Ludwig-Maximilians-Universität München
Faculty for Mathematics, Computer Science and Statistics

Masterarbeit

Automatische Textgenerierung für finanzielle Berichte

Autor:

Henryk Borzymowski

Betreuung:

Prof. Dr. Christian Heumann

20. Januar 2019

Abstract

Automatische Textgenerierung für finanzielle Berichte

Die vorliegende Masterarbeit gibt einen Einblick in die Theorie und Anwendung von verschiedenen Methoden der Textgenerierung. Da viele Bereiche der Wirtschaft mit textbasierten Problemen zusammenhängen, wurde in dieser Arbeit, zur Erlernung der finanziellen Sprache, eine Datenquelle der Jahresabschlüsse von über 5000 Unternehmen herangezogen. Mit Hilfe von verschiedenen Algorithmen, die auf *Neuronalen Netzen* basieren, wurde versucht, Text, anhand eines vorgegebenen Themas, automatisch zu generieren. Diese Automatisierung von wiederkehrenden und wiederholbaren Aufgaben könnte in vielen Situationen zur Einsparung von Zeit sowie zur Vermeidung von menschlichen Fehlern führen. Die Masterarbeit ist vor allem an Personen gerichtet, die ein starkes Vorwissen an statistischen und mathematischen Methoden haben. Die Theorie und Praxisanwendung von *Deep Learning* ist keine Voraussetzung, da die grundlegenden Konzepte in der Arbeit weitreichend beschrieben werden.

Inhaltsverzeichnis

1 Einleitung

In allen Bereichen des Lebens trifft man auf geschriebenen Text. Um nur ein paar Beispiele zu nennen: Bücher, die man liest (in Papierformat wie auch Digitalformat), Briefe, die man schreibt, sowie E-Mails, Reporte, Berichte und noch eine Menge mehr. Viele dieser Tätigkeiten sind sehr zeitaufwendig, weil man sich erstens überlegen muss, wovon der zu verfassende Text handeln soll und zweitens macht man sich Gedanken darüber, in welcher Form und in welchem Stil der Text geschrieben werden soll. Beispielsweise bei E-Mails an Bekannte oder beim Schreiben eines Buches will man sich Zeit nehmen und genau überlegen, um einen einmaligen und persönlichen Text zu erhalten. Bei anderen Aufgaben, wie zum Beispiel beim *Reporting* oder beim Schreiben eines Berichts, die oft eine vergleichbare Form annehmen und dazu dienen, gewisse Prozesse zu dokumentieren, möchte man nicht zu viel Zeit investieren, da diese dem eigentlich zu behandelnden Problem gewidmet werden könnte. Oft sieht der Prozess des geschriebenen Textes ähnlich aus, ist aber jedes Mal auf gewisse Zusatzinformationen zurechtgeschnitten, weshalb immer erneut eine gewisse Zeit in Anspruch genommen werden muss.
In dieser Arbeit wird deshalb der Schwerpunkt auf finanzielle Berichte gelegt, da sie oft mandatorisch sind und sehr detailliert sein müssen. Auf die genaue Struktur und den Aufbau dieser Berichte wurde in den folgenden Kapiteln noch näher eingegangen.

1.1 Problemstellung

Hauptziel der Arbeit wird es sein, einen Algorithmus in Richtung einer vollautomatisierten Textgenerierung zu konzipieren.

Der Prozess der Textgenerierung ist wie folgt aufgebaut: es ist notwendig, konstante, also sich nicht verändernde, Sätze zu generieren. Diese Sequenzen dienen im nächsten Schritt als Anker und Anfangssequenzen für den restlichen Text, der im zweiten und letzten Schritt generiert wird. In der Regel folgen Texte der gleichen Art ähnlichen Schemata. Dies erlaubt es, eine einheitliche Struktur zu bestimmen, die wiederum in einem automatisierten Prozess berücksichtigt werden kann. Die Aufgabe besteht darin, dem Computer mit Hilfe eines Algorithmus die *Sprache eines gewissen Schreibstils*, basierend auf bereits vorliegenden Texten, beizubringen. Diese *erlernte* Sprache würde im Idealfall mit den, in den vorherigen Schritten generierten Anfangssequenzen dazu führen, dass man sowohl konstante und strikte Sätze, die rein quantitative und informative Zwecke haben, als auch Sätze, die semantisch ausgerichtet sind, kombinieren könnte. Zusammenfassend sind zwei Schritte notwendig, um einen Bericht vollständig automatisieren zu können:

1. Bauen von konstanten Sätzen, die als Anfangssequenzen dienen.
2. Lernen von einem bestimmten Schreibstil, um Semantik in den Text zu integrieren.

Diese Masterarbeit wird sich hauptsächlich mit dem zweiten Punkt beschäftigen, da er den Unterschied zu einem rein technisch generierten Text ausmachen kann. Darüber hinaus ist für

viele Gebiete das Erstellen von text-basierten Aufgaben ein zeitaufwendiger Prozess, weshalb die angewandten Methoden und das Wissen, die in dieser Arbeit generiert werden, von bedeutendem Interesse sein können.

Die Problemstellung der Masterarbeit beschränkt sich folglich darauf, einen Algorithmus zu entwickeln, der anhand einer Anfangssequenz selbstständig in der Lage ist, eine darauf folgende Sequenz von Charakteren in eine logische Reihenfolge zu bringen und dadurch verständliche Sätze zu bilden. Dies wird mit Hilfe von *Recurrent Neural Networks (RNN)* und genauer *Long Short-Term Memory (LSTM)* Methoden erreicht. Diese werden unter anderem auch in der Musik- und Spracherkennung verwendet und eignen sich besonders gut für Aufgaben, bei denen zeitlich codierte Informationen erkannt werden müssen.

Eine besondere Schwierigkeit dieser Aufgabe besteht darin, dass Texte, im Gegensatz zur Musik und Bildern, die eher eine künstlerische Darstellung sind, sehr strikten grammatikalischen Regeln unterliegen. Das bedeutet, dass ganz allgemein verstandene Fehler, bezugnehmend auf Musik, als ein besonderes Attribut angesehen werden können, wobei ein Fehler im Text dazu führen kann, dass der Sinn des Satzes oder des ganzen Textes missverstanden werden kann. Dies kann weitaus größere Konsequenzen zur Folge haben.

Die Aufgabe wird auf Zeichen-Basis analysiert, was dazu führt, dass einerseits erreicht werden kann, dass Wörter nicht einzeln aus dem Kontext gegriffen werden, weil sie als Sequenz von Zeichen aufgenommen werden. Allerdings ist der Lernprozess einer solchen Darstellung viel aufwendiger, da man die ganze Sprache vom Aufbau von Wörtern über die grammatischen Zusammenhänge lernen muss. Die Rechenzeit auf Zeichen-Basis ist zudem viel aufwendiger als beispielsweise bei Analysen auf Wörter-Basis.

Zu berücksichtigen:

1. Manche Arten von Texten, wie z.B. Berichte, können einen relativ komplizierten und strikten Aufbau bezüglich der Sätze haben. Demnach kann das Erlernen einer solchen Sprache technisch und lexikalisch sehr anspruchsvoll sein.

2. *RNN's* benötigen einen sehr großen Datensatz. Für manche Aufgaben könnten die vorhandenen Texte nicht ausreichend sein. In diesem Fall könnten Texte aus einem anderen Gebiet hinzugezogen werden, was einerseits der Bedingung des großen Datensatzes entgegenkommt, aber andererseits dazu führt, dass sich die Sprache ändern kann und nicht mehr genau der Sprache, der in erster Linie erlernten Richtung, folgt.

1.2 Textgenerierung

Das Problem der Textgenerierung kann unterschiedlich angegangen werden - wie viel Text und in welcher Form der Text generiert werden soll, hängt stark von dem behandelten Problem ab. Die einfachste Möglichkeit, Text zu einem bestimmten Problem zu generieren, wäre es, einen konstanten und universellen Text zu schreiben, der allerdings keine spezifischen Merkmale des individuellen Beispiels enthalten könnte. Eine zweite Möglichkeit bestünde darin, einen konstanten Text zu schreiben, in welchen an den relevanten Stellen beispielsweise spezifische Informationen (Text oder Zahlen) eingefügt werden. Diese Methode ist relativ einfach und ist aus diesem Grund auch weit verbreitet. Der große Nachteil dieser Methode ist, dass die Informationen, die eingebaut werden, in Kombination mit dem restlichen Text dazu führen können, dass der Sinn des Textes verloren geht. Dies würde nur gut bei solchen Texten funktionieren, in denen Informationen weitergegeben werden, die keiner Beurteilung oder Meinung unterliegen. Es wird also deutlich, dass man in Bezug auf manche Probleme nicht unbedingt von Textgenerierung sprechen kann, da der Text oft nur einmal generiert und unterschiedlich angepasst wird.

