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 Bachelor of Science 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: ?. Februar 2017

Marina Knabbe

Thema der Arbeit
Erzeugung von Musiksequenzen mit LSTM-Netzwerken

Stichworte
Musiksequenzen, LSTM, Neuronale Netze, Maschinenlernen, DeepLearning4J

Kurzzusammenfassung
Dieses Dokument …
Marina Knabbe

Title of the paper
English title

Keywords
keyword 1, keyword 2

Abstract
This document …

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Glossar(unfinished)

...

1. Einleitung (unfinished)

...

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 nächsten 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

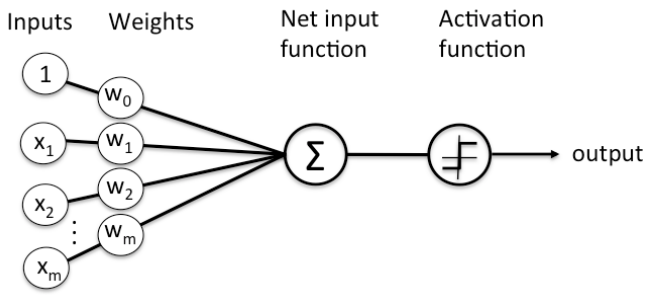

Abb. 2.1.: mögliches Aussehen eines Feedforward Neurons (Quelle: [3])

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

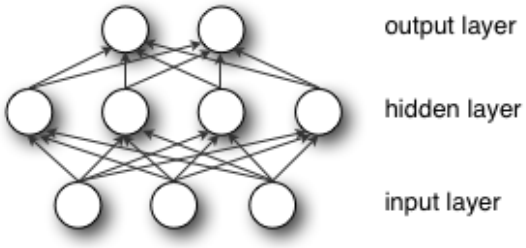

Abb. 2.2.: Aufbau eines Feedforward Netzes (Quelle: [3])

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 Gewichten 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

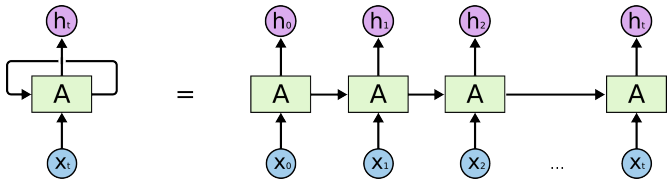

Abb. 2.3.: vereinfachte Darstellung eines RNN Neurons (Quelle: [2])

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

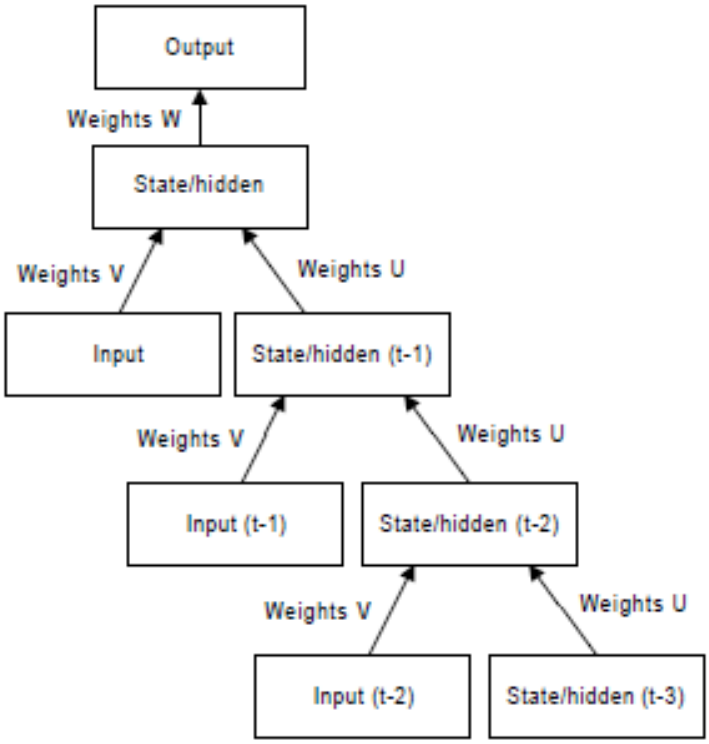

Abb. 2.4.: entrolltes RNN für BPTT (Quelle: [1])

