Auszug aus einem Beispielprogramm von [DL4J].

```
1  NeuralNetConfiguration.Builder builder = new NeuralNetConfiguration.
   Builder();
2  builder.iterations(10);
3  builder.learningRate(0.001);
4  builder.optimizationAlgo(OptimizationAlgorithm.
   STOCHASTIC_GRADIENT_DESCENT);
5  builder.seed(123);
6  builder.biasInit(0);
7  builder.miniBatch(false);
8  builder.updater(Updater.RMSPROP);
9  builder.weightInit(WeightInit.XAVIER);
```

Quellcode 3.1: NeuralNetConfiguration.Builder Beispiel

Dieser Code enthält lediglich eine Auswahl aller einstellbaren Parameter, welche in der folgenden Tabelle 3.1 zeilenweise erklärt werden. Für Informationen zu weiteren verfügbaren Parametern kann die DL4J Dokumentation zu Rate gezogen werden.

Zeile	Parameter	Beschreibung
2	iterations(int)	Anzahl der Optimierungsdurchläufe
3	learningRate(double)	Lernrate (Defaulteinstellung: 1e-1)
4	optimizationAlgo(OptimizationAlgorithm)	benutzter Optimierungsalgorithmus (z. B.: Conjugate Gradient, Hessian free, ...)
5	seed(long)	Ursprungszahl für Zufallszahlengenerator (wird zur Reproduzierbarkeit von Durchläufen benutzt)
6	biasInit(double)	Initialisierung der Netzwerk Bias (Default: 0.0)
7	miniBatch(boolean)	Eingabeverarbeitung als Minibatch oder komplettes Datenset. (Default: true)
8	updater(Updater)	Methode zum aktualisieren des Gradienten (z.B.: Updater.SGD = standard stochastic gradient descent)
9	weightInit(WeightInit)	Initiallationsschema der Gewichte (z.B.: normalized, zero, ...)

Tabelle 3.1.: Übersicht einiger Netzwerkparameter

Schritt 2: ListBuilder

Mit Hilfe des erstellten NeuralNetConfiguration.Builders kann ein ListBuilder erstellt werden (siehe Quellcode 3.2). Der ListBuilder ist für die Netzstruktur zuständig und verwaltet die Netzwerk-Layer. Beim Erstellen wird ihm die Anzahl aller verwendeten Layer mitgeteilt. (In diesem Beispiel wurde das Input Layer als Hidden Layer mitgezählt, wodurch bei der Übergabe der Layeranzahl lediglich das Output Layer hinzuaddiert werden muss.)

```
ListBuilder listBuilder = builder.list(HIDDEN_LAYER_CONT + 1);
```

Quellcode 3.2: Erstellen des ListBuilders

Quellcode 3.3 zeigt das Erzeugen der einzelnen Layer für ein RNN. RNNs nutzen in DL4J den GraveLSTM.Builder zum Erzeugen des Input und der Hidden Layer (siehe Zeile 2 bis 5). In Zeile 3 wird die Anzahl der Eingangsknoten übergeben, welche für das Input Layer in diesem Beispiel die Anzahl aller zulässigen Buchstaben ist und für die Hidden Layer eine vorher festgelegt Konstante. Zeile 4 gibt die nötigen Verbindungen zum folgenden Layer an, welche zwingend mit der Eingangsgröße des nächsten Layers übereinstimmen muss. Anschließend wird in Zeile 5 die Aktivierungsfunktion festgelegt, bevor in Zeile 6 des erstellte Layer dem ListBuilder übergeben wird.

Das Output Layer wird mit Hilfe des RnnOutputLayer.Builders erstellt (siehe Zeile 9 bis 12). Die übergebene LossFunction in Zeile 9 gibt die Methode an, mit der der Fehler zwischen Netzwerk-Ergebnis und tatsächlichem Ergebnis berechnet wird. Die Aktivierungsfunktion „softmax“ in Zeile 10 normalisiert die Output Neuronen, so dass die Summe aller Ausgaben 1 ist. Zeile 12 gibt die Anzahl der Output Neuronen an, was in diesem Beispiel der Anzahl der zulässigen Buchstaben entspricht.

```
for (int i = 0; i < HIDDEN_LAYER_CONT; i++) {
    GravesLSTM.Builder hiddenLayerBuilder = new GravesLSTM.Builder();
    hiddenLayerBuilder.nIn(i == 0 ? LEARNSTRING_CHARS.size() :
HIDDEN_LAYER_WIDTH);
    hiddenLayerBuilder.nOut(HIDDEN_LAYER_WIDTH);
    hiddenLayerBuilder.activation("tanh");
    listBuilder.layer(i, hiddenLayerBuilder.build());
}

RnnOutputLayer.Builder outputLayerBuilder = new RnnOutputLayer.Builder(
LossFunction.MCXENT);
outputLayerBuilder.activation("softmax");
outputLayerBuilder.nIn(HIDDEN_LAYER_WIDTH);
outputLayerBuilder.nOut(LEARNSTRING_CHARS.size());
```

```
13    listBuilder.layer(HIDDEN_LAYER_CONT, outputLayerBuilder.build());
```

Quellcode 3.3: Erstellen der Netzwerk-Layer

Anschließend kann der ListBuilder abgeschlossen werden (siehe Quellcode 3.4). Hierzu kann festgelegt werden, ob ein Vortrainieren stattfinden (Zeile 1) und/oder Backpropagation angewendet werden soll (Zeile 2).

```
1    listBuilder.pretrain(false);
2    listBuilder.backprop(true);
3    listBuilder.build();
```

Quellcode 3.4: Fertigstellen des ListBuilder

Schritt 3: MultiLayerNetwork

Wurde die Vorarbeit mit dem NeuralNetConfiguration.Builder und ListBuilder erledigt, kann wie im Quellcode 3.5 ein Netzwerk erstellt werden. Hierfür wird das MultiLayerNetwork verwendet, welches alle Informationen als MultiLayerConfigurations vom ListBuilder erhält. Nach der Initialisierung (Zeile 3) ist das Netzwerk bereit trainiert zu werden.

```
1    MultiLayerConfiguration conf = listBuilder.build();
2    MultiLayerNetwork net = new MultiLayerNetwork(conf);
3    net.init();
```

Quellcode 3.5: Ein Netz erzeugen

MultiLayerNetwork wird in DL4J für Netzwerke benutzt, die im Groben eine einzige Bearbeitungsrichtung haben (ausgenommen netztypische Rückführungen) und Daten vom Eingang ohne Umwege zum Ausgang weiterreichen. Für komplexere Netzwerkarchitekturen stellt DL4J die Klasse ComputationGraph zur Verfügung, welche eine willkürlich gerichtete azyklische Grafenverbindungsstruktur zwischen den Layern erlaubt. Da dies in dieser Arbeit aber nicht zur Anwendung kommt, soll hier nicht weiter auf die Implementierung eingegangen werden.