Ein weiterer Faktor, der für die textgenerierende Methode von Bedeutung ist, ist in welcher Form das zu beschreibende Problem vorliegt. Beispiele von *Inputs* und *Outputs*, wie auch von den verwendeten Methoden, findet man in der folgenden Tabelle wieder:

Tabelle 1: Beispiele Textgenerierung

Input	Output	Methode	Beschreibung
Bild	Beschreibung des Bildes	CNN + RNN	On the Automatic Generation of Medical Imaging Reports (Baoyu Jing 2018)
Spielergebnisse	Spielberichte	RNN	Spielberichte für die Begegnungen des aktuellen Bundesligaspieltags (retrescro 2018)
Stichwort	Nachrichten-artikel	RNN	I taught a computer to write like Engadget (Souppouris 2015)
IFRS Zahlen	Finanzbericht	Lücken füllen	Automatisierung von Finanzberichten

Quelle: Eigene Darstellung

Wie man in **Tabelle 1** sehen kann, wird immer öfter, in verschiedenen Bereichen, eine *Deep Learning* Methode angewandt. Hierbei werden Methoden wie *Convolutional Neural Networks (CNN)* und *Recurrent Neural Networks (RNN)*, die im späteren Verlauf der Arbeit erklärt werden, verwendet (Karpathy 2015). Bei manchen Anwendungsfeldern werden jedoch nach wie vor Methoden verwendet, die auf konstanten Texten basieren. Zum Beispiel werden finanzielle Berichte, basierend auf IFRS (*International Financial Reporting Standards*) Kennzahlen, von manchen Unternehmen dazu verwendet, bestehende Lücken eines konstanten Texts zu füllen. Hier ergibt sich demnach ein Anwendungsfeld, dass deutlich verbessert werden kann. Im ersten Beispiel der **Tabelle 1** geht es darum, gewisse Merkmale auf medizinischen Bildern zu erkennen

und anschließend zu beschreiben. Für den ersten Schritt wurde ein *CNN* benutzt, um darauf folgend die extrahierten *Features* mit Hilfe eines *RNNs* zu beschreiben. Im zweiten und dritten Beispiel wurde Text anhand eines *RNN* generiert, wobei beim ersten der beiden nur das Ergebnis eines Spiels übergeben wurde, um den Spielbericht zu verfassen. Beim zweiten Beispiel wird ein Thema vorgegeben, zu dem im Nachhinein ein Nachrichtenartikel geschrieben wird.

2 Theorie

In diesem Kapitel wurde genauer auf die Theorie der angewandten Methoden eingegangen. Da *Deep Learning* Methoden relativ komplex sind und diese die Summe vieler kleinerer analytischer Schritte sind, ist dieses Kapitel in einer modularen Struktur aufgebaut. Die Wahl des *Encodings*, Optimierung Algorithmus oder auch die Architektur des Neuronalen Netzes ist arbiträr - dies ermöglicht dem Leser, die für ihn relevanten Methoden und Algorithmen frei zu wählen. Darüber hinaus ist dieser Bereich der Statistik ein sich schnell veränderndes Gebiet, in dem immer häufiger neue Lösungen für unterschiedliche Probleme vorgestellt werden.

Im **Unterkapitel 2.1** wurden verschiedene Möglichkeiten des *Encodings* der Daten, wie auch die Vor- und Nachteile in Hinsicht auf die darauf folgenden Methoden, beschrieben. Das **Unterkapitel 2.2** stellt das *Gradient Descent* und dessen Erweiterungen dar. Im Folgenden wurde die Idee von *Recurrent Neural Networks*, mit Schwerpunkt auf *Gated Recurrent Units* und *Long Short-Term Memory Cells*, vorgestellt. Da *RNNs* auf Neuronalen Netzen basieren, wurde im ersten Schritt eine Einführung in die jeweiligen Elemente eines *NN* aufgezeigt.

2.1 Encoding

Das *Encoding* der Daten entscheidet darüber, wie der Algorithmus funktionieren wird, sowohl aus Sicht der Inanspruchnahme von Rechenkraft, wie auch aus Sicht der Ergebnisse, die man erhält. Erstens werden unterschiedliche Datentypen und Formate in Bezug auf deren *Tensor* Aufbau, die die Grundlage jeder *Deep Learning* Methode darstellt, besprochen. Zweitens wird die Ebene, auf der das *Encoding* stattfindet, definiert. Hierfür wird zwischen einem Zeichen-basierten und einem Wörter-basierten *Encoding* entschieden. Im dritten und letzten Schritt werden die zwei häufigst verwendeten Methoden des *Encodings* dargestellt und deren praktische Anwendung anhand eines Beispiels erklärt.

2.1.1 Inputspace (Tensoren)

Der grundlegende Baustein eines jeden *Machine Learning* Algorithmus ist der *Input*, also die Daten, die man zur Durchführung von Analysen benötigt. Daten können in *Deep Learning* die unterschiedlichsten Formen annehmen, da dieses Gebiet verschiedenste Probleme lösen kann. In **Tabelle 2** werden Beispiele von Daten und deren möglichen Dateiformaten, in denen sie gespeichert werden, dargestellt. Es ist gleichgültig, von welchem Datentyp man redet oder in was für einem Format eine Datei gespeichert ist - man muss sich dennoch überlegen, wie man den *Input* einheitlich transformieren kann. Der *Input* muss eine ganz bestimmte Form haben, damit er in ein in *Keras* (*Deep Learning Framework*) entwickeltes Modell einfließen und verarbeitet werden kann. Diese bestimmte Form nennt man *Tensoren*, woher auch der Name des *Deep Learning* Programms von Google stammt - *TensorFlow* (Abadi et al. 2015). Egal ob man mit Bildern, Videos, Audioaufnahmen oder Text arbeiten möchte - die Form, in der diese Daten eingelesen werden, sind Zahlen, die als *Tensoren* verpackt werden.

Tabelle 2: Zuordnung von Datentypen und deren Formaten

Type	Format	Tensor Dimension
Audio	MP3, WAV, FLAC	4 D Tensor
Video	AVI, MP4, FLV	5 D Tensor
Bild	JPEG, PNG, BMP	4 D Tensor
Text	TXT, RTF, JSON	3 D Tensor

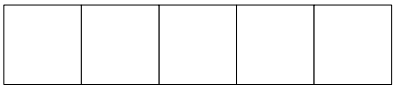
Quelle: Eigene Darstellung

Tensoren sind eine Verallgemeinerung von Vektoren, da sie eine beliebige Dimension annehmen können (Francois Chollet 2018). So ist ein null-dimensionaler *Tensor* eine einzige Zahl - Skalar (**Abbildung 1 a)**). Ein ein-dimensionaler *Tensor* (1 D *Tensor*) wird als die Zusammensetzung von mehreren Skalaren dargestellt, die einen Vektor bilden (**Abbildung 1 b)**). Hierbei gibt es zwei Begriffe, die nicht verwechselt werden sollten: die *Tensor* Dimensionen, die bei einem Vektor, wie bereits erwähnt, 1 beträgt und die Dimension des Vektors an sich. Bei einem Vektor der Länge 5 beispielsweise, redet man von einem fünf-dimensionalen Vektor - dieser wiederum hat aber nichts mit einem fünf-dimensionalen *Tensor* zu tun. Um sich die Dimension eines Tensors besser vorstellen zu können, könnte man auch die Anzahl der Achse, in der die Daten verbreitet sind, als Dimension des *Tensors* hernehmen. Eine Matrix wie auf **Abbildung 1 c)** hat die Dimension 5 auf 5 und ist somit ein 2 D *Tensor*. Wenn man diese 2 D *Tensoren* aufstockt, erhält man, so wie in **Abbildung 1 d)** einen Datenwürfel, der ein 3 D *Tensor* ist. Um entsprechende *Tensoren* der Dimension 4 und 5 zu erhalten, stockt man die jeweils vorhergehende Daten der Struktur ***n*** *D Tensor* auf (**Abbildung 1 e)** und **f)**). In der Praxis sind dies die häufigsten Beispiele von *Tensoren*. Es ist jedoch möglich, einen *Tensor* einer beliebigen Dimension aufzubauen.