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 das 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

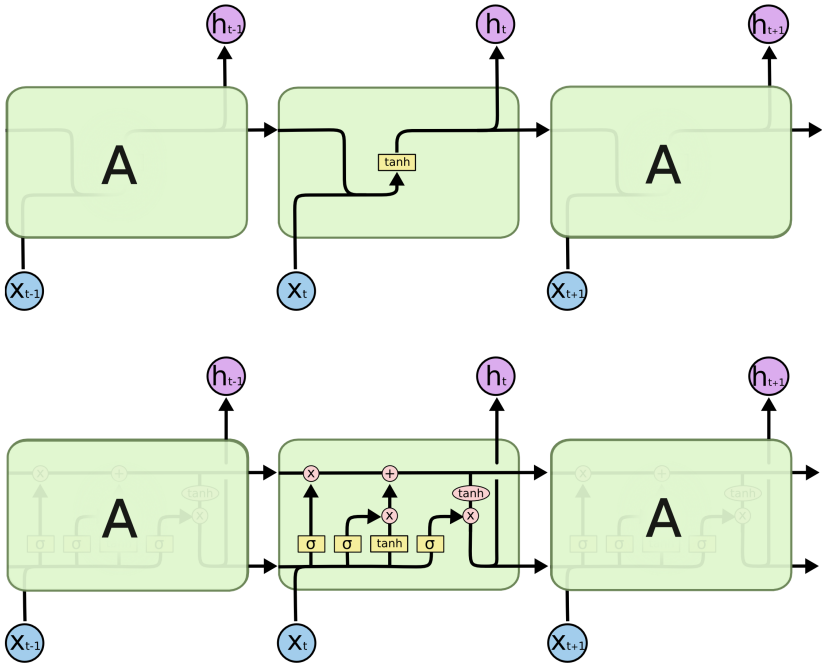

Abb. 2.5.: Vergleich RNN und LSTM (Quelle: [2])

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.

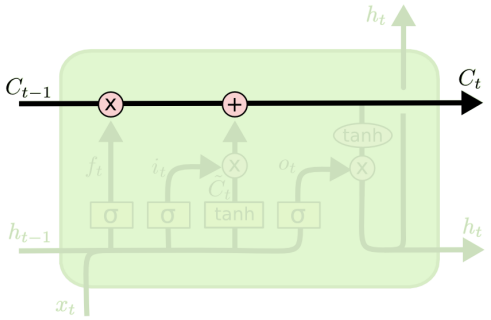

Abb. 2.6.: LSTM Zellzustand (Quelle: [2])

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.

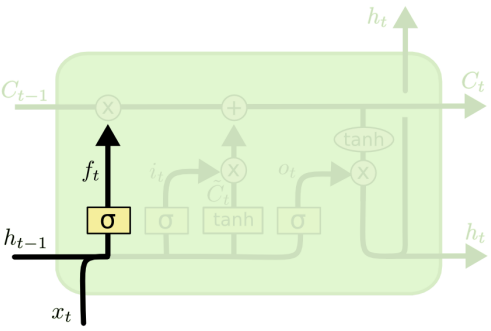

Abb. 2.7.: Forget Gate (Quelle: [2])

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 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.

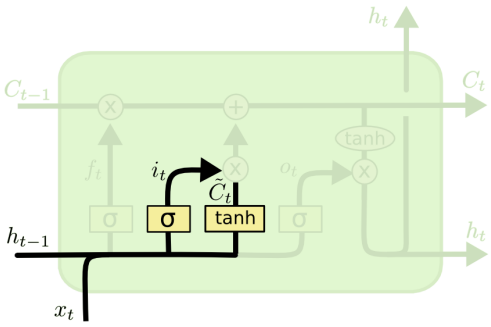

Abb. 2.8.: Eingangsgate (Quelle: [2])

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.