3.1.2. Ein Netzwerk erstellen (Kurzversion)

In diesem Beispiel wird ein Feedforward Netzwerk mit zwei Layern erstellt. Bei der Implementation des Quellcodes 3.6 wird auf die Unterteilung der Komponenten verzichtet und alle benötigten Netzwerkparameter sowie die Netzstruktur werden direkt in die MultiLayerConfiguration geschrieben ohne Variablen für den NeuralNetConfiguration.Builder und ListBuilder anzulegen.

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .iterations(1)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .learningRate(learningRate)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .list(2)
    .layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(numHiddenNodes)
        .weightInit(WeightInit.XAVIER)
        .activation("relu")
        .build())
    .layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
        .weightInit(WeightInit.XAVIER)
        .activation("softmax").weightInit(WeightInit.XAVIER)
        .nIn(numHiddenNodes).nOut(numOutputs).build())
    .pretrain(false).backprop(true).build();

MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
```

Quellcode 3.6: Netzwerk erstellen (Kurzversionbeispiel)

3.1.3. Layertypen

DL4J stellt verschiedene Layerarten zum problemspezifischen Aufbau von Netzwerken zur Verfügung. So benutzt ein Beispiel von [DL4J] sechs verschiedene Layertypen für ein Netzwerk, welches jeden Frame eines Videos klassifiziert.

Für ein simples Feedforward Netzwerk kann der DenseLayer.Builder() für das Input und die Hidden Layer verwendet werden. Das Output Layer kann mittels OutputLayer.Builder() implementiert werden.

Zur Zeit stellt [DL4J] kein spezielles Layer zur Implementierung von Recurrent Netzen zur Verfügung und arbeiten in ihren Beispielen mit dem GraveLSTM.Builder() für das Input und die Hidden Layer. Für das Output Layer steht der RnnOutputLayer.Builder() zur Verfügung.

LSTM Netzwerke werden ebenfalls mit dem GraveLSTM.Builder() und dem RnnOutputLayer.Builder() implementiert.

3.2. Netzwerke trainieren

3.2.1. Ein Netzwerk trainieren

Ein erstelltes Netzwerk lässt sich durch die Methode fit() trainieren. Quellcode 3.7 zeigt die Implementierung für das Netzwerkmodel net mit Trainingsdaten im Format DataSet. Ein Trainieren eines Netzes ist in DL4J nur mit diesem Datenformat möglich und eine Umwandlung der Ursprungsdaten somit unumgänglich.

```
net.fit(trainingData);
```

Quellcode 3.7: Ein Netz trainieren

3.2.2. Daten erstellen

Neuronale Netze in DL4J arbeiten mit dem Datenformat DataSet. Ein DataSet besteht aus Eingabedaten und Ausgabedaten vom Typ INDArray. Quellcode 3.8 zeigt eine Erstellung eines DataSets für ein KNN, welches die XOR-Funktion erlernen soll.

```
INDArray input = Nd4j.zeros(4, 2);
INDArray labels = Nd4j.zeros(4, 2);

// ({Sample Index, Input Neuron}, Value)
input.putScalar(new int[] { 0, 0 }, 0);
input.putScalar(new int[] { 0, 1 }, 0);
// ({Sample Index, Output Neuron}, Value)
labels.putScalar(new int[] { 0, 0 }, 1);
labels.putScalar(new int[] { 0, 1 }, 0);

input.putScalar(new int[] { 1, 0 }, 1);
input.putScalar(new int[] { 1, 1 }, 0);
labels.putScalar(new int[] { 1, 0 }, 0);
labels.putScalar(new int[] { 1, 1 }, 1);