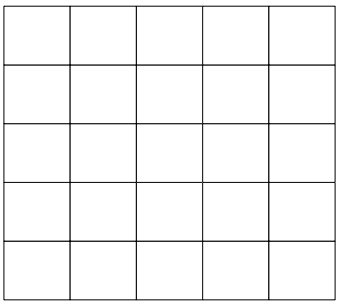
Abbildung 1: Tensor Darstellung

(a) 0 D Tensor

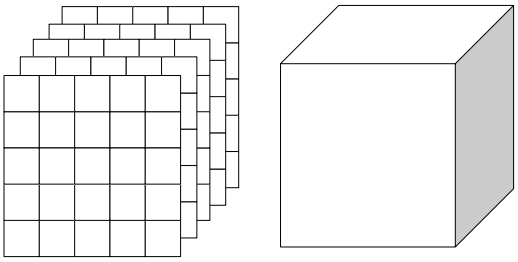
(b) 1 D Tensor



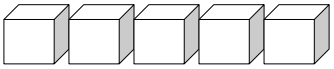
(c) 2 D Tensor



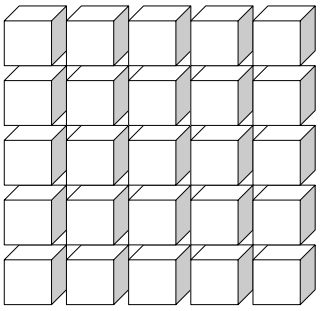
(d) 3 D Tensor



(e) 4 D Tensor



(f) 5 D Tensor



Quelle: Eigene Darstellung

2.1.2 Zeichen- und Wörter-basiertes Encoding

Die Daten, die in dieser Arbeit verwendet werden, sind Textdokumente. Dies wurde genauer in späteren Kapiteln besprochen, jedoch ist diese Information wichtig um festzustellen, mit welcher Art von Tensoren gearbeitet wird. Da die Aufgabe darin besteht neuen Text zu generieren, bestehen zwei Möglichkeiten der Ebenen. Die erste ist ein Zeichen-basiertes *Encoding* was bedeutet, dass das Ziel sein wird, ein auf eine Anfangssequenz folgendes Zeichen vorauszusagen. Die zweite Möglichkeit ist, dass man nicht ein nächstes Zeichen, sonder gleich ein ganzes Wort voraussagen möchte (Wörter-basiert).
Bei der ersten Option ist es ein großer Vorteil, dass die Anzahl der Klassen, die man voraussagen möchte, erheblich begrenzt ist (sie ist gleich der Größe des Vokabulars). Bei der Wörterbasierten Methode hingegen ist die Anzahl der Klassen gleich der Anzahl der möglichen Wörter im Text (dies wird meistens auf die 10 000 häufigsten Wörter begrenzt). Das macht es schwierig, herkömmliche Verlustfunktionen (**Unterkapitel 2.3)** zu benutzen und macht die Aufgabe des Ausarbeitens einer Architektur eines Neuronalen Netzes erheblich schwerer. Um diesem Problem auszuweichen, werden oft *Embeddings* (**Unterkapitel 2.1.4**) benutzt, die die *Output* Dimension verringern sollen. Ein Nachteil der Zeichen-basierten Methode ist erstens, dass Wörter und Sätze gebildet werden können, die keinen Sinn ergeben, oder nicht existieren. Die zweite Methode hingegen wird nur Wörter voraussagen, die auch tatsächlich existieren - oder im schlimmsten Fall ein OOV (*Out-of-vocabulary*) Wort. Des Weiteren sind Wörter-basierte Modelle robuster und produzieren, im Gegensatz zu Zeichen-basierten Modellen, häufiger grammatikalisch korrekte Sätze.

2.1.3 Lable Encoding und One-hot Encoding

Das *Encoding* erlaubt dem Algorithmus die Daten in einer entsprechenden Form einzulesen. Diese Form ist, wie im vorherigem Abschnitt erklärt, ein *Tensor*. In dieser Arbeit wird Text verarbeitet und deswegen muss man die Zeichen nummerisch darstellen. Dazu werden zwei Methoden zum *Encoden* kategorischer Variablen vorgestellt: das *Lable Encoding* und *One-hot Encoding*. *Lable Encoding* ist die einfachste Form des *Encodings*, da jeder Kategorie eine bestimmt Zahl zugewiesen wird. Dies kann man in **Abbildung 2** erkennen - jedes Zeichen in der Sequenz “annual report! wird als eine vorher festgelegte Zahl dargestellt. Das Produkt aus diesem *Encoding* ist ein *1 D Tensor*, der der maximalen Länge der Sequenz entspricht.

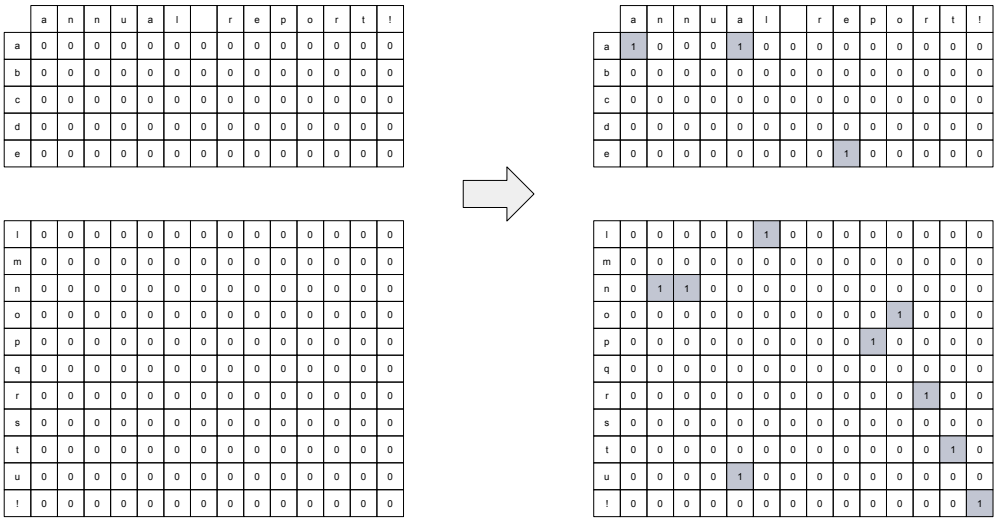
Abbildung 2: Label Encoding von Texten

a	n	n	u	a	l		r	e	p	o	r	t	!
1	14	14	21	1	12	27	18	5	16	15	18	20	30

Quelle: Eigene Darstellung

Das *One-hot Encoding* hingegen ist eine Methode, in der jedes Zeichen in Form eines Vektors dargestellt wird. Diese Vektoren sind mit 0-len gefüllt - bis auf die Stelle, die der Position des jeweiligen *encodetem* Zeichen entspricht. Dies wurde in **Abbildung 3** dargestellt, in der die gleiche Beispiel-Phrase wie im vorherigen Beispiel *encodet* wurde. Diese Methode liefert im Endeffekt eine Matrix, also einen *2 D Tensor* mit Anzahl der Zeilen gleich der Länge des Vokabulars und Anzahl der Spalten gleich der maximalen Länge einer Sequenz.

Abbildung 3: One-hot Encoding von Texten



Quelle: Eigene Darstellung

Allgemein zählen zu den Vorteilen des *One-hot Encodings*, dass jede Kategorie (hier: Zeichen) unabhängig betrachtet werden kann - wobei andererseits das *Lable Encoding* eine Abhängigkeit oder einen Zusammenhang der Kategorien voraussetzt. Das Zweitere kann allerdings bei sogenannten *Embeddings* von Vorteil sein (diese Methode wurde im weiteren Verlauf der Arbeit vorgestellt), bei denen man ähnliche Kategorien in einem Vektorraum möglichst gut gruppieren möchte.

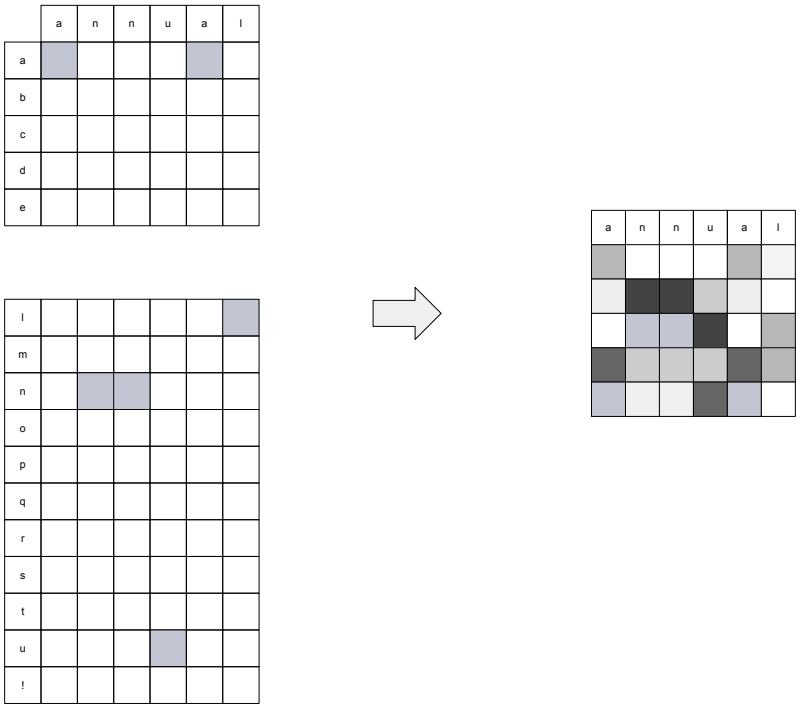
2.1.4 Embedding

Beim *Label Encoding* wurde im späteren Schritt, also beim Einlesen ins Modell die jeweils entsprechende Zahl der Position eines Zeichens im Vokabular auch in einen Vektor transformiert. Hierbei ist der Unterschied zum *One-hot Encoding* der, dass man versucht eine weniger Sparse Matrix zu erhalten - also eine dichtere Repräsentation der Vektoren. Als Beispiel könnte man sich vorstellen, dass man anstatt von Zeichen, Wörter *Encodet* und im Falle eines *One-hot Encoding* eine Matrix mit Anzahl der Zeilen gleich der Anzahl der Wörter im behandelten Problem. Oft kann diese Zahl in die Tausenden gehen und wenn man mit ganzen Dokumenten arbeitet, wird diese Zahl auf 10 oder 20 Tausend der am häufigsten vorkommenden Wörter be-