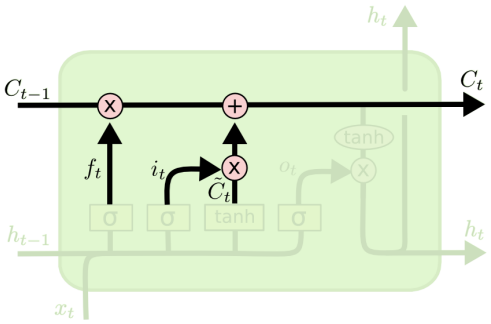

Abb. 2.9.: Zellzustands Update (Quelle: [2])

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.

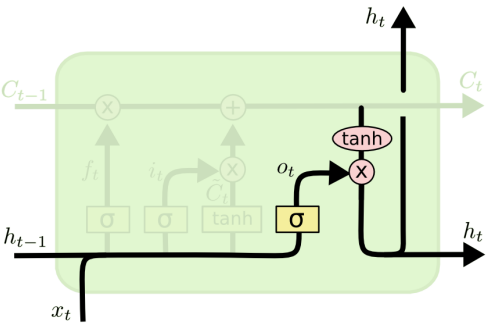

Abb. 2.10.: Ausgabe (Quelle: [2])

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 Gucklöcher-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.

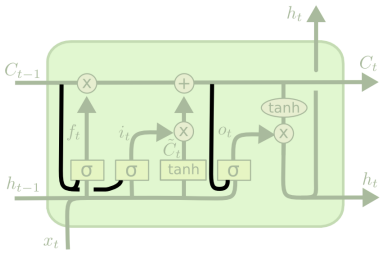

Abb. 2.11.: Variante Gucklöcher (Quelle: [2])

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.

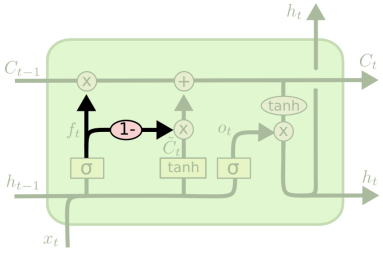

Abb. 2.12.: Variante Zusammengeführte Gates (Quelle: [2])

2.4. Aktueller Forschungsstand (unfinished)

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 representativen Aufgaben: Spracherkennung, Handschrifterkennung und Musikmodelierung. 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 (unfinished)

Musiksequenzmodellierung mit LSTM-RBM-Netzwerk

Das Modell von [Lyu u. a. 2015] verbindet die Fähigkeiten von LSTM-Netzwerken und Restricted Boltzmann Maschinen. 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 hochdimensionalen Datenmodellierung verantwortlich. Dieser Ansatz verbessert die LeistungsfÃ¤higkeit von polyphoner Musiksequenzmodellierung erheblich.

Googles Magenta

Magenta is Googleâ€™s open source deep learning music project. They aim to use machine learning to generate compelling music. The project went open source in June 2016 and currently implements a regular RNN and two LSTMâ€™s. Challenges: At this point, Magenta can only generate a single stream of notes. Efforts have been made to combine the generated melodies with drums and guitars â€“ but based on human input, as of yet. Once a model that can process polyphonic music has been trained, it could start to create harmonies (or at least multiple streams of notes). This would indeed be a mighty step on their quest for the generation of some compelling music. Source: http://www.asimovinstitute.org/analyzing-deep-learning-tools-music/ https://magenta.tensorflow.org/ Programmiersprache: Python in Tensorflow, MIDI File based