// sample data 2 and 3
{...}

DataSet ds = new DataSet(input, labels);
```

Quellcode 3.8: Daten erstellen

Zeile 1 erstellt ein INDArray für die Eingabedaten mit den Größen 4 (Anzahl der Trainingsbeispiele) und 2 (Anzahl der Eingangsneuronen) und initialisiert dieses mit Nullen. Das gleiche

geschieht in Zeile 2 für die Ausgangsdaten, welche von [DL4J] meist als Labels benannt sind. Obwohl ein XOR eigentlich nur einen Ausgang braucht, werden in diesem Beispiel zwei verwendet, wobei Neuron 0 für „false“ steht und Neuron 1 für „true“.
Zeile 5 und 6 bilden die Eingangsdaten der zwei Neuronen für das erste Trainingsbeispiel. Beide Neuronen werden mit dem Wert Null belegt und somit ergibt sich durch XOR-Logik, dass das Ergebnis „false“ sein muss. Dies ist in den Zeilen 8 und 9 umgesetzt.
Zeilen 11 bis 14 zeigen die Implementation des zweiten Trainingsbeispiels. Hier erhält das Eingangsneuron 0 den Wert 1 und das Eingangsneuron 1 den Wert 0. Dadurch wird das XOR-Ergebnis „true“ und Ausgangsneuron 1 (welches für „true“ steht) wird mit dem Wert 1 belegt. Die Implementierung der Trainingsbeispiele 2 und 3 erfolgt der XOR-Logik folgend und anschließend wird mit den nun vollständigen Eingangsdaten (input) und Ausgangsdaten (labels) in Zeile 19 ein neues DataSet erzeugt.

Auf die Benutzung von DataSets für LSTM-Netzwerke wird im nächsten Kapitel genauer eingegangen.

4. Musiksequenzen mit Hilfe von DL4J erzeugen

Dieses Kapitel befasst sich mit der Umsetzung eines LSTM-Netzwerkes, welches anhand einer gegebenen Beispielmusik Musiksequenzen generiert und dessen Implementierung in DeepLearning4J (DL4J). Als Musikformat wurden MIDI-Dateien gewählt und es wurden zwei Ansätze zur Verwaltung und Benutzung der Daten verfolgt, wovon der erste nicht zum Ziel führte.

Aufgeteilt ist dieses Kapitel in die Bereiche Eingabe, Netzwerk und Ausgabe. Die Eingabe umfasst die Themen „Was sind MIDI-Dateien?“, wie wurden MIDI-Dateien für diese Arbeit eingelesen (inklusive einer Erläuterung der zwei Implementierungsansätze) und anschließend in das DL4J DataSet-Format gebracht. Der Abschnitt Netzwerk erläutert den Aufbau des verwendeten LSTM-Netzwerkes und geht kurz auf die Unterschiede ein, die durch die zwei Ansätze der Datenverwaltung entstehen. Der Abschnitt Ausgabe geht auf die Bereiche der Ermittlung der Netzwerkausgabe, die Umwandlung der Ausgabe in MIDI-Dateien und die vom Netzwerk erzeugten Ergebnisse ein.

4.1. Die Eingabe

4.1.1. Was sind MIDI-Dateien?

Das Musical Instrument Digital Interface (MIDI) Format wird zum Speichern von Audiodateien benutzt. Im Gegensatz zu anderen Formaten enthält eine MIDI-Datei eine Liste von Ereignissen, die zum Beispiel von einer Soundkarte in entsprechende Töne umgewandelt werden können.

> „Dadurch sind die MIDI-Dateien sehr viel kleiner als digitale Audiodateien, und die Ereignisse und Klänge sind editierbar, wodurch die Musik neu arrangiert, editiert und interaktiv Komponiert werden kann.“ - - - [ITWissen]

MIDI-Dateien bestehen aus einer Sequenz, die einen oder mehrere Tracks beinhaltet, welchen wiederum Ereignisse zugeordnet sind. Diese Ereignisse können als Nachrichten ausgelesen

werden und besitzen eine MIDI-Zeit, die in Ticks angegeben ist. Abbildung 4.1 zeigt den Aufbau so einer Nachricht, welche aus zwei Teilen, Status und Daten, besteht. Das Statusbyte beginnt immer mit einer 1, gefolgt von drei Bits, die die Art der Nachricht enthalten (in Abbildung 4.1 mit s gekennzeichnet). Die letzten vier Bits geben einen von 16 möglichen Kanälen an (in Abbildung 4.1 mit n gekennzeichnet). Die Datenbytes beginnen immer mit einer 0 und geben somit Raum für 128 mögliche Werte.

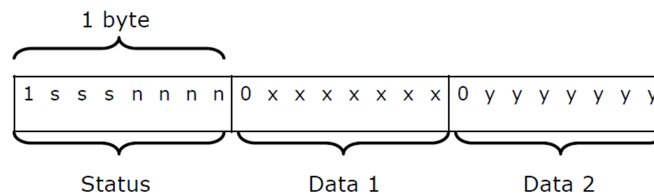

Abb. 4.1.: MIDI Nachricht (Quelle: [1])

Tabelle 4.1 zeigt beispielhaft den Aufbau der Nachrichten „Note aus“ und „Note an“. Das n im Statusbyte steht für die Kanalnummer, welche Hexadezimal angegeben wird und somit einen Bereich von 0 bis F umfasst. Daten 1 enthält eine der 128 möglichen Noten und Daten 2 die Geschwindigkeit bzw. die Intensität mit der die Note gespielt oder losgelassen wird. Eine „Note an“-Nachricht mit der Geschwindigkeit 0 ist gleichbedeutend zu „Note aus“. ([Bello])

Nachricht	Status	Daten 1	Daten 2
Note aus	8n	Notennummer	Geschwindigkeit
Note an	9n	Notennummer	Geschwindigkeit

Tabelle 4.1.: MIDI Nachrichtenformat

4.1.2. MIDI-Dateien lesen

Zum Einlesen einer MIDI-Datei wurde eine Java-Klasse geschrieben, die die Tracks einer Sequenz durchläuft und alle „Note an“ und „Note aus“-Ereignisse in richtiger Reihenfolge abspeichert. Um die Komplexität dieses Projektes am Anfang zu verkleinern, wurde auf die Erfassung der MIDI-Zeit erstmal verzichtet und der Schwerpunkt auf die reine Ereignisabfolge gelegt. Geplant war diese Komponente zu berücksichtigen, sobald das LSTM-Netzwerk in der Lage wäre eine harmonische Musiksequenz anhand von gelernten Notenfolgen zu erzeugen. Da dieses Ziel aber nicht zufriedenstellend erreicht werden konnte, blieb hierfür leider keine Zeit mehr.

Für das Abspeichern der Ereignisse wurden die folgenden zwei Ansätze implementiert:

Variante 1: Ereignisobjekt

Die erste Idee der Datenverwaltung war die eines Ereignisobjektes. Da ein MIDI-Ereignis sich aus drei Hauptkomponenten, Note, Geschwindigkeit und Typ (an/aus) zusammensetzt, wurden diese als Variablen gewählt und die Berücksichtigung des MIDI-Kanals vorerst ausgelassen. Quellcode 4.1 zeigt den Konstruktor für so ein Objekt. In ein solches Objekt werden die Ereignisse einer MIDI-Datei komponentenweise zu Beginn eingelesen. Dadurch erhält man drei Listen in der Größe des geladenen Musikstückes, bzw. der Anzahl der Ereignisse dieses Stückes.

```
1      public MidiEvents(){
2          keys = new ArrayList<Integer>();
3          velocities = new ArrayList<Integer>();
4          eventTypes = new ArrayList<Integer>();
5      }
```

Quellcode 4.1: MidiEvents Konstruktor

Diese Listen enthalten in vermutlich allen Fällen doppelte Werte, da es unwahrscheinlich ist, dass ein Musikstück nur aus unterschiedlichen Noten und Geschwindigkeiten besteht. Somit muss noch eine Ermittlung der tatsächlich benutzten Werte erfolgen. Für den Ereignistyp ist bekannt, dass er nur zwei Werte annehmen kann. Die Ermittlung der Anzahl der verschiedenen Noten und Geschwindigkeiten wurde wie im Quellcode 4.2 gezeigt umgesetzt.

```
1    LinkedHashSet<Integer> validKeys = new LinkedHashSet<Integer>();
2    for (int i : midiEvents.keys){
3      validKeys.add(i);
4    }
5    VALID_KEY_LIST.addAll(validKeys);
6
7    LinkedHashSet<Integer> validVelos = new LinkedHashSet<Integer>();
8    for (int i : midiEvents.velocities){
9      validVelos.add(i);
10   }
11   VALID_VELO_LIST.addAll(validVelos);
```

Quellcode 4.2: Ereignisobjekt: Listen von gültigen Daten

Die Liste der Noten (Codezeilen 1 bis 5) und die Liste der Geschwindigkeiten (Codezeilen 7 bis 11) wurden einmal komplett durchlaufen und mit Hilfe eines LinkedHashSet alle benutzten Werte je einmal erfasst. Anschließend erfolgt eine Abspeicherung der einmaligen Werte in eine ArrayList. Würde man diese Reduzierung nicht machen, müsste man das Netzwerk so auslegen,

dass es in der Lage ist den Zusammenhang von 32768 (128*128*2) möglichen Ereignissen zu lernen.

Nun stehen die Eingangsdaten als Objekt zum Erstellen einer DL4J DataSets zur Verfügung und die Größe der Netzwerk-Layer kann durch die Anzahl der verschiedenen Ereignisse in der komprimierten Liste bestimmt werden.

Variante 2: Ereignisschlüssel

Da die erste Variante der Implementierung nicht zum Erfolg führte, musste ein anderer Wege gefunden werden. Der zweite Ansatz verfolgt die Idee einem Ereignis eine eindeutige Zahl (in dieser Arbeit als Schlüssel bezeichnet) zu zuweisen. Vorgesehen für diesen Schlüssel ist ein neunstelliger Integer. Dieser setzt sich zusammen aus einer zweistelligen Kanalnummer, einem einstelligen Ereignistyp, einer dreistelligen Notennummer und einer dreistelligen Geschwindigkeit. Tabelle 4.2 zeigt die verwendete Kodierung der Ereignisschlüssel mit einem Beispiel. Das Beispiel beschreibt das Ereignis „Note an" für die Note A5 (81) mit der Geschwindigkeit 64 auf Kanal 1. Daraus ergibt sich der Schlüssel 001081064.

	Kanalnr.	**Ereignistyp**	**Notennr.**	**Geschwindigkeit**	**Schlüssel**
Stellen	kk	t	nnn	ggg	kktnnnggg
Beispiel	00	1	081	064	001081064

Tabelle 4.2.: Ereignisschlüssel

Das Speichern der Daten aus einer MIDI-Datei erfolgt in eine ArrayList über Integer, wobei die Ereignisse beim Einlesen direkt in die Schlüsselform gebracht werden. Somit erhält man eine Liste aller Ereignisse und kann daraus ermitteln aus wie vielen verschiedenen Ereignissen die Musiksequenz besteht. Quellcode 4.3 zeigt diese Erfassung, wie schon bei der ersten Variante, mit Hilfe eines LinkedHashSet, wobei „track" die Liste aller Ereignisse ist.