grenzt. Um nicht auf Matrizen arbeiten zu müssen, die tausende Zeilen (mit Einträgen 0 oder 1) enthält, benutzt man die sogenannten *Embeddings*. *Embeddings* komprimieren die Information eines Datenpunktes (hier Wörter oder Zeichen) an einem gewiesen *Time Step* in einen Vektor beliebiger Dimension. Diese Vektoren bilden ein bestimmtes Wort in einen z.B 100 dimensionalen Raum ab, sogar wenn die Anzahl aller Wörter deutlich größer ist. Dies hat erstens den Vorteil, dass man das Problem der Sparsen Matrix umgehen kann und zweitens kann man Zusammenhänge zwischen Wörtern modellieren. Wenn man diese *Embeddings* anschließend mit einer Dimension-Reduktion behandelt, ist man in der Lage, diese Zusammenhänge zwischen Wörtern oder Zeichen in einem 2 oder 3 dimensionalen Koordinatensystem darzustellen und somit zu visualisieren (Falbel 2017).

In **Abbildung 4** wurde das *One-hot-Encoding* mit dem *Embedding* dargestellt und verglichen. Man sieht, dass das *Embedding* im Vergleich zum *One-hot Encoding* eine deutlich niedrigere Dimension und dichtere Vektoren hat, die darüber hinaus aus den Daten erlernt werden können. Beim *One-hot Encoding* sind die Vektoren *Sparser*. Diese haben eine höhere Dimension und sind hart *gecodet*. *Embeddings* sind somit besonders gut geeignet für Beispiele in denen die Dimension beim *One-hot Encoding* zu hoch ist. Ein Beispiel für ein *Embedding* wurde auch für das Textgenerierungsproblem dieser Arbeit im Kapitel mit den Ergebnissen, dargestellt.

Abbildung 4: One-hot Encoding und Embeddings

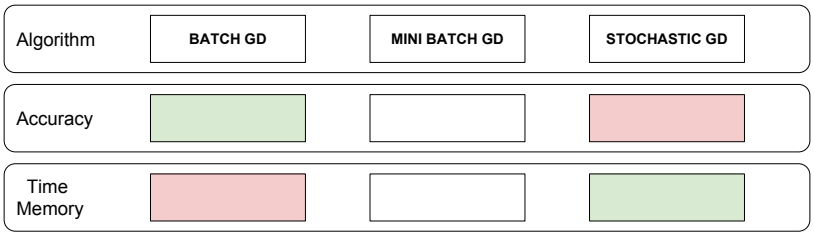


Quelle: Eigene Darstellung (Francois Chollet 2018)

2.2 Optimierungsalgorithmen

Es gibt unterschiedliche Optimierungsalgorithmen, die in der Statistik verwendet werden. Einer davon ist *Gradient Descent*, welcher in der Literatur sehr oft verwendet wird. Bei *Deep Learning*-Ansätzen wird er besonders gern benutzt, da er nicht nur relativ einfach zu verstehen und implementieren ist, sondern auch für hochdimensionale Optimierungsprobleme gut geeignet ist. Es gibt drei *Gradient Descent* Varianten (Tushar 2017):

Abbildung 5: Varianten von Gradient Descent



Quelle: Eigene Darstellung (Tushar 2017)

Batch Gradient Descent nutzt den ganzen Datensatz, um ein *Update* zu machen - dies kann jedoch sehr zeit- und rechenaufwendig sein, besonders wenn der Datensatz zu groß ist, um in die *Memory* eingelesen zu werden. Auf der anderen Seite gibt es *Stochastick Gradient Descent*, welcher es erlaubt, nach jedem eingelesenen Beispiel ein *Update* des Gradienten durchzuführen, was den Vorteil bringt, dass der Algorithmus viel schneller iterieren kann. Der Nachteil hingegen ist, dass das *Update* nur anhand eines Beispiels durchgeführt wird, was äußerst ungenau sein kann. Ein Kompromiss zwischen den beiden Methoden wird durch den *Mini-Batch Gradient Descent* Algorithmus erreicht, der das *Update* anhand einer gewissen Anzahl (*Mini-Batch Size*) von Beispielen durchführt. Diese Methode ermöglicht es, schneller als *Batch Gradient Descent* und genauer als *Stochastick Gradient Descent* zu sein.

Gradient Descent tritt in der Literatur in vielen Abwandlungen auf (nicht abhängig von der *Batch Size*). Im weiteren Verlauf wurde der elementare *Gradient Descent* Algorithmus, sowie auch zwei seiner Erweiterungen, dargestellt:

1. *Gradient Descent* mit *Momentum*
2. *RMSprop*

2.2.1 Gradient Descent

Die allgemeinste Form vom *Gradient Descent* ist das Abstiegsverfahren, in dem es darum geht, eine stetige differenzierbare Funktion zu minimieren. Die zentrale Idee vom Abstiegsverfahren

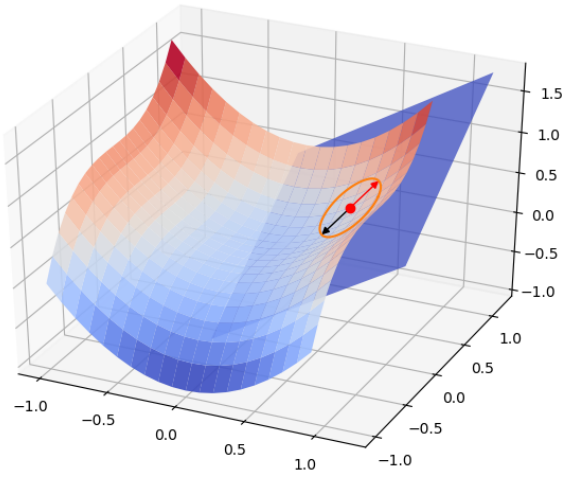
ist wie folgt: Ist man an einem Punkt $x \in R^n$, so sucht man eine Richtung d aus, in welche der Funktionswert fällt (Abstieg). Entlang dieser Richtung d geht man so lange, bis man den Funktionswert von Funktion $f(x)$ hinreichend verkleinert hat. Das Gradienten-Verfahren geht ähnlich vor, ist aber effizienter, weil man nicht in die Richtung eines beliebigen Abstiegs, sondern in die Richtung des steilsten Abstiegs (deswegen oft *Steepest Descent* genannt) geht, was dazu führt, dass man sich schneller dem Minimum nähert. Man könnte das Verfahren formal darstellen, in dem (Bischl 2016):

$f(x)$ eine beliebige, uneingeschränkte, differenzierbare Zielfunktion, welche man minimieren möchte, ist(**Abbildung 6**)

- Der Gradient $\nabla f(x)$ zeigt immer in die Richtung des steilsten Anstiegs (roter Pfeil)
- $-\nabla f(x)$ zeigt somit in die Richtung des steilsten Abstiegs (schwarzer Pfeil)

Man folgt der Richtung des negativen Gradienten so lange, bis man den Funktionswert hinreichend minimiert hat.

Abbildung 6: 3D Gradient Descent



Quelle: Eigene Darstellung

Wenn man während der Minimierung der Funktion $f(x)$ am Punkt x_t steht, kann man diesen Punkt verbessern, indem man folgenden Schritt durchführt:

$$x_{t+1} = x_t - \eta \nabla f(x_t) \tag{1}$$

η ist die Schrittlänge, die auch Lernrate genannt wird. Die Wahl der entsprechenden Schrittlänge ist entscheidend: eine zu große Schrittlänge könnte dazu führen, dass man nie nah genug an das Minimum kommt und im schlimmsten Fall über dem Minimum hin und her springt - eine zu kleine Schrittlänge hingegen könnte zur Folge haben, dass man eine sehr große Anzahl an Iterationen brauchen würde, um beim Minimum anzukommen. Zusammenfassend ist das *Gradient Descent* Verfahren einfach und unkompliziert, weshalb es weit verbreitet und zudem auch

bei hochdimensionalen Optimierungsproblemen anwendbar ist (Bischl 2016). Zu den Nachteilen zählen unter anderem, dass es in einem lokalen Minimum stecken bleiben kann und nie am globalen Minimum ankommen würde. Zudem kann es vorkommen, dass es für schlecht konditionierte Probleme in einem zunehmenden Zick-Zack-Kurs zum Minimum schreitet, wodurch wiederum die Anzahl der Iterationen und somit die Dauer des Verfahrens erheblich ansteigen könnte.