BachBot

A research project by Feynman Liang at Cambridge University, also using an LSTM. This time it is used to train itself on Bach chorales. Itâ€™s goal is to generate and harmonize chorales indistinguishable from Bachâ€™s own work. The website offers a test where one can listen to two streams and guess which one is an actual composition by Bach. GitHub: https://github.com/feynmanliang/bachbot/ Great, because: Research found that people have a hard time distinguishing generated Bach from the real stuff. Also, this is one of the best efforts in handling polyphonic music as the algorithm can handle up to four voices. Challenges: BachBot works best if one or more of the voices are fixed. Otherwise the algorithm just generates wandering chorales.The algorithm could be used to add chorales to a generated melody. Source: http://www.asimovinstitute.org/analyzing-deep-learning-tools-music/ Programmiersprache: Python, wav-file based

GRUV

A Stanford research project that, similar to Wavenet, also tries to use audio waveforms as input, but with an LSTMâ€™s and GRUâ€™s rather than CNNâ€™s. They have showed their proof of concept to the world in June 2015. GitHub: https://github.com/MattVitelli/GRUV Great, because: The Stanford researchers were one of the first to show how to generate sounds with an LSTM using raw waveforms as input. Challenges: The demonstration they provide seems over-fitted on a particular song, due to the small training corpus and the sheer amount of layers of the NN. The researchers themselves did not have the time nor computational power to experiment further with this. Fortunately, this void is starting to get filled by researchers from WaveNet and other enthusiasts. Jakub Fiala has used this code to generate an interesting amen drum break, see this blog post. Source: http://www.asimovinstitute.org/analyzing-deep-learning-tools-music/ Programmiersprache: Python, mp3-file based

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 aufgesetz werden kann, wird im Anhang A erläutert.

3.1. Netzwerke erstellen

Ein Neuronales Netz wird in DL4J durch drei Komponeten 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 Paramter, welche in der folgenden Tabelle 3.1 zeilenweise erklärt werden. Für Informationen zu weiteren verfügbaren Paramtern 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 aktuallisieren 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 alse Hidden Layer mit gezählt, wodurch bei der Übergabe der Layeranzahl lediglich das Output Layer hinzuaddiert werden muss.)

```
1    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 Beipiel 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.

```
1    for (int i = 0; i < HIDDEN_LAYER_CONT; i++) {
2        GravesLSTM.Builder hiddenLayerBuilder = new GravesLSTM.Builder();
3        hiddenLayerBuilder.nIn(i == 0 ? LEARNSTRING_CHARS.size() :
     HIDDEN_LAYER_WIDTH);
4        hiddenLayerBuilder.nOut(HIDDEN_LAYER_WIDTH);
5        hiddenLayerBuilder.activation("tanh");
6        listBuilder.layer(i, hiddenLayerBuilder.build());
7    }
8
9    RnnOutputLayer.Builder outputLayerBuilder = new RnnOutputLayer.Builder(
     LossFunction.MCXENT);
10   outputLayerBuilder.activation("softmax");
11   outputLayerBuilder.nIn(HIDDEN_LAYER_WIDTH);
12   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 erhällt. 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 Graphenverbingsstruktur 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 Kurzversion Beispiel

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 RnnOutput-Layer.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.

```
1         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.

```
1  INDArray input = Nd4j.zeros(4, 2);
2  INDArray labels = Nd4j.zeros(4, 2);
3
4  // ({Sample Index, Input Neuron}, Value)
5  input.putScalar(new int[] { 0, 0 }, 0);
6  input.putScalar(new int[] { 0, 1 }, 0);
7  // ({Sample Index, Output Neuron}, Value)
8  labels.putScalar(new int[] { 0, 0 }, 1);
9  labels.putScalar(new int[] { 0, 1 }, 0);
10
11 input.putScalar(new int[] { 1, 0 }, 1);
12 input.putScalar(new int[] { 1, 1 }, 0);
13 labels.putScalar(new int[] { 1, 0 }, 0);
14 labels.putScalar(new int[] { 1, 1 }, 1);
15
16 // sample data 2 and 3
17 {...}
18
19 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 umgesetz.
Zeilen 11 bis 14 zeigen die Implementation des zweiten Trainingsbeispiels. Hier erhällt 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.

4. Musiksequenzen mit Hilfe von DL4J erzeugen (unfinished)

...

