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
SchlÃ¼sselwort 1, SchlÃ¼sselwort 2

Kurzzusammenfassung
Dieses Dokument …
Marina Knabbe

Title of the paper
English title

Keywords
keyword 1, keyword 2

Abstract
This document …

Inhaltsverzeichnis

Abbildungsverzeichnis

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

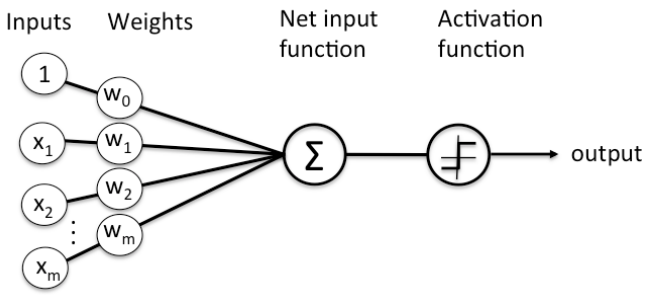

Abb. 2.1.: mögliches Aussehen eines Feedforward Neurons (Quelle: [3])

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

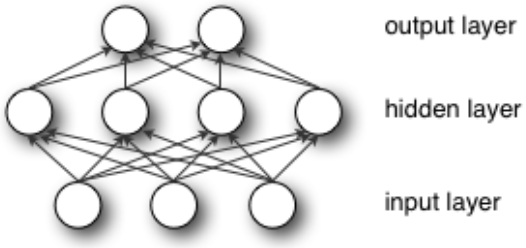

Abb. 2.2.: Aufbau eines Feedforward Netzes (Quelle: [3])

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

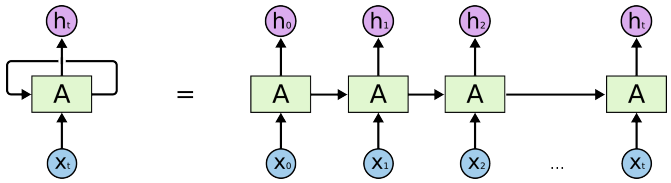

Abb. 2.3.: vereinfachte Darstellung eines RNN Neurons (Quelle: [2])

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

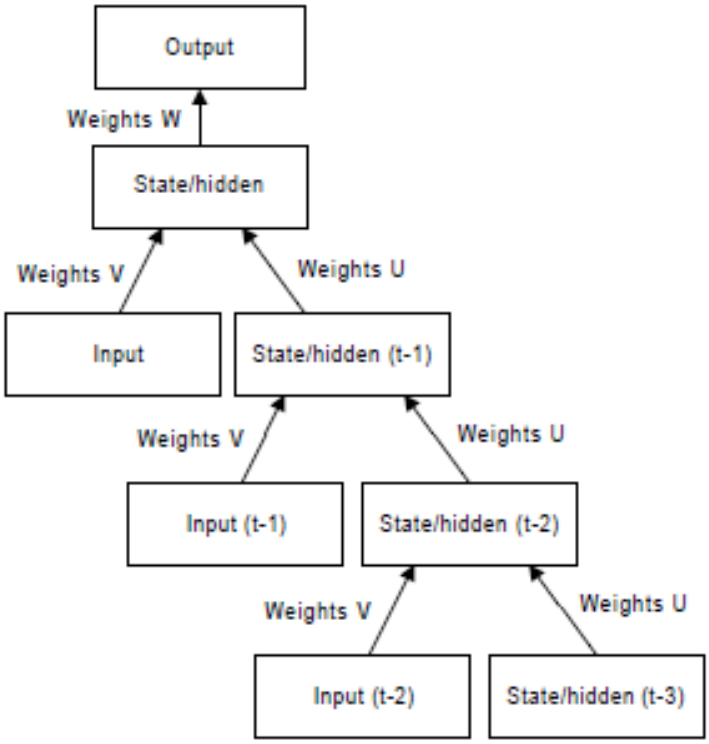

Abb. 2.4.: entrolltes RNN für BPTT (Quelle: [1])