2.2.2 Gradient Descent mit Momentum

Den *Update* Schritt in **Gleichung 1** kann man auch als *Update* der Parameter des Modells darstellen:

$$\theta_{t+1} = \theta_t - \eta \nabla f(\theta_t) \tag{2}$$

Da im folgenden Updates mit Hilfe von *Mini-Batch GD* durchgeführt werden und somit nicht mit Hilfe des ganzen Datensatzes, muss die **Gleichung 2** um die entsprechende Observation $x^{(i:i+n)}$ und $y^{(i:i+n)}$ erweitert werden - wo n die *Batch Size* und i das erste Element des *Batches* sind:

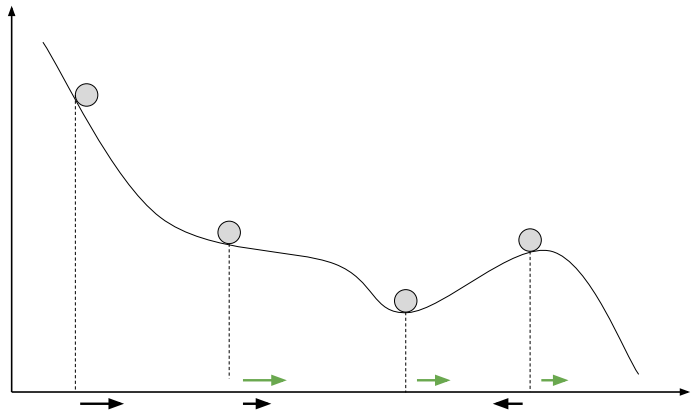
$$\theta_{t+1} = \theta_t - \eta \nabla f(\theta_t; x^{(i:i+n)}; y^{(i:i+n)}) \tag{3}$$

Gradient Descent mit Momentum ist eine Erweiterung des Optimierungsalgorithmus, um das Problem mit dem zunehmenden Zick-Zack-Kurs bei schlecht konditionierten Funktionen zu bekämpfen. Anstatt in die Richtung des steilsten Abstiegs zu gehen, wird ein gewichtetes (mit Gewicht γ) Update der letzten Iteration mit auf den mit Schrittlänge multiplizierten Gradienten aufaddiert. Um die folgenden Gleichungen übersichtlicher zu machen, wird der *Mini-Batch GD* Gradient $\nabla f(\theta_t; x^{(i:i+n)}; y^{(i:i+n)})$ als g_t und das Update in Iteration t als ν^t, dargestellt. *Gradient Descent mit Momentum* nimmt im Endeffekt die folgende Form an (Ruder 2016):

$$\theta_{t+1} = \theta_t - (\eta g_t + \gamma \nu_{t-1}) \tag{4}$$

Wenn man sich das *Gradient Descent* Verfahren als einen Ball vorstellt der in Richtung des Minimum rollt, kann man den Parameter γ als *Momentum*, also Schwung, interpretieren, den der Ball im Laufe das Fortbewegens aufnimmt. Bei dem ursprünglichen Algorithmus ist dies nicht der Fall und der Ball würde sich bewegen, als ob er keinen Schwung aufnehmen könnte. Dies erlaubt es schneller zu konvergieren und zum globalen bzw. lokalen Minimum zu gelangen. Darüber hinaus könnte *Momentum* dabei helfen, nicht in einem lokalen Minimum stecken zu bleiben, da der aufgenommene Schwung dazu führen könnte, dass man ein solches Minimum überspringen und weiter in Richtung globales Minimum iterieren könnte.

Abbildung 7: Momentum Darstellung



Quelle: Eigene Darstellung

In **Abbildung 7** wurde genau diese Situation dargestellt. Der Ball rollt in Richtung des steilsten Abstiegs - hier befindet man sich in einem zweidimensionalen Raum. In der ersten Position (von links) bewegt er sich nach rechts - mit der Kraft des Gradienten, der an dieser Stelle berechnet wurde (schwarzer Pfeil). In der zweiten Position ist diese Kraft deutlich schwächer, da die berechnete Tangente nicht mehr so steil ist wie an der ersten Stelle. Dafür sieht man jetzt den grünen Pfeil, der die Kraft vom *Momentum* repräsentiert, die im ersten Schritt entstanden ist. An der dritten Position ist die Kraft des Gradienten, die sich auf den Ball auswirkt gleich Null und würde damit im Falle des normalen *Gradient Descent* bedeuten, dass der Ball stecken bleibt und nicht über die nächste Steigung kommt. *Momentum* hingegen hat den Ball zur vierten Stelle gebracht, wo wiederum der berechnete Gradient den Ball zurück drückt (negativer schwarzer Pfeil an Position vier), aber der übriggebliebene Schwung (grüner Pfeil an Position vier) ausreichend ist, um ihn weiter in Richtung Globalen Minimums zu befördern.

2.2.3 RMSprop(Root mean square) Gradient Descent

Eine weitere Variante des *Gradient Descent* Verfahrens ist *RMSprop GD*. Die Idee ist hier, dass man nicht alle Parameter mit der gleichen Lernrate η gewichtet. Die Lernrate variiert in der Hinsicht, dass Parameter, die mit Klassen, die häufiger auftreten, in Verbindung stehen, ein kleineres η erhalten - andererseits erhalten Parameter, die Einfluss auf Klassen haben, die nicht so häufig vorkommen, eine größere Lernrate, um eine dementsprechende Hochgewichtung zur erreichen. Zusätzlich wird der η Parameter von der Iteration, in der man sich befindet, abhängen. Diese Merkmale sind besonders von Vorteil für sparse Daten, in denen man unterschiedliche Häufigkeiten der Zielklasse hat (dies ist auch der Fall bei der Textegenerierung). Der Vollständigkeit halber muss man sagen, dass die oben beschriebenen Erweiterungen dem Algorithmus *Adagrad* (**Gleichung 5**) entsprechen, der eine Basis des *RMSprop* Algorithmus darstellt (Ruder 2016):

$$\theta_{t+1,j} = \theta_{t,j} - \frac{\eta}{\sqrt{G_{t,jj} + \epsilon}} g_{t,j} \tag{5}$$

In **Gleichung 5** enthält die Matrix $G_{t,jj}$ auf der Diagonalen j, j die Summe der quadratischen Gradienten hinsichtlich des Parameters θ_j bis zum entsprechenden Zeitpunkt t. Der ϵ Parameter (ein sehr kleiner Wert) soll der Dividierung durch Null vorbeugen. Vektorisiert erhält man:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \otimes g_t \tag{6}$$

Ein großer Vorteil hierbei ist, dass sich die Lernrate alleine anpasst und mit der Anzahl der Iterationen immer näher an Null geht, was bedeutet, dass die Schritte mit Annäherung an ein Minimum immer kleiner werden, wodurch das *tunen* der Lernrate überflüssig wird. Der Nachteil davon ist, dass die Schrittlänge durch die anwachsende Anzahl der Iterationen, voranschreitend ausgelöscht wird. *RMSprop* ermöglicht es, diese Auslöschung der Lernrate zu umgehen, indem anstatt der Matrix mit quadratischen Gradienten auf den Diagonalen, ein sogenannter *Moving Average* $E[g^2]_t$ verwendet wird (Ruder 2016):

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{7}$$

Der *Moving Average* wird aus dem aktuellen quadrierten Gradienten und dem gemittelten Gradienten aller vorheriger Iterationen bis Zeitpunkt $t - 1$ berechnet:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2 \tag{8}$$

Der Parameter γ ist hier so ähnlich wie der vorher vorgestellte γ Parameter aus dem *Momentum* Beispiel. Die Erfinder dieses Algorithmus schlagen als guten *default* Wert für $\gamma = 0.9$ und für die Lernarate $\eta = 0.01$ vor (Ruder 2016). Der Name *RMSprop* kommt daher, das man im Nenner eigentlich nichts anderes hat, als das quadratische Mittel (*Root mean square*) - Fehler Term des Gradienten:

$$\theta_{t+1} = \theta_t - \frac{\eta}{RMS[g^2]_t} g_t \tag{9}$$

2.3 Neural Networks

Dieses Kapitel befasst sich mit den Grundlagen von *Deep Learning*. Dies wird die Voraussetzung sein, um die später erklärten *Recurrent Neural Networks* zu verstehen. Anfangs gibt es eine kurze Einführung in die Geschichte von *Deep Learning* und anschließend wurde in den darauf folgenden Kapiteln auf die grundlegenden Bausteine eines Neuronalen Netzes eingegangen. Folgende Begriffe werden eingeführt: *Neuron*, *Backpropagation*, Verlustfunktionen und *Activationfunctions*, da sie sowohl für Neuronale Netze, sowie für *Recurent Neural Networks* relevant sind.