```
1  LinkedHashSet<Integer> validInputs = new LinkedHashSet<Integer>();
2  for (int i : track){
3    validInputs.add(i);
4  }
5  VALID_INTEGER_LIST.addAll(validInputs);
```

Quellcode 4.3: Ereignisschlüssel: Liste von gültigen Daten

Auch in dieser Variante stehen nun sowohl die Eingangsdaten zum Erstellen eines DL4J DataSets, sowie die Anzahl der verschiedenen Ereignisse zur Verfügung, um die Größen der Netzwerk-Layer zu bestimmen.

4.1.3. DataSets erstellen

DL4J Netzwerke arbeiten mit dem Datenformat DataSet, welches aus einem Input und einem Label (Output) bestehen. Da das Netzwerk daraufhin trainiert werden soll, dass es bei Eingabe eines Ereignisses das Folgeereignis mit der höchsten Wahrscheinlichkeit ausgibt, wird als Label jeweils das folgende Ereignis zugewiesen.

Für die Backpropagation Through Time wird von DL4J die Benutzung von 3-dimensionalen INDArrays für Input und Label gefordert. Hierauf wird in der Beschreibung der zwei Varianten genauer eingegangen.

Anders als erwartet, war dieser Bereich einer der schwierigsten in der Umsetzung und hat viel Zeit in Anspruch genommen. Die Umwandlung von den Daten in ein Format mit dem das Netzwerk arbeiten kann, stellte ein großes Verständnis voraus, welches zu Beginn der Arbeit noch nicht vorhanden war. Dies führte zu mehreren Implementierungsfehlern und auch die sonst sehr ausführliche und verständliche Dokumentation von [DL4J] konnte hier die Verständnislücken nicht füllen. Erst die von [DL4J] zur Verfügung gestellten Code-Beispiele vermittelten ein erstes Gefühl für die richtige Benutzung, reichten aber nicht aus, um bereits jetzt festzustellen, dass die Variante 1 (Ereignisobjekt) wie geplant nicht umsetzbar war.

Variante 1: Ereignisobjekt

Obwohl diese Implementierung zu keinem Ergebnis gekommen ist, soll hier trotzdem weiter darauf eingegangen werden, um auf die begangenen Fehler und gewonnenen Erkenntnisse hinzuweisen.

Quellcode 4.4 zeigt die komplette Methode, die zum Erstellen eines DataSets geschrieben wurde. Zeile 3 und 4 legen die 3-dimensionalen INDArrays für den Input und die Labels an. Als erster Parameter wird die Anzahl der Mini-Batches angegeben. Mini-Batches können in DL4J bei großer Datenmenge in externen Dateien sinnvoll sein. Mit ihnen kann man diese Menge in kleinere Stücke zerlegen und somit in mehreren Schritten dem Netzwerk zuführen. Das hat den Vorteil, dass nicht alle Daten auf einmal geladen werden müssen, sondern immer nur kleinere Teile. In dieser Arbeit wurde auf die Unterteilung in Mini-Batches verzichtet und der erste Parameter ist 1, denn alle Daten werden dem Netzwerk zusammen zugeführt.

Der zweite Parameter ist der Platz für den Eingang (bzw. Ausgang) und wurde als

„Anzahl der verschiedenen Noten“ * „Anzahl der verschiedenen Geschwindigkeiten“ * 2

festgelegt. Dadurch erhält man die Anzahl aller möglichen Ereignisse des benutzten Musikstückes.

Der dritte Parameter gibt die Länge aller Trainingsdaten an und wurde durch die Größe einer der Listen aus dem Ereignisobjekt angegeben.

```
public DataSet createTrainingsData(MidiEvents midiTrack, List<Integer>
    validKeyList, List<Integer> validVeloList) {
    // create input/output arrays: miniBatchSize, nIn/nOut, timeSeriesLength
    INDArray input = Nd4j.zeros(1, validKeyList.size()*validVeloList.size()*
    2, midiTrack.keys.size());
    INDArray labels = Nd4j.zeros(1, validKeyList.size()*validVeloList.size()
    *2, midiTrack.keys.size());

    int samplePos = 0;
    for(int currentKey : midiTrack.keys){
        int currentVelo = midiTrack.velocities.get(samplePos);
        int currentType = midiTrack.eventTypes.get(samplePos);
        int nextVelo = midiTrack.velocities.get((samplePos + 1) % midiTrack.
    velocities.size() );
        int nextKey = midiTrack.keys.get((samplePos + 1) % midiTrack.keys.
    size() );
        int nextType = midiTrack.eventTypes.get((samplePos + 1) % midiTrack.
    eventTypes.size() );
        input.putScalar(new int[]{0, validKeyList.indexOf(currentKey)*
    validVeloList.indexOf(currentVelo)*currentType, samplePos}, 1);
        labels.putScalar(new int[]{0, validKeyList.indexOf(nextKey)*
    validVeloList.indexOf(nextVelo)*nextType, samplePos}, 1);
        samplePos++;
    }