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

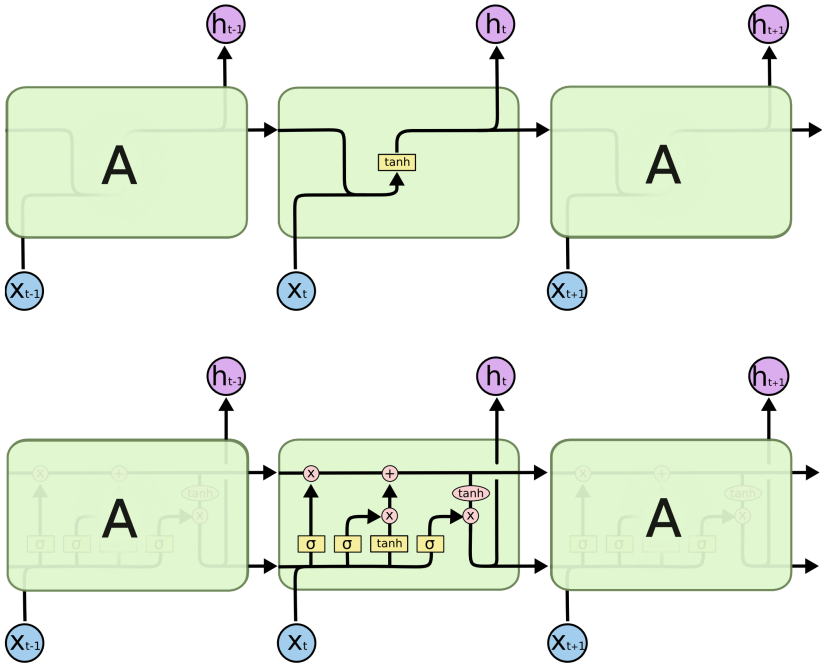

Abb. 2.5.: Vergleich RNN und LSTM (Quelle: [2])

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.

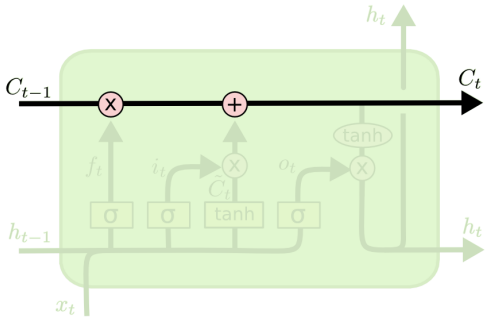

Abb. 2.6.: LSTM Zellzustand (Quelle: [2])

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.

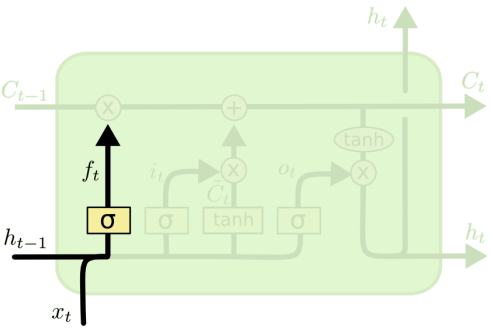

Abb. 2.7.: Forget Gate (Quelle: [2])

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.

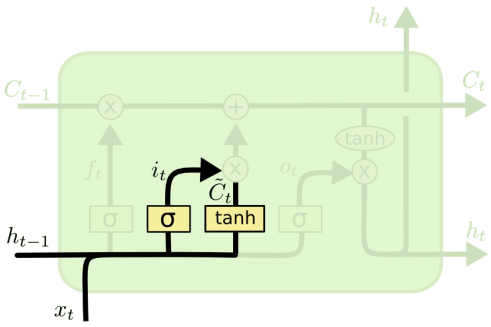

Abb. 2.8.: Eingangsgate (Quelle: [2])

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.