4.1. Ïdee”(unfinished)

- LSTM Netz soll mit Beispielmusik im MIDI-Format gefüttert werden und daraufhin Musiksequenzen erzeugen

4.2. Die Eingabe (unfinished)

4.2.1. Was sind MIDI Files? (unfinished)

- Musikdatei, Informationen als Events

4.2.2. MIDI Files lesen (unfinished)

- einlesen eines MIDI Files - Vernachlässigung des MIDI-Takts

4.2.3. DataSets erstellen (unfinished)

- 2 Varianten implementiert - umwandeln der eingelesenen MIDI-Events

Variante 1 (unfinished)

...

Variante 2 (unfinished)

...

Vergleich der Varianten (unfinished)

...

4.3. Das Netz (unfinished)

4.4. Die Ausgabe (unfinished)

4.4.1. MIDI Files schreiben (unfinished)

4.4.2. Ergebnisse (unfinished)

Variante 1 (unfinished)

Variante 2 (unfinished)

4.5. Mögliche Erweiterungen und Verbesserungen (unfinished)

5. Fazit (unfinished)

...

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

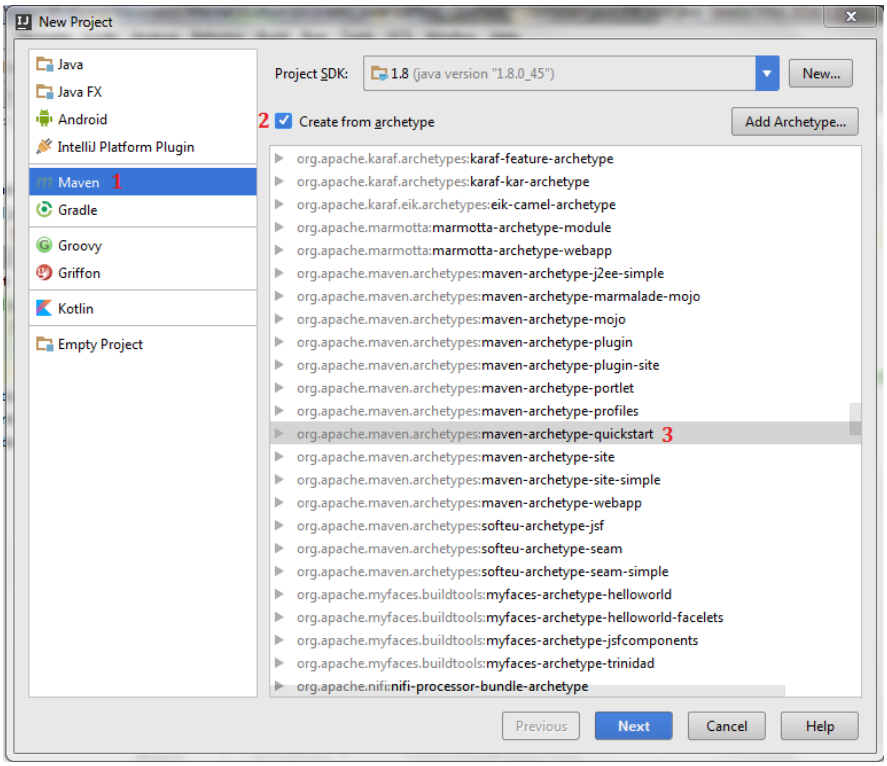

Abb. A.1.: „New Project“ Fenster

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.

```
1 <dependency>
2   <groupId>org.nd4j</groupId>
3   <artifactId>nd4j-native</artifactId>
4   <version>${nd4j.version}</version>
5 </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].

```
1     <dependency>
2       <groupId>org.deeplearning4j</groupId>
3       <artifactId>deeplearning4j-core</artifactId>
4       <version>${dl4j.version}</version>
5     </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.

```
1 <nd4j.version>0.4-rc3.9</nd4j.version>
2 <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

[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).

[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).

[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] 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).

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

[3] 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, ?. Februar 2017 Marina Knabbe