    return new DataSet(input, labels);
}
```

Quellcode 4.4: Ereignisobjekt: Trainingsdaten

Anschließend wurde eine For-Schleife zum Befüllen des Input und der Label benutzt, welche über alle Trainingsdaten iteriert. Von Zeile 7 bis Zeile 12 werden den Variablen ihre zu diesem Ereignis (bzw. dem Folgeereignis) zugehörigen Werte zu gewiesen, mit der Besonderheit, dass das allerletzte Ereignis das erste Ereignis als Nachfolger erhält.

In Zeile 13 werden die Inputs in das INDArray eingetragen und in Zeile 14 die Labels. An dieser Stelle befindet sich der Fehler im Gedankengang und der Implementierung. Der zweite Parameter müsste eindeutig einem Ereignis zu zuordnen sein, was aber durch die Verwendung des Ereignisobjektes (ohne Umwege) nicht möglich ist. Durch diese Art der Implementierung werden verschiedene Ereignisse mit demselben Identifikator versehen, was zu einem falschen

Datensatz führt. Schon hier hätte auffallen müssen, dass das nicht funktioniert, was leider durch die Unsicherheiten in der DataSet-Erstellung nicht passierte.

Variante 2: Ereignisschlüssel

Das Bemerken des Fehlers in der Variante 1 führte schließlich zur Idee des Ereignisschlüssels. Da hierbei die Ereignisse nicht aus mehreren kombinierbaren Teilen bestehen, war eine eindeutige Identifikation möglich.

Quellcode 4.5 zeigt die Implementierung der Methode zum Erstellen eines DataSets für Ereignisschlüssel. In Zeile 3 und 4 werden die 3-dimensionalen INDArrays für den Input und die Labels angelegt. Wie bei der Variante 1 wurde auf die Benutzung von Mini-Batches verzichtet und der erste Parameter ist 1. Der zweite Parameter, die Anzahl aller möglichen Eingänge, ist durch die Größe der Liste mit den verschiedenen Ereignissen festgelegt. Der dritte Parameter ist die Anzahl aller Ereignisse der eingelesenen Musik.

```
1  public static DataSet createTrainingsData(List<Integer> VALID_INTEGER_LIST,
       ArrayList<Integer> track) {
2      // create input/output arrays: miniBatchSize, nIn/nOut, timeSeriesLength
3      INDArray input = Nd4j.zeros(1, VALID_INTEGER_LIST.size(), track.size());
4      INDArray labels = Nd4j.zeros(1, VALID_INTEGER_LIST.size(), track.size())
       ;
5
6      int samplePos = 0;
7      for(int currentInt : track){
8          int nextInt = track.get((samplePos + 1) % track.size() );
9          input.putScalar(new int[]{0, VALID_INTEGER_LIST.indexOf(currentInt),
        samplePos}, 1);
10         labels.putScalar(new int[]{0, VALID_INTEGER_LIST.indexOf(nextInt),
       samplePos}, 1);
11         samplePos++;
12     }
13
14     return new DataSet(input, labels);
15 }
```

Quellcode 4.5: Ereignisschlüssel: Trainingsdaten

Das Befüllen erfolgt mit Hilfe einer For-Schleife, die einmal über alle Ereignisse der eingelesenen Musik iteriert. Auch hier wird dem letzten Ereignis das Erste als Nachfolger zugewiesen. In den Zeilen 9 und 10 erhalten das Input und Label INDArray ihre Werte. Anzumerken ist hier,

dass das DataSet lediglich einen Index auf das Ereignis in der Auflistung der verschiedenen Ereignisse erhält und nicht der Ereignisschlüssel übergeben wird.

4.2. Das Netzwerk

Das benutzte LSTM-Netzwerk besteht aus einem Eingangs-, einem Ausgangs- und einem Hiddenlayer. Es wurden verschiedene Parametrierungen getestet, um ein bestmögliches Ergebnis zu erzielen. Die Varianten des Ereignisobjektes und der Ereignisschlüssel unterscheiden sich in der Größe der Eingangs- und Ausgangslayer, wie nachfolgend erläutert.

Variante 1: Ereignisobjekt

Für das Eingangs- und Ausgangslayer dieser Variante wurde die Größe, wie im Quellcode 4.6 gezeigt, als

„Anzahl der verschiedenen Noten“ * „Anzahl der verschiedenen Geschwindigkeiten“ * 2

festgelegt, was oftmals mehr als benötigt ist, da nicht alle möglichen Kombinationen in einer Musikdatei vorkommen müssen.

```
nIn = VALID_KEY_LIST.size() * VALID_VELO_LIST.size() * 2;
nOut = VALID_KEY_LIST.size() * VALID_VELO_LIST.size() * 2;
```

Quellcode 4.6: Ereignisobjekt: Größe der Ein- und Ausgangslayer

Diese Festlegung bedeutete für ein Musikstück (welches als Trainingsdaten benutzt wurde), die Layergröße von 16 Noten * 4 Geschwindigkeiten * 2 = 128 Eingänge/Ausgänge.

Variante 2: Ereignisschlüssel

In der zweiten Variante wurde die Größe, wie im Quellcode 4.7 gezeigt, als Anzahl aller möglichen Ereignisschlüssel bestimmt.

```
nIn = VALID_INTEGER_LIST.size();
nOut = VALID_INTEGER_LIST.size();
```

Quellcode 4.7: Ereignisschlüssel: Größe der Ein- und Ausgangslayer

Für dasselbe Musikstück aus Variante 1, werden hier 36 Eingänge bzw. Ausgänge benötigt.

Dies zeigt, dass die Wahl der Verwaltung der Daten eine Rolle in der Netzwerkgröße spielt und gut durchdacht werden sollte.

4.3. Die Ausgabe

4.3.1. Auswerten der Netzwerkausgabe

Zum Erzeugen einer Ausgabe wird dem Netzwerk ein einzelnes Ereignis als Starteingabe gegeben, woraufhin das Netz ein Folgeereignis ausgibt. Diese Ausgabe wird dem Netzwerk wieder als Eingabe zugeführt und dieser Vorgang wird wiederholt bis die gewünschte Größe der zu erzeugenden Musiksequenz erreicht ist.

Die Auswertung der Netzwerkausgabe erfolgt in beiden Varianten durch die Ermittlung des Ausganges mit dem höchsten Wert.

Variante 1: Ereignisobjekt

Der Quellcode 4.8 zeigt die For-Schleife zum Auslesen der Netzwerkausgabe und dem Erstellen einer Musiksequenz. In Zeile 3 wird ein Array in der Größe der Ausgabe angelegt, das durch Zeile 4 bis 6 mit den Werten des Ausgangslayers gefüllt wird. Anschließend wird in Zeile 7 eine Methode aufgerufen, welche den höchsten Wert ermittelt und den zugehörigen Index des Ausganges zurückgibt. Somit wurde vom Netzwerk für das Eingangsereignis ein Nachfolger bestimmt.