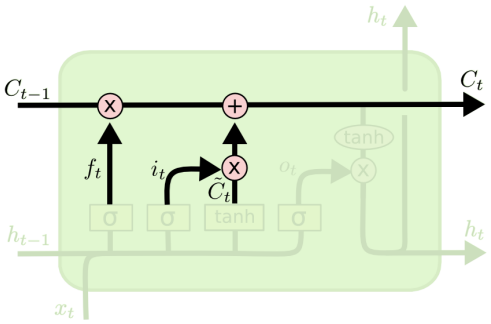

Abb. 2.9.: Zellzustands Update (Quelle: [2])

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.

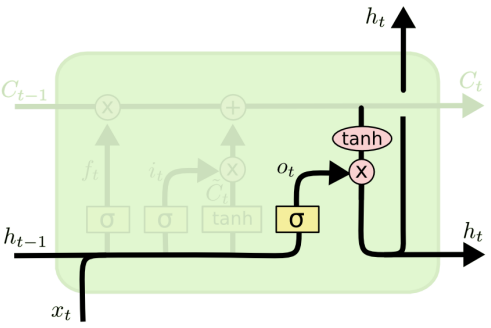

Abb. 2.10.: Ausgabe (Quelle: [2])

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 Gucklоch-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.

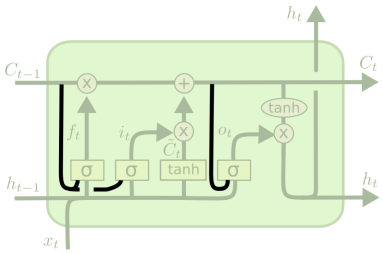

Abb. 2.11.: Variante Gucklöcher (Quelle: [2])

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.

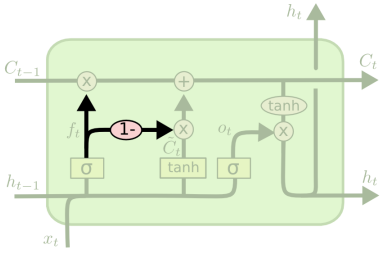

Abb. 2.12.: Variante Zusammengeführte Gates (Quelle: [2])

2.4. Aktueller Forschungsstand (unfinished)

2.4.1. noch ohne Titel

Analyse von LSTM-Netz-Varianten (LSTM: A Search Space Odyssey)

- In recent years, these networks have become the state-of-the-art models for a variety of machine learning problems. - In this paper, we present the first large-scale analysis of eight LSTM variants on three representative tasks: speech recognition, handwriting recognition, and polyphonic music modeling. - In total, we summarize the results of 5400 experimental runs (15 years of CPU time), which makes our study the largest of its kind on LSTM networks. - Our results show that none of the variants can improve upon the standard LSTM architecture significantly, and demonstrate the forget gate and the output activation function to be its most critical components. - The focus of our study is to compare different LSTM variants, and not to achieve state-of-the-art results. Therefore, our experiments are designed to keep the setup simple and the comparisons fair. The vanilla LSTM is used as a baseline and evaluated together with eight of its variants. Each variant adds, removes or modifies the baseline in exactly one aspect, which allows to isolate their effect. Three different datasets from different domains are used to account for cross-domain variations. Quelle: 1503.04069v1.pdf

Bedienung dynamischer Systeme (Control of Dynamic Systems Using LSTM supported Neural Network)

- operation of dynamic system is challenged by non-linearity, disturbances and multivariate interactions - well-suited for to handle this is MPC - combination of LSTM and NN(Neural Network) to learn complex policies of MPC(Model Predictive Control) - MPC is a multivariate control algorithm that uses an internal dynamic model of the process, history of past control moves to yield optimal control actions. - In MPC, the control actions are computed by solving an optimization objective that minimizes a cost function (function of difference in target output and system output) while accounting for system dynamics (using a prediction model) and satisfying output and control action constraints. - However, solving the optimization objective in real time is computationally demanding and often takes lot of time for complex systems. Moreover, MPC requires the estimation of hidden system states (hidden states characterize system dynamics) which can be challenging in complex non-linear dynamic system. - We propose LSTM supported NN model (LSTMSNN). The output of LSTMSNN is a weighted combination of outputs from LSTM and NN. We use this combination because the current control action depends on past control actions, current system output and target output.