2.3.1 Die Geschichte von Deep Learning

Die ersten Anwendungen von *Deep Learning* gehen auf das Jahr 1943 zurück - Walter Pitts und Warren McCulloch erschufen ein erstes Computerprogramm, das aus einer Kombination von Algorithmen und Mathematik entstand (Walter Pitts 1943). Viele Forscher waren skeptisch, wenn es um die praktische Anwendung von *Neronalen Netzen* ging, da es anfangs keine *Deep Networks* waren und sie damit in ihrer Verarbeitung sehr limitiert waren. Das führte dazu, dass die wissenschaftliche Arbeit in diesem Gebiet bis in die späten 70ger und Anfang der 80ger eingeschränkt wurde.

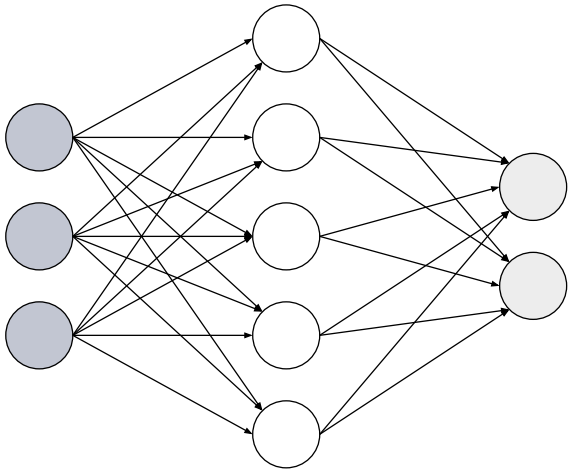
In dieser Zeit wurde eines der größten Hindernisse, das die weitere Entwicklung hinderte - also die Erfindung eines Optimierungsalgorithmus (*Backpropagation*), der es erlaubt, effizient mit Hilfe von *Gradient Descent* rückwirkend Parameter zu trainieren, überwunden (LeCun 1988). Die erste erfolgreiche Anwendung gelang Yann LeCun, der 1989 den *Backpropagation* Algorithmus mit *Convolutional Neural Networks* verbunden hatte, um es anschließend auf einem Datensatz von handgeschriebenen Zahlen auszuprobieren. Das daraus resultierende Neuronale Netz, das *LeNet* genannt wurde, wurde 1990 von dem Postdienst der Vereinigten Staaten für die automatische Erkennung von Postleitzahlen verwendet. Viele Erwartungen wurden in die weitere und schnelle Entwicklung von *Deep Learning* gesetzt, die aufgrund von computationalen Gründen nicht erfüllt werden konnten. Dies führte dazu, dass die Forschung zum zweiten Mal zurückgegangen ist und erst wieder größeres Aufsehen erlangte, nachdem im Jahr 1997 Sepp Hochreiter und Jürgen Schmidhuber die LSTM Zelle für *Recurrent Neural Networks* entwickelt haben (Sepp Hochreiter 1997).

Im folgendem Jahrzehnt wurde die Rechengeschwindigkeit dank GPU's (*Graphics Processing Units)* um das 1000-fache erhöht, was dazu führte, dass der *Deep Learning* Forschung zunehmend Aufmerksamkeit gewidmet wurde. Heutzutage werden, dank stets optimierender Rechenpower, immer mehr Probleme mit Neuronalen Netzen gelöst, wie zum Beispiel: Autonomes Fahren, Übersetzungen, Bild- und Audio-Verarbeitung.

2.3.2 Neuronen und Layers

Ein *Neuron* ist das Basis-Element jedes *Neural Networks*. Sein *Input* könnte sowohl von außen kommen, sowie von einem früherem *Neuron* (Alpaydin 2010). In **Abbildung 8** sind drei *Layers* von *Neuronen* dargestellt: die ersten dienen als *Input Layer* (dunkelgraue *Neuronen*) und enthalten *Neuronen*, die ihren *Input* von außen erhalten werden - dieser *Input* wird nachstehend als x_j $j \in 1, ..., n$. beschrieben. Weiterhin wird der erste *Layer* mit Hilfe von Gewichten w_{hj} (weiße *Neuronen* z_h $h \in 1, ..., m$) mit dem sogenannten *Hidden Layer*, der wiederum mit dem *Output Layer* verbunden ist (hellgraue *Neuronen* y_i $i \in 1, ..., o$), aufgezeigt. Zwischen dem *Hidden Layer* und dem *Output Layer* befinden sich auch Gewichte v_{ih}.

Abbildung 8: Beispiel Neuronales Netz



Quelle: Eigene Darstellung

Der *Output* ist somit die gewichtete Kombination der *Hidden Units (Neurons)*:

$$y_i = \sum_{h=1}^{m} v_{ih} z_h + v_{i0} = v_i^T z \tag{10}$$

v_{i0} repräsentiert den Wert für die Neuronen-spezifische Verzerrung (*Bias*). Die Werte von z_h wurden aus der gewichteten Kombination von x_j berechnet:

$$z_h = \sum_{j=1}^{n} w_{hj} x_j + w_{h0} = w_h^T x \tag{11}$$

Wenn man es jetzt mit einem Klassifizierungsproblem zu tun hat, dann würden die *Output Units* in Kombination mit einer Verlustsfunktion bzw. Aktivierungsfunktion, darüber entscheiden, welche Klasse die wahrscheinlichste sein würde. Klasse C_i wird gewählt, wenn $y_i = \max_o(y_o)$ ist (Alpaydin 2010).

Der Name *Deep Learning* stammt daher, dass man theoretisch eine beliebige Anzahl an *Hidden Layers* in einem *Neural Networks* verbinden könnte, was zu *Deepen* Netzen führen würde. Die Aufgabe eines *Neural Network* ist es, die Gewichte w_{hj} und v_{ih} so zu erlernen, dass am Ende die richtige Klasse vorausgesagt wird. Die einfachste Form eines solchen Netzes würde ein *Input* und ein *Output Neuron* sein, die im Endeffekt genutzt werden könnten, um eine lineare Regression zu implementieren:

$$y = wx + w_0 \tag{12}$$

In **Gleichung 12** ist w, genauso wie in der linearen Regression, die Steigung und w_0 stellt den *Intercept* dar. Wenn ein Datensatz gegeben ist, könnte man mit Hilfe der entsprechenden Verlustfunktion die Regression-Parameter finden.

2.3.3 Activation Function

Der Zweck einer Aktivierungsfunktion in einem *Deep-Learning-Kontext* ist es, sicherzustellen, dass die Darstellung vom *Input* auf einen anderen Raum im *Output* abgebildet wird. Sie sorgt erstens dafür, dass ein gewichteter *Neuron* aktiviert werden kann - also, dass seine Information weiter gegeben werden kann. Zweitens sorgt sie dafür, dass sich der Raum des *Inputs* zum *Output* ändert, um tiefe nicht lineare Repräsentationen der Daten extrahieren zu können. Die Aktivierung eines *Neuron*, also das Weitergeben einer relevanten Information, könnte wie folgt aussehen: ein Schwellenwert entscheidet, ob der Wert eines *Neuron* ausreichend groß ist, um weitergegeben zu werden (Sharma 2017).

Ein solcher Mechanismus würde entweder eine 1 (aktiviert) oder eine 0 (nicht aktiviert) einem Wert des Neuron zuordnen. Ein Beispiel einer Funktion, die genau das macht, sieht man in **Tabelle 3** unter dem Namen *Step Function*. Allerdings ergibt sich hierbei ein Problem: bei einer *Multi class* Klassifizierung dürfte nur ein Wert der *Neuronen* im *Output Layer* eine 1 zugeschrieben werden (die der wahren Klasse) und dem Rest müsste eine 0 zugeordnet werden. Ein solches Problem ist allerdings schwer in einem *Neural Network* zu trainieren, da in einem *Multi class* Klassifikationsproblem die Wahrscheinlichkeit groß ist, dass mehrere *Neuronen* aktiviert wurden, wodurch man mehrere Klassen gleichzeitig wählen müsste. Eine bessere Lösung wäre es, wenn man möglichst unterschiedliche Aktivierungen den Werten der *Output Neuronen* zuordnen könnte, um im Nachhinein den *Neuron* mit der größten Aktivierung wählen zu können. Dies führt zu eindeutigen Ergebnissen und ist einfacher zu trainieren.