```
1  for (int j = 0; j < sampleSize; j++) {
2      // find the most likeliest output
3      double[] outputProbDistribution = new double[output.length()];
4      for (int k = 0; k < outputProbDistribution.length; k++) {
5          outputProbDistribution[k] = output.getDouble(k);
6      }
7      int sampledEventIdx = findIndexOfHighestValue(outputProbDistribution);
8
9      // add event to output track
10     outputTrack.keys.add(...);
11     outputTrack.velocities.add(...);
12     outputTrack.eventTypes.add(...);
13
14     // use the last output as input
15   {...}
16 }
```

Quellcode 4.8: Ereignisobjekt: Netzausgabe

Danach muss von dem konkreten Ausgang auf ein Ereignis zurückgeschlossen werden und hier wurde der Fehler dieser Implementierung erkannt. Da es bei der DataSet Erstellung zu

keiner eindeutigen Zuordnung von Ereignissen zu Ausgängen kam, war eine Aufschlüsselung (Zeile 10 bis 12) , welches Ereignis als Nachfolger gewählt wurde, an dieser Stelle unmöglich.

Erst an diesem Punkt wurde die richtige Verwendung von DataSets verstanden und der Umstieg in der Implementierung auf die zweite Variante mit dem Ereignisschlüssel folgte.

Variante 2: Ereignisschlüssel

Für die zweite Variante sieht die For-Schleife zum Auslesen der Netzwerkausgabe und dem Erstellen einer Musiksequenz wie in Quellcode 4.9 aus. Zeile 3 bis 7 übernimmt die gleiche Arbeit wie in Variante 1 und ermittelt den Ausgang mit dem Folgeereignis mit der höchsten Wahrscheinlichkeit. In Zeile 10 wird anhand des Ausgangsindex das zugehörige Ereignis aus der Liste alle möglichen Ereignisse ausgelesen und in eine Ausgabeliste eingetragen. An dieser Stelle entsteht durch das Durchlaufen der Schleife eine List von Ereignisschlüsseln, die später zur Erstellung einer MIDI-Datei benutzt werden kann.

```
for (int j = 0; j < sampleSize; j++) {
    // find the most likeliest output
    double[] outputProbDistribution = new double[VALID_INTEGER_LIST.size()];
    for (int k = 0; k < outputProbDistribution.length; k++) {
        outputProbDistribution[k] = output.getDouble(k);
    }
    int sampledEventIdx = findIndexOfHighestValue(outputProbDistribution);

    // add event to output track
    outputTrack.add(VALID_INTEGER_LIST.get(sampledEventIdx));