The LSTM part of LSTMSNN takes past control actions as input. Because there is temporal dependency between control actions, and we want to use LSTM to capture it. The NN part of LSTMSNN takes current system output and target output as input. Because we want to train NN to make a decision on control action by using current system and target output. - Our trained LSTMSNN Model is computationally less expensive than MPC. - Moreover, our approach does not involve the burden of estimating the hidden states that characterizes system dynamics. Quelle: 2_lstmsnn.pdf

LSTM-Netzwerk-basierte Merkmalextraktion (LSTM Networks for Mobile Human Activity Recognition)

- In this paper, we propose a LSTM-based feature extraction approach to recognize human activities using tri-axial accelerometers data. - The predominant approach to HAR is based on a sliding window procedure, where a fixed length analysis window is shifted along the signal sequence for frame extraction. Preprocessing then transforms raw signal data into feature vectors, which are subjected to statistical classifiers that eventually provide activity hypotheses. - Activity recognition has a wide range of applications in mobile applications â€" from fitness and health tracking to context-based advertising and employee monitoring. - The features used in most of researches on HAR are selected by hand. Designing hand-crafted features in a specific application requires domain knowledge [18], and maybe result in loss of information after extracting features. Quelle: 014.pdf

Semantically Conditioned LSTM-based Natural Language Generation for Spoken Dialogue Systems

- This paper presents a statistical language generator based on a semantically controlled Long Short-term Memory (LSTM) structure. The LSTM generator can learn from unaligned data by jointly optimising sentence planning and surface realisation using a simple cross entropy training criterion, and language variation can be easily achieved by sampling from output candidates. With fewer heuristics, an objective evaluation in two differing test domains showed the proposed method improved performance compared to previous methods. Human judges scored the LSTM system higher on informativeness and naturalness and overall preferred it to the other systems. - The most common and widely adopted today is the rule-based (or template-based) approach (Cheyer and Guzzoni, 2007; Mirkovic and Cavedon, 2011). Despite its robustness and adequacy, the frequent repetition of identical, rather stilted, output forms make talking to a rule-based generator rather tedious. Quelle: 1508.01745v2.pdf

2.4.2. Musikerzeugung

Modelling High-Dimensional Sequences with LSTM-RTRBM: Application to Polyphonic Music Generation

- We propose an automatic music generation demo based on artificial neural networks, which integrates the ability of Long Short-Term Memory (LSTM) in memorizing and retrieving useful history information, together with the advantage of Restricted Boltzmann Machine (RBM) in high dimensional data modelling. Our model can generalize to different musical styles and generate polyphonic music better than previous models. - In this context, we wish to combine the ability of RBM to represent a complicated distribution for each time step, together with a temporal model in sequence. We consider both long-term memory and short-term memory in our design of guide and learning modules, by increasing a bypassing channel from data source filtered by a recurrent LSTM layer and we show that our model increases performance generally. Quelle: 582.pdf

Polyphonic Music Modelling with LSTM-RTRBM

- Our model integrates the ability of Long Short-Term Memory (LSTM) in memorizing and retrieving useful history information, together with the advantage of Restricted Boltzmann Machine (RBM) in high dimensional data modelling. Our approach greatly improves the performance of polyphonic music sequence modelling, achieving the state-of-the-art results on multiple datasets. - For example, in order to complete a melody line, the beginning of the music sequence needs to be held in mind while the rest is played, a task which is carried out by the short-term memory. And the long-term memory will serve as the theme and emotion that will help maintain the global coherence of music. The existence of both the short-term and the longterm memory is vital for generating melodic and coherent music sequences. Quelle: p991-lyu.pdf

https://colah.github.io/posts/2015-08-Understanding-LSTMs/ LSTMs were a big step in what we can accomplish with RNNs. Itâ€™s natural to wonder: is there another big step? A common opinion among researchers is: ”Yes! There is a next step and itâ€™s attention!”The idea is to let every step of an RNN pick information to look at from some larger collection of information. For example, if you are using an RNN to create a caption describing an image, it might pick a part of the image to look at for every word it outputs. In fact, Xu, et al. (2015) do exactly this â€“ it might be a fun starting point if you want to explore attention! Thereâ€™s been a number of really exciting results using attention, and it seems like a lot more are around the cornerâ€¦ Attention isnâ€™t the only exciting thread in RNN research. For example, Grid LSTMs by