Die einfachste Alternative wäre eine lineare *Activation Function* (**Tabelle 3** *Identity*), die proportional zu dem *Input* ist (die gewichtete Summe). Darüber hinaus erhält man keine binäre Lösung, sondern einen Bewertungsbereich von Aktivierungen, aus dem man zum Beispiel das Maximum als Entscheidung wählen könnte. Das Problem, das hier auftaucht, ist, dass man keine nicht linearen Repräsentationen erlernen kann, da sogar mehrere *Layer*, die aufgestockt werden und mit einer linearen *Activation Function* aktiviert werden, immer noch eine lineare Kombination des *Inputs* darstellen.

Sigmoid (**Tabelle 3** *Sigmoid*) verbindet beide guten Eigenschaften der vorher genannten *Activation Functions*. Erstens ändert sich der Raum des *Inputs* auf den *Output*, was es erlaubt, nicht lineare Repräsentationen zu erhalten und zweitens ist der Aktivierungswert in einem eingegrenzten Bereich zwischen 0 und 1, wobei nicht nur die Extrem-Werte vergeben werden können, sondern eine beliebige Zahl dazwischen. Die Werte für die Argumente zwischen -2 und 2 sind sehr steil verteilt und die Veränderungen im Funktionswert sind groß bei kleinen Veränderungen in x. Diese Eigenschaft führt dazu, dass Aktivierungen schnell in eine der beiden Enden gedrückt werden, was dabei hilft, Abgrenzungen für Prädiktionen zu schaffen. Der Nachteil bei der *Sigmoid* Funktion ist allerdings, dass die Änderungen an den Enden der Funktion bei Veränderungen in x, sehr klein sind, da die Funktion dort flach ist - das wiederum bewirkt, dass der berechnete Gradient an diesen Stellen auch klein sein wird und somit die *Updates* bei dem *Backpropagation* ausgelöscht werden. Dieses Problem wird *Vanishing Gradient Problem* genannt und wurde, wie der *Backpropagation* Algorithmus, im späteren Verlauf der Arbeit beschrieben. Eine skalierte Version der *Sigmoid* Funktion ist die *Tanh* Funktion (**Tabelle 3** *Tanh*), die den Vorteil hat, dass allgemein der Gradient an den Enden stärker ist (Ableitung steiler) - das *Vanishing Gradient Problem* ist jedoch immer noch präsent. Die Aktivierungswerte befinden sich im Intervall $(-1, 1)$.

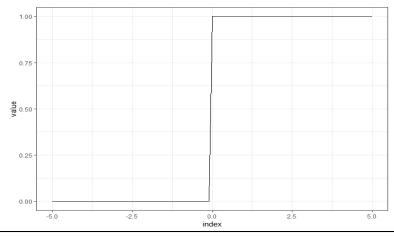
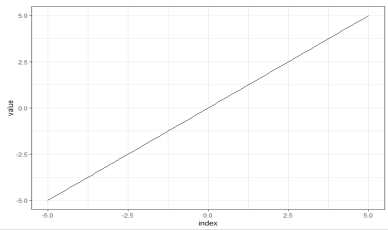
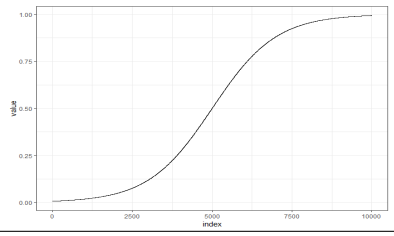
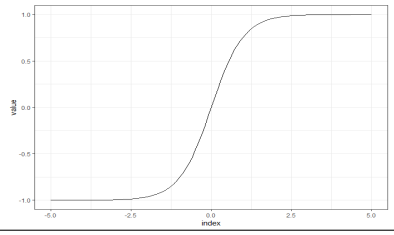
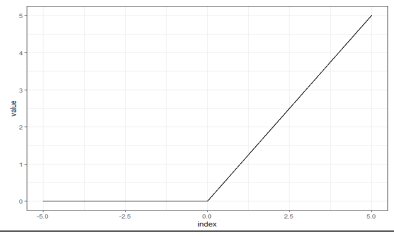
Die am meisten genutzte *Activation Function* ist *ReLU* (*Rectified Linear Unit*). Diese Funktion ist im positiven Bereich $f(x) = x$ und im negativen Bereich gleich Null. Es ist also eine nicht lineare Funktion, deren Kombination auch wieder eine nicht lineare Funktion darstellt. Mit deren Hilfe kann man eine beliebige andere Funktion approximieren. Ein weiterer großer Vorteil von *ReLU* ist die *sparsity* der Aktivierungen, die man erhält. Bei zum Beispiel der *Sigmoid* und *Tanh* Funktion werden fast alle *Neuronen* aktiviert und zum Entscheiden über die vorausgesagte Klasse verwendet. Dies sind dichte Aktivierungsfunktionen, die im Hinblick auf Zeit und Rechenaufwand sehr kostspielig sind. *ReLU* hingegen hält manche *Neuronen* auf Null, wodurch sie bei manchen Iterationen gar nicht unter Betracht genommen werden. Aus diesem Grund hat die Aktivierung eine geringere Dichte und ist einfacher zu trainieren. Diese Eigenschaft könnte allerdings auch ein Nachteil sein, da der Gradient im negativen Teil Null ist und es nicht möglich ist, für solche *Neuronen* auf die Variation im Gradienten *Update* zu reagieren. Dieses Problem nennt sich *Dying ReLU Problem* und kann durch verschiedene *ReLU* Abwandlungen begrenzt werden, wie z.B. *leaky ReLu*, bei der man anstatt einer horizontalen Linie im negativen Bereich eine Funktion $f(x) = 0.1x$ hat, die es erlaubt, die *Neuronen*, die in diesem Aktivierungsbereich liegen, zu befreien.

Die letzte zu beschreibende Aktivierungsfunktion ist die *Softmax* Funktion, die im Allgemeinen die Wahrscheinlichkeiten jeder Zielklasse über alle möglichen Zielklassen berechnet. Diese berechneten Wahrscheinlichkeiten haben die Eigenschaft, dass sie sich zu 1 aufsummieren, wodurch man sich im späterem Schritt für die Zielklasse mit der höchsten Wahrscheinlichkeit entscheiden kann. Die Gleichung für die *Softmax* Aktivierungsfunktion sieht wie folgt aus:

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}, \; k, i \in 1, ..., k \tag{13}$$

wobei K die Anzahl der Klassen ist und i die i-te Klasse repräsentiert.

Tabelle 3: Activation Functions

Name	Abbildung	Gleichung
Step Function		$S(x) = \begin{cases} 0, & \text{if } x < 0. \\ 1, & \text{if } x > 0. \end{cases}$
Identity		$f(x) = x$
Sigmoid		$\sigma(x) = \frac{1}{1+e^{-x}}$
Tanh		$tanh(x) = \frac{2}{1+e^{-2x}} - 1$
Rectified Linear Unit (ReLU)		$R(x) = \begin{cases} 0, & \text{if } x < 0. \\ x, & \text{if } x > 0. \end{cases}$

Quelle: Eigene Darstellung (Sharma 2017)

2.3.4 Verlustfunktionen

Eine Verlustfunktion zeigt, wie unzufrieden man mit einer Fehlprädiktion ist (also mit der Abweichung der Schätzung von dem wahren Wert der Funktion). Der *Verlust* ist ein nicht negativer Wert, bei dem die Robustheit mit absteigender Verlustfunktion zunimmt. Die allgemeine Form

der Verlustfunktion sieht man in **Gleichung 14**:

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(x_i, \theta)) \tag{14}$$

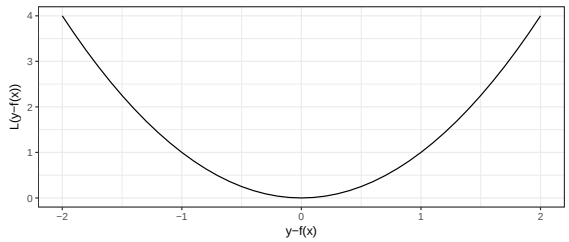
Wobei y_i der wahre Wert oder Klasse ist und $f(x_i, \theta)$ die dazugehörende Prädiktion darstellt (θ sind die zu trainierenden Parameter). Im *Deep Learning* Kontext ist die Funktion $f(x_i)$ die entsprechende Aktivierungsfunktion und $x_i = \{x_i^1, x_i^2, ..., x_1^m\}$ das Trainings *Sample* der Größe m. Folgende Charakteristiken von Verlustfunktionen sind relevant (Bischl 2016):