    // use the last output as input
    INDArray nextInput = Nd4j.zeros(VALID_INTEGER_LIST.size());
    nextInput.putScalar(sampledEventIdx, 1);
    output = net.rnnTimeStep(nextInput);
}
```

Quellcode 4.9: Ereignisschlüssel: Netzausgabe

Anschließend wird in Zeile 13 ein INDArray für die nächste Netzwerkeingabe erstellt. Dieses wird in Zeile 14 mit der letzten Ausgabe befüllt und in Zeile 15 dem Netzwerk übergeben, so dass das Folgeereignis dieses Ereignisses bestimmt werden kann. Dieser Vorgang wiederholt sich solange bis die Anzahl der gewünschten Ereignisse für die zu generierende Musiksequenz erreicht wurde.

4.3.2. MIDI Files schreiben

Nachdem des Netzwerk eine Liste von Ereignissen erzeugt hat, müssen diese noch in eine MIDI-Datei übersetzt werden. Da das Netzwerk zur Zeit nur darauf ausgelegt ist die Ereignisse zu lernen, müssen alle anderen Parameter, wie z.B. Tempo und Art des Instruments manuell festgelegt werden. Auch die Festlegung des Zeitpunktes zu welchen MIDI Tick (das heißt in welchem Abstand zum vorherigen Ereignis) das Ereignis stattfindet wird nicht vom Netzwerk abgedeckt.

Sobald die Rahmenbedingungen festgelegt sind, wird die erzeugte Ereignisschlüsselliste durchlaufen und jedes Element in eine MIDI Nachricht umgewandelt, welche anschließend in eine MIDI-Datei geschrieben werden.

4.3.3. Ergebnisse

Variante 1: Ereignisobjekt

Der erste Ansatz der Datenverwaltung mit einem Ereignisobjekt lieferte aufgrund der speziellen DataSet-Anforderungen kein Ergebnis, da keine vollständige Implementierung möglich war.

Variante 2: Ereignisschlüssel

Die zweite Variante, welche mit Ereignisschlüsseln arbeitet, erbrachte nur einen Teilerfolg. Das Netzwerk war in der Lage zu erlernen, dass auf ein „Note an“-Ereignis irgendwann ein „Note aus“-Ereignis folgen muss, um eine Note nicht endlos lange zu spielen. Auch das Erzeugen desselben Ereignisses direkt aufeinanderfolgend konnte vom Netzwerk oftmals als nicht sinnvoll erkannt werden. Je länger trainiert wurde, um so seltener folgten z.B. „Note an“-Ereignisse für dieselbe Note mit gleicher Geschwindigkeit aufeinander.

Abb. 4.2.: Beispielausgabe 1

Als Trainingsdaten wurden drei unterschiedliche MIDI-Dateien verwendet. Die erste bestand aus 1040 Ereignissen und benutzte 36 verschiedenen Ereignisschlüssel. Mit diesen Vorgaben wurden die Musiksequenzen in Abbildung 4.2 und 4.3 erzeugt. Auffällig ist, dass das Netzwerk sich auf einzelne Noten fixiert und diese sehr oft mehrfach hintereinander spielt. Durch die Implementierung der ersten Variante war bekannt, dass in der Trainingsdatei 16 verschiedene Noten und 4 Geschwindigkeiten vorkamen. Diese Tatsache lässt auf eine geringe Varianz in der Tonfolge schließen und wird als Grund vermutet, warum das Netz nicht in der Lage ist eine Melodie zu erzeugen.

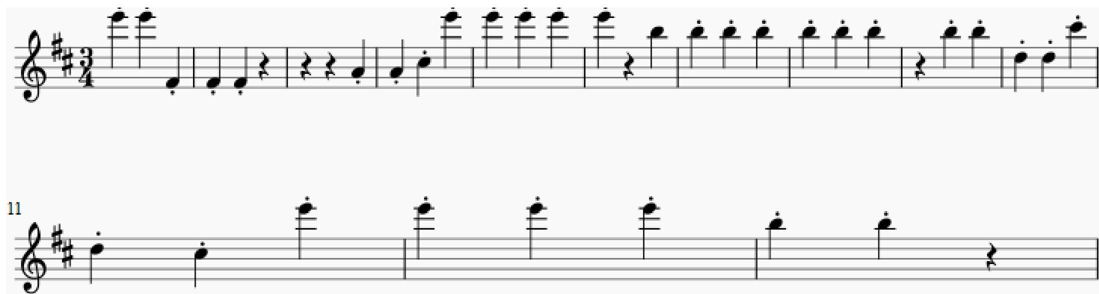

Abb. 4.3.: Beispielausgabe 2

Die zweite Datei bestand aus 1254 Ereignissen und benutzte 417 verschiedene Ereignisschlüssel. Es zeigte sich, dass je mehr Eingänge benutzt wurden, um so kleiner musste die Anzahl der Backpropagation-Schritte sein, da es sonst zu Speicherengpässen kam. Die Ausgabe des Netzes unterschied sich kaum von der Benutzung der ersten MIDI-Datei.

Die dritte Datei bestand aus 2620 Ereignissen und benutzte 1053 verschiedene Ereignisschlüssel. Hierbei traten Speicherplatzprobleme auf und das System brach den Vorgang mit einer Fehlermeldung („Speicher konnte nicht allokiert werden“) ab.

Mehrere Experimente mit verschiedenen Netzwerkparametern führten leider zu keinem zufriedenstellenden Ergebnis. Es ist unklar, ob DL4J für diese Problemstellung komplett ungeeignet ist, der Implementierungsansatz schlecht gewählt oder einfach nicht die richtige Parametrierung gefunden wurde. Auszuschließen ist auch nicht, dass der benutzte Rechner für diese Aufgabe einfach über zu wenig Speicherplatz und Rechenleistung verfügt. Möglicherweise war die Reduzierung der Eingabedaten auf die reine Notenfolge auch zu groß, so dass zu viele Melodie-Informationen verloren gingen.

5. Fazit

Eine Implementierung eines LSTM-Netzwerkes zur Erzeugung von Musiksequenzen in DL4J stellte sich als anspruchsvoller als erwartet heraus. Die Benutzung von DataSets war hierbei eine besondere Herausforderung, kostete viel Zeit und führte aufgrund unzureichender Dokumentation zu Fehlern in der Planung und Umsetzung. Die richtige Benutzung war nicht selbsterklärend und konnte nur durch ein Erlernen durch Ausprobieren erfolgen. Die Erkenntnis, dass DataSets lediglich einen Verweis auf Daten in Form eines Indizes enthalten, kam zu spät, so dass von der ursprünglichen Implementierungsidee (Ereignisobjekt) auf eine andere (Ereignisschlüssel) umgeschwenkt werden musste.

Der mehrstufige Weg von der Eingabe zur Ausgabe (Eingabedaten → Umwandlung → DataSet → Netzwerk → DataSet → Umwandlung → Ausgabedaten) wirkt umständlich und kann je nach Problemstellung sehr aufwendig sein.

Die Netzwerkgröße hängt bei der verwendeten Implementation von der Anzahl der verschiedenen Daten ab. Je mehr Werte die Eingabe annehmen kann, umso größer werden Eingangs- und Ausgangslayer. Ob eine Implementierung anders als „Anzahl der verschiedenen Eingabewerte = Anzahl der Eingänge/Ausgänge“ möglich ist, konnte in der zur Verfügung stehenden Zeit nicht geklärt werden.

Die Trainingsdaten schienen entweder zu klein zu sein, so dass das Netzwerk keine sinnvolle Notenfolge lernen konnte oder sie waren so groß, dass die Backpropagation verringert werden musste oder nicht genügend Speicherkapazität zur Verfügung stand. Trotz verschiedener Durchläufe mit variierenden Parametern konnte kein zufriedenstellendes Ergebnis erzielt werden.

Abschließend lässt sich sagen, dass die Benutzung von DeepLearning4J für diese Problemstellung nicht zwingend ungeeignet sein muss. Mit mehr Zeit, größerer Rechenleistung und Speicherkapazität könnte eine erfolgreiche Umsetzung vielleicht machbar sein. Jedoch ist die Benutzung sehr aufwendig und umständlich, wodurch DL4J nicht als gute Wahl empfohlen werden kann und eher zu einer Umsetzung von Musikerzeugung durch neuronale Netzwerke in Python geraten wird, da es dort bereits mehrere erfolgreiche Projekte gibt.

A. DL4J-Projekt in IntelliJ aufsetzen

Nach erfolgreicher Installation von IntelliJ und Maven kann ein DL4J-Projekt mithilfe von Maven eingerichtet werden. Hierfür wählt man „File“ → „New“ → „Project ...“. Es öffnet sich

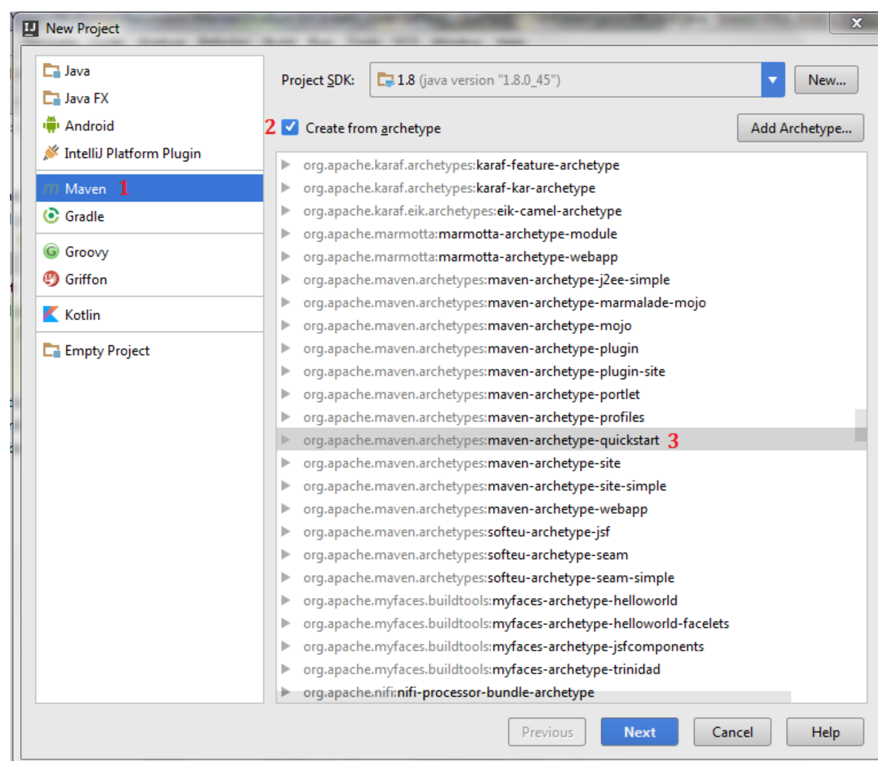

Abb. A.1.: „New Project“ Fenster

das „New Project“-Fenster, wie in Abbildung A.1 abgebildet und man wählt auf der linken Seite „Maven“ (in der Abbildung mit einer roten 1 versehen). Das Kästchen „Create from archetype“ (in Abbildung gekennzeichnet mit 2) wird ausgewählt und aus der Liste, der verfügbaren Typen der Typ „maven-archetype-quickstart“ (in Abbildung gekennzeichnet mit 3) gewählt. Danach auf „Next“ und die GroupId sowie ArtifactId festlegen. Anschließend noch einen Projektname festlegen und „Finish“ drücken.

Nachdem Maven für einen die Projektstruktur erstellt hat, muss man die neu erstellte POM.xml Datei noch anpassen. Die POM.xml enthält die Projektabhängigkeiten, welche je Projekt variieren können. Hier kann man festlegen ob das Projekt auf der CPU oder GPU laufen soll. Das Standard-Backend für CPUs is „nd4j-native“ und kann wie im Listing A.1 in die POM.xml eingefügt werden.