Kalchbrenner, et al. (2015) seem extremely promising. Work using RNNs in generative models â€“ such as Gregor, et al. (2015), Chung, et al. (2015), or Bayer & Osendorfer (2015) â€“ also seems very interesting. The last few years have been an exciting time for recurrent neural networks, and the coming ones promise to only be more so!

http://deeplearning4j.org/lstm.html In the mid-90s, a variation of recurrent net with so-called Long Short-Term Memory units, or LSTMs, was proposed by the German researchers Sepp Hochreiter and Juergen Schmidhuber as a solution to the vanishing gradient problem.

3. Deeplearning4j (unfinished)

See also [DL4J]
von http://deeplearning4j.org/quickstart
DL4J targets professional Java developers who are familiar with production deployments, IDEs and automated build tools.

How to: http://deeplearning4j.org/documentation ...

3.1. Getting started (unfinished)

von http://deeplearning4j.org/quickstart

3.1.1. Vorraussetzungen und Empfehlungen (unfinished)

- Java 1.7 oder hÃ¶her (nur 64-Bit Version wird unterstÃ¼tzt
- Apache Maven (Maven is a dependency management and automated build tool for Java projects.) check https://books.sonatype.com/mvnex-book/reference/public-book.html for how to use
- IntelliJ IDEA oder Eclipse (An Integrated Development Environment (IDE) allows you to work with our API and configure neural networks in a few steps. We strongly recommend using IntelliJ, which communicates with Maven to handle dependencies.)
- Git

Installation: http://deeplearning4j.org/gettingstarted Follow the ND4J Getting Started instructions to start a new project and include necessary POM dependencies. http://nd4j.org/getstarted.html http://nd4j.org/dependencies.htm

- neues Projekt in IntelliJ im Anhang A

3.2. Feedforward Netze (unfinished)

http://deeplearning4j.org/quickstart.html Everything starts with a MultiLayerConfiguration, which organizes those layers and their hyperparameters. Hyperparameters are variables that determine how a neural network learns. They include how many times to update the weights of the model, how to initialize those weights, which activation function to attach to the nodes, which optimization algorithm to use, and how fast the model should learn. This is what one configuration would look like:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .iterations(1)
    .weightInit(WeightInit.XAVIER)
    .activation("relu")
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .learningRate(0.05)
    // ... other hyperparameters
    .backprop(true)
    .build();
```

With Deeplearning4j, you add a layer by calling layer on the NeuralNetConfiguration.Builder(), specifying its place in the order of layers (the zero-indexed layer below is the input layer), the number of input and output nodes, nIn and nOut, as well as the type: DenseLayer.

```
    .layer(0, new DenseLayer.Builder().nIn(784).nOut(250)
            .build())
```

Once youâ€™ve configured your net, you train the model with model.fit.

Configuring the POM.xml File

To run DL4J in your own projects, we highly recommend using Maven for Java users, or a tool such as SBT for Scala. The basic set of dependencies and their versions are shown below. This includes:

- deeplearning4j-core, which contains the neural network implementations - nd4j-native, the CPU version of the ND4J library that powers DL4J - canova-api - Canova is our library vectorizing and loading data

http://deeplearning4j.org/gettingstarted.html Reproducible Results

Neural net weights are initialized randomly, which means the model begins learning from a different position in the weight space each time, which may lead it to different local optima.

Users seeking reproducible results will need to use the same random weights, which they must initialize before the model is created. They can reinitialize with the same random weight with this line:

```
Nd4j.getRandom().setSeed(123);
```

3.3. Recurrent Neuronal Network (unfinished)

3.3.1. Builder

http://deeplearning4j.org/doc/ Parametereinstellungen: - iterations(int): Number of optimization iterations. - learningRate(double): Learning rate. Defaults to 1e-1 - optimizationAlgo(OptimizationAlgorithm): - seed(long): Random number generator seed. Used for reproducability between runs - biasInit(double): ?? - miniBatch(boolean): Process input as minibatch vs full dataset. Default set to true. - updater(Updater): Gradient updater. - weightInit(WeightInit): Weight initialization scheme.

check http://deeplearning4j.org/glossary.html for explanations

- OptimizationAlgorithm: verfügbare Algorithmen - Updater: For example, Updater.SGD for standard stochastic gradient descent, Updater.NESTEROV for Nesterov momentum, Updater.RSMPROP for RMSProp, etc. (alle verfügbaren: ADADELTA, ADAGRAD, ADAM, CUSTOM, NESTEROVS, NONE, RMSPROP, SGD) - WeightInit: DISTRIBUTION Distribution: Sample weights from a distribution based on shape of input NORMALIZED Normalized: Normalize sample weights RELU Delving Deep into Rectifiers SIZE Size: Sample weights from bound uniform distribution using shape for min and max UNIFORM Uniform: Sample weights from bound uniform distribution (specify min and max) VI VI: Sample weights from variance normalized initialization (Glorot) XAVIER N(0,2/nIn): He et al. (2015) ZERO Zeros: Generate weights as zeros

```
NeuralNetConfiguration.Builder builder = new NeuralNetConfiguration.
Builder();
builder.iterations(10);
builder.learningRate(0.001);
builder.optimizationAlgo(
OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT);
builder.seed(123);
builder.biasInit(0);
builder.miniBatch(false);
```

```
9     builder.updater(Updater.RMSPROP);
10    builder.weightInit(WeightInit.XAVIER);
```

Listing 3.1: Beispiel zur RNN-Erstellung: Builder

3.3.2. ListBuilder

first difference, for rnns we need to use GravesLSTM.Builder we need to use RnnOutputLayer for our RNN

// softmax normalizes the output neurons, the sum of all outputs is 1 // this is required for our sampleFromDistribution-function

```
1     ListBuilder listBuilder = builder.list(HIDDEN_LAYER_CONT + 1);
2
3     for (int i = 0; i < HIDDEN_LAYER_CONT; i++) {
4         GravesLSTM.Builder hiddenLayerBuilder = new GravesLSTM.Builder();
5         hiddenLayerBuilder.nIn(i == 0 ? LEARNSTRING_CHARS.size() :
      HIDDEN_LAYER_WIDTH);
6         hiddenLayerBuilder.nOut(HIDDEN_LAYER_WIDTH);
7         hiddenLayerBuilder.activation("tanh");
8         listBuilder.layer(i, hiddenLayerBuilder.build());
9     }
10
11    RnnOutputLayer.Builder outputLayerBuilder = new RnnOutputLayer.Builder(
      LossFunction.MCXENT);
12    outputLayerBuilder.activation("softmax");
13    outputLayerBuilder.nIn(HIDDEN_LAYER_WIDTH);
14    outputLayerBuilder.nOut(LEARNSTRING_CHARS.size());
15    listBuilder.layer(HIDDEN_LAYER_CONT, outputLayerBuilder.build());
16
17    // finish builder
18    listBuilder.pretrain(false);
19    listBuilder.backprop(true);
20    listBuilder.build();
```

Listing 3.2: Beispiel zur RNN-Erstellung: ListBuilder

3.3.3. Netz erzeugen

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

```
4     net.init();
5     net.setListeners(new ScoreIterationListener(1));
```

Listing 3.3: Beispiel zur RNN-Erstellung: Netz erzeugen

3.3.4. Daten erstellen

// create input and output arrays: SAMPLE_INDEX, INPUT_NEURON, // SEQUENCE_POSITION