```
<dependency>
  <groupId>org.nd4j</groupId>
  <artifactId>nd4j-native</artifactId>
  <version>${nd4j.version}</version>
</dependency>
```

Quellcode A.1: Backend Dependency CPU

Durch einfügen der Zeilen aus dem Listing A.2, werden dem Projekt die Kernabhängigkeiten von DL4J hinzugefügt. Andere Abhängigkeiten DL4J betreffend, die für einige Projekte sinnvoll sind, wie z.B. parallele Ausführung auf Hadoop oder Spark, stehen zu Verfügung und Informationen hierzu findet man auf [ND4J].

```
    <dependency>
      <groupId>org.deeplearning4j</groupId>
      <artifactId>deeplearning4j-core</artifactId>
      <version>${dl4j.version}</version>
    </dependency>
```

Quellcode A.2: DL4J Dependency

Die Variablen „nd4j.version“ (Listing A.1 Zeile 4) und „dl4j.version“ (Listing A.2 Zeile 4) geben die Versionen an und müssen weiter oben in der POM-Datei zwischen <properties> ... </properties> festgelegt werden. Dies kann wie im Listing A.3 erfolgen.

```
<nd4j.version>0.4-rc3.9</nd4j.version>
<dl4j.version>0.4-rc3.10</dl4j.version>
```

Quellcode A.3: Versionsvariablen

Weitere Informationen bezüglich GPU Betrieb oder Benutzung mit anderen Betriebssystemen können auf der [ND4J] Seite nachgeschlagen werden.

Quellenverzeichnis

Literatur

[ITWissen] URL: http://www.itwissen.info/definition/lexikon/standard-MIDI-format-SMF-SMF-Format.html (besucht am 19. 02. 2017).

[GRUV] Matthew Vitelli und Aran Nayebi. URL: https://github.com/MattVitelli/GRUV (besucht am 02. 02. 2017).

[Vitelli u. Nayebi 2015] Matthew Vitelli und Aran Nayebi. “GRUV: Algorithmic Music Generation using Recurrent Neural Networks”. In: (2015). URL: https://cs224d.stanford.edu/reports/NayebiAran.pdf (besucht am 02. 02. 2017).

[Bello] Juan Pablo Bello. “MIDI code”. In: (). URL: https://www.nyu.edu/classes/bello/FMT_files/9_MIDI_code.pdf (besucht am 13. 02. 2017).

[Breitner 2014] Michael H. Breitner. *Neuronales Netz*. 2014. URL: http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/technologien-methoden/KI-und-Softcomputing/Neuronales-Netz (besucht am 09. 08. 2016).

[Brinkkemper 2016] Frank Brinkkemper. *Analyzing six deep learning tools for music generation*. 2016. URL: http://www.asimovinstitute.org/analyzing-deep-learning-tools-music/ (besucht am 02. 02. 2017).

[DL4J] *Deeplearning4j: Open-source distributed deep learning for the JVM*. Version Apache Software Foundation License 2.0. Deeplearning4j Development Team DL4J. URL: http://deeplearning4j.org/ (besucht am 30. 06. 2016).

[Chen u. a. 2016] Yuwen Chen und Kunhua Zhong und Ju Zhang und Qilong Sun und Xueliang Zhao. "LSTM Networks for Mobile Human Activity Recognition". In: (2016). URL: http://www.atlantis-press.com/php/paper-details.php?from=author+index&id=25849464&querystr=authorstr%3DC (besucht am 08. 09. 2016).

[Magenta] *Magenta: Music and Art Generation with Machine Intelligence.* Google Brain Team. URL: https://github.com/tensorflow/magenta (besucht am 02. 02. 2017).

[BachBot] Feynman Liang und Mark Gotham und Marcin Tomczak und Matthew Johnson und Jamie Shotton. *The BachBot Challenge: Can you tell the difference between Bach and a computer?* URL: http://bachbot.com/#/?_k=y9plwb (besucht am 02. 02. 2017).

[Wen u. a. 2015] Tsung-Hsien Wen und Milica Gasic und Nikola Mrksic und Pei-hao Su und David Vandyke und Steve J. Young. "Semantically Conditioned LSTM-based Natural Language Generation for Spoken Dialogue Systems". In: *CoRR* abs/1508.01745 (2015). URL: http://arxiv.org/abs/1508.01745 (besucht am 08. 09. 2016).

[ND4J] *ND4J: N-dimensional arrays and scientific computing for the JVM.* Version Apache Software Foundation License 2.0. ND4J Development Team. URL: http://nd4j.org/getstarted.html (besucht am 02. 07. 2016).

[Olah 2015] Christopher Olah. *Understanding LSTM Networks.* 2015. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (besucht am 12. 07. 2016).

[Greff u. a. 2015] Klaus Greff und Rupesh Kumar Srivastva und Jan Koutnik und Bas R. Steunebrink und Jürgen Schmidhuber. "LSTM: A Search Space Odyssey". In: *CoRR* abs/1503.04069 (2015). URL: http://arxiv.org/abs/1503.04069 (besucht am 08. 09. 2016).

[Lyu u. a. 2015] Qi Lyu und Zhiyong Wu und Jun Zhu. "Polyphonic Music Modelling with LSTM-RTRBM". In: (2015). URL: http://dl.acm.org/citation.cfm?id=2806383 (besucht am 08. 09. 2016).

Bilder

[1] Juan Pablo Bello. URL: https://www.nyu.edu/classes/bello/FMT_files/9_MIDI_code.pdf (besucht am 13. 02. 2017).

[2] Mikael Boden. In: (2001). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.16.6652&rep=rep1&type=pdf (besucht am 16. 07. 2016).

[3] Christopher Olah. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (besucht am 12. 07. 2016).

[4] Deeplearning4j Development Team. URL: http://deeplearning4j.org/neuralnet-overview (besucht am 30. 06. 2016).

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

Hamburg, 24. Februar 2017 Marina Knabbe