```
1     INDArray input = Nd4j.zeros(1, LEARNSTRING_CHARS_LIST.size(),
      LEARNSTRING.length);
2     INDArray labels = Nd4j.zeros(1, LEARNSTRING_CHARS_LIST.size(),
      LEARNSTRING.length);
3     // loop through our sample-sentence
4     int samplePos = 0;
5     for (char currentChar : LEARNSTRING) {
6         // small hack: when currentChar is the last, take the first char as
7         // nextChar - not really required
8         char nextChar = LEARNSTRING[(samplePos + 1) % (LEARNSTRING.length)];
9         // input neuron for current-char is 1 at "samplePos"
10        input.putScalar(new int[] { 0, LEARNSTRING_CHARS_LIST.indexOf(
      currentChar), samplePos }, 1);
11        // output neuron for next-char is 1 at "samplePos"
12        labels.putScalar(new int[] { 0, LEARNSTRING_CHARS_LIST.indexOf(
      nextChar), samplePos }, 1);
13        samplePos++;
14    }
15    DataSet trainingData = new DataSet(input, labels);
```

Listing 3.4: Beispiel zur RNN-Erstellung: Daten erstellen

3.3.5. Netz trainieren

```
1     for (int epoch = 0; epoch < 100; epoch++) {
2         System.out.println("Epoch " + epoch);
3
4         // train the data
5         net.fit(trainingData);
6
7         // clear current stance from the last example
8         net.rnnClearPreviousState();
9
10        // put the first caracter into the rrn as an initialisation
```

```
11          INDArray testInit = Nd4j.zeros(LEARNSTRING_CHARS_LIST.size());
12          testInit.putScalar(LEARNSTRING_CHARS_LIST.indexOf(LEARNSTRING[0]),
    1);
13
14          // run one step -> IMPORTANT: rnnTimeStep() must be called, not
15          // output()
16          // the output shows what the net thinks what should come next
17          INDArray output = net.rnnTimeStep(testInit);
18
19          // now the net should guess LEARNSTRING.length mor characters
20          for (int j = 0; j < LEARNSTRING.length; j++) {
21              // first process the last output of the network to a concrete
22              // neuron, the neuron with the highest output cas the highest
23              // cance to get chosen
24              double[] outputProbDistribution = new double[LEARNSTRING_CHARS.
    size()];
25              for (int k = 0; k < outputProbDistribution.length; k++) {
26                  outputProbDistribution[k] = output.getDouble(k);
27              }
28              int sampledCharacterIdx = findIndexOfHighestValue(
    outputProbDistribution);
29
30              // print the chosen output
31              System.out.print(LEARNSTRING_CHARS_LIST.get(sampledCharacterIdx)
    );
32
33              // use the last output as input
34              INDArray nextInput = Nd4j.zeros(LEARNSTRING_CHARS_LIST.size());
35              nextInput.putScalar(sampledCharacterIdx, 1);
36              output = net.rnnTimeStep(nextInput);
37          }
38          System.out.print("\n");
39      }
```

Listing 3.5: Beispiel zur RNN-Erstellung:

3.4. LSTM Netze (unfinished)

A commented example of a Graves LSTM learning how to replicate Shakespearian drama, and implemented with Deeplearning4j

Hyperparameter Tuning

http://deeplearning4j.org/lstm.html Here are a few ideas to keep in mind when manually optimizing hyperparameters for RNNs: - Watch out for overfitting, which happens when a neural network essentially ”memorizes”the training data. Overfitting means you get great performance on training data, but the networkâ€™s model is useless for out-of-sample prediction. - Regularization helps: regularization methods include l1, l2, and dropout among others. -So have a separate test set on which the network doesnâ€™t train. - The larger the network, the more powerful, but itâ€™s also easier to overfit. Donâ€™t want to try to learn a million parameters from 10,000 examples â€“ parameters ¿ examples = trouble. - More data is almost always better, because it helps fight overfitting. - Train over multiple epochs (complete passes through the dataset). - Evaluate test set performance at each epoch to know when to stop (early stopping). - The learning rate is the single most important hyperparameter. Tune this using deeplearning4j-ui; see [this graph] - In general, stacking layers can help. - For LSTMs, use the softsign (not softmax) activation function over tanh (itâ€™s faster and less prone to saturation (0 gradients)). - Updaters: RMSProp, AdaGrad or momentum (Nesterovs) are usually good choices. AdaGrad also decays the learning rate, which can help sometimes. - Finally, remember data normalization, MSE loss function + identity activation function for regression, Xavier weight initialization

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

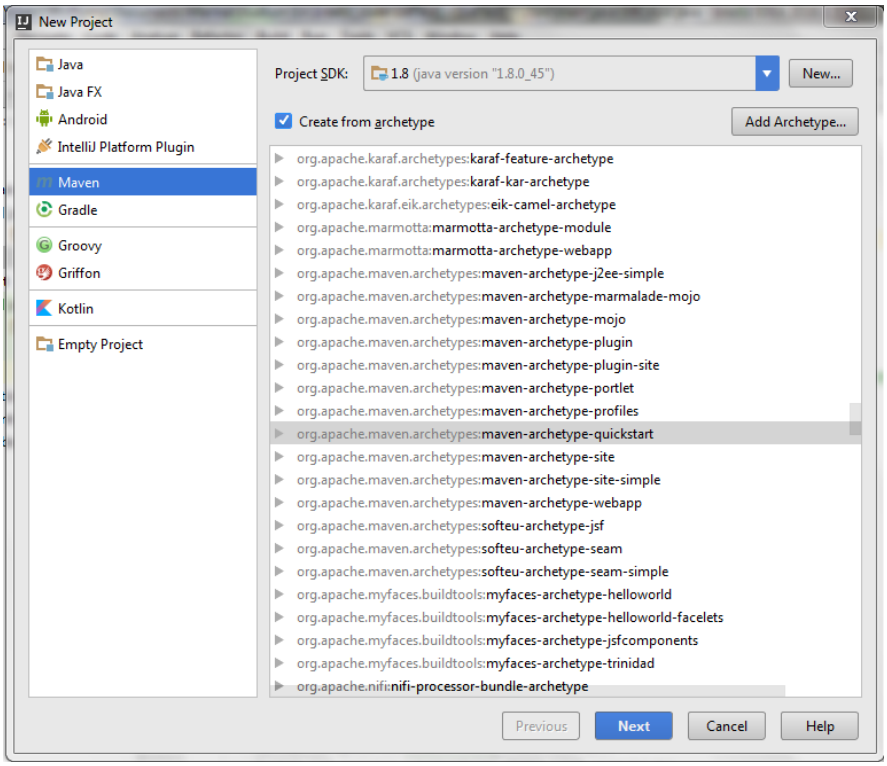

Abb. A.1.: „New Project“ Fenster

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 (Package Name) sowie ArtifactId (?) festlegen. Anschließend noch einen Projektname festlegen und „Finish“ drücken.

http://nd4j.org/getstarted.html

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.

The default backend for CPUs is nd4j-native. You can paste that into the <dependencies> ... </dependencies> section of your POM like this:

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

Listing A.1: applicationContext.xml

ND4Jâ€™s version is a variable here. It will refer to another line higher in the POM, in the <properties> ... </properties> section, specifying the nd4j version and appearing similar to this:

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

Listing A.2: applicationContext.xml

The DL4J dependencies you add to the POM will vary with the nature of your project.

In addition to the core dependency, given below, you may also want to install deeplearning4j-cli for the command-line interface, deeplearning4j-scaleout for running parallel on Hadoop or Spark, and others as needed.

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

Listing A.3: applicationContext.xml

Weitere Informationen bezüglich GPU Betrieb oder Benutzung andere Betriebssysteme 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).

[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).

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