1. Differenzierbarkeit
2. Robustheit
3. Konvexität

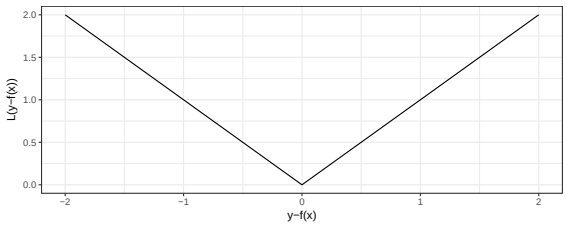
Die Differenzierbarkeit einer Verlustfunktion erleichtert das Optimieren, die Robustheit zeigt wie stark eine Verlustfunktion auf Abweichungen reagiert und die Konvexität garantiert, dass es nur ein globales Minimum der Funktion gibt. Zwei Bespiele von Verlustfunktionen:

Abbildung 9: Beispiele von Verlustfunktionen

(a) L_2-Verlust



(b) L_1-Verlust



Quelle: Eigene Darstellung

Die L_2-Verlustfunktion (**Abbildung 9 a)**) ist differenzierbar und konvex, was verursacht, dass die Optimierung einfacher ist als im Fall des L_1-Verlustes (**Abbildung 9 b)**) der nicht differenzierbar ist. Beide Funktionen reagieren anders auf Abweichungen ($y - f(x)$). Der L_2-Verlust ist sensibler und bestraft zum Beispiel eine Abweichung von 2 mit dem Verlust 4 - der L_1-Verlust hingegen nur mit 2, was bedeutet, dass der L_1-Verlust robuster ist als der L_2-Verlust. Diese Eigenschaft zeigt auf, dass man gut den Erwartungswert des Modells mit dem L_2-Verlust und zum Beispiel gut den Median mit dem L_1-Verlust, schätzen kann. Der L_2-Verlust wird, wie gerade vorgestellt, oft bei der linearen Regression genutzt und hat folgende Form:

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2} (y_i - f(x_i))^2 \tag{15}$$

Und die Gleichung des L_1-Verlustes wurde in **Gleichung 16** gezeigt:

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} |y_i - f(x_i)| \tag{16}$$

Die Verlustfunktion, die für diese Arbeit von Interesse ist, ist die *Categorical Cross-entropy*. Die *Categorical Cross-entropy* lässt sich aus der Kullback-Leibler-Divergenz erschließen. Die KL-Divergenz wird auch relative Entropie genannt und erlaubt es, ein Maß der Unterschiedlichkeit zwischen zwei Wahrscheinlichkeitsverteilungen zu berechnen. Die relative Entropie hat folgende Form:

$$\begin{aligned}\mathcal{L} &= -\frac{1}{n}\sum_{i=1}^{n} D_{KL}(y_i||\hat{y}_i) \\ &= -\frac{1}{n}\sum_{i=1}^{n}[y_i log(\frac{y_i}{\hat{y}_i})] \\ &= \underbrace{\frac{1}{n}\sum_{i=1}^{n}(y_i log(y_i))}_{\text{entropy}} \underbrace{-\frac{1}{n}\sum_{i=1}^{n}(y_i log(\hat{y}_i))}_{\text{cross-entropy}}\end{aligned} \tag{17}$$

Die *Cross-entropy* gibt an, wie weit man mit der geschätzten Verteilung durch das Modell von der wahren Verteilung entfernt liegt (Hao 2017). Sie wird oftmals bei Neuronalen Netzen mit *Softmax* Aktivierungfunktion verwendet, da hierbei die Aktivierungen als Wahrscheinlichkeitsverteilungen verstanden werden können. Die Verlustfunktion hat folgende Form:

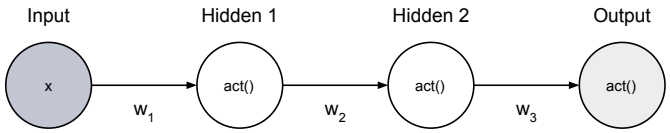
$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{k} y_i \log(f(x_i)) \tag{18}$$

wobei k die Anzahl der Klassen aus der *Softmax* und $f(x_i)$ die Aktivierung des x_i-ten Element darstellt.

2.3.5 Backpropagation

Im folgenden wird der Algorithmus vorgestellt, mit dessen Hilfe man in der Lage ist, die Gewichte des *Neural Networks* zu trainieren. Dieser Algorithmus heißt *Backpropagation* (LeCun 1988) und wird anhand eines vereinfachten *Neural Network* (**Abbildung 10**) erklärt (Kapur 2017). In diesem Beispiel gibt es einen Input, zwei *Hidden Layers* und ein *Output Layer*. In den jeweiligen *Layern* befinden sich einfachheitshalber nur ein *Neuron*, der mit dem darauf folgenden *Layer* auch nur mit einem *Weight* w_i gewichtet wird. Die *Activation Functions* werden durch *act*() markiert. Diese Funktionen werden vorerst nicht festgelegt, könnten jedoch eine beliebige Aktivierung sein (**Unterkapitel 2.3.3**).

Abbildung 10: Simple Neural Network



Quelle: Eigene Darstellung (Kapur 2017)

Bevor man mit dem *Backpropagation* beginnt, wird zuerst der *Forward Pass* durchgeführt, um einen entsprechenden *Output* zu erhalten. Dies erreicht man durch das Durchführen des *Inputs* durch die aufeinanderfolgenden *Layer*, die durch die *Activation Functions* der *Hidden Layer* und im Endeffekt des *Output Layers* transformiert werden. Den *Output* kann man wie folgt berechnen:

$$Hidden1 = act(w_1 x) \rightarrow Hidden2 = act(w_2 Hidden1) \rightarrow Output = act(w_3 Hidden2)$$

Wenn man die ersten zwei Gleichungen in die Dritte einsetzt, erhält man die Formel für den *Output*:

$$Output = act(w_3 act(w_2 act(w_1 x)))$$

Im folgenden wird die Ableitung dieser Gleichung benötigt. Darüber hinaus ist der *Output* zusätzlich in eine Verlustfunktion eingebunden, wodurch das Ergebnis mit dem wahren Wert verglichen wird (**Unterkapitel 2.3.4**). Eine Beispiel Ableitung in Bezug auf den Parameter w_1 könnte man mit Hilfe der Kettenregel bestimmen:

$$\frac{\partial error}{\partial w_1} = \frac{\partial error}{\partial Output} \frac{\partial Output}{\partial Hidden2} \frac{\partial Hidden2}{\partial Hidden1} \frac{\partial Hidden1}{\partial w_1}$$

Wichtig ist, dass die Verlustfunktion immer noch eine Funktion des *Inputs* ist. Als nächstes würde die gleiche Ableitung in Bezug auf alle anderen Parameter des *Neural Networks* berechnet werden. Diese berechneten Gradienten dienen dem *Update* des Gradienten mit einem der in **Unterkapitel 2.2** besprochenen Optimierungsalgorithmen. Daher kommt der Name *Backpropagation*: man optimiert die Gewichte in einer vom Verlust berechneten rückwirkenden Art über die jeweiligen *Layers*. Dies ist natürlich nur ein einfaches Beispiel für einen unwahrscheinlichen Fall eines solchen *Neural Networks*, aber in Wirklichkeit ändert sich das Vorgehen nicht wirklich, bei z.B einem Netz wie in **Abbildung 8**.

Erwähnenswert ist hierbei, dass die Konvergenz des Algorithmus von den Verlust- und Aktivierungsfunktionen, die im Netz verwendet werden, stark abhängig ist. Als Beispiel könnte man den MSE (L_2) mit der *Binary Cross-entropy* vergleichen, bei denen die Ableitungen mit *Sigmoid Activationfunction* im *Output Layer* berechnet wurden (θ die zu optimierenden Parameter im Netz): Der Gradient für $\hat{y}_i = \sigma(Z_i) = \sigma(\theta^T x_i)$ würde für den MSE wie folgt aussehen

$\frac{\partial \mathcal{L}}{\partial \theta} = -(y - \sigma(z))\sigma(z)'x$ und für die *Cross-entropy* $\frac{\partial \mathcal{L}}{\partial \theta} = -(y - \sigma(z))x$. Der Unterschied ist also, dass im Fall des MSE zusätzlich die Ableitung der *Sigmoid* Funktion $\sigma(z)'$ enthalten ist, die nah an Null ist, wenn $\sigma(z)$ Richtung 1 oder 0 geht und das Maximum erreicht, wenn $\sigma(z)$ 0.5 ist. Dies verursacht, dass der Gradient klein ist, wenn der Verlust groß ist, wobei genau dass Gegenteil erwünscht ist. Die Cross-entropy hat den zusätzlichen $\sigma'(z)$ nicht und verhält sich somit so, wie man wollen würde.

2.4 Recurrent Neural Network

Dieses Kapitel beschäftigt sich vertieft mit dem Thema *Recurrent Neural Networks*, die in dieser Arbeit verwendet werden, um die Textgenerierung durchzuführen. *Recurrent Neural Networks* werden hauptsächlich bei sequentiellen Problemstellungen angewandt - wie zum Beispiel Zeitreihenanalyse oder wie in diesem Fall Texte. Am Anfang dieses Kapitels wurde eine kurze Einleitung in das Thema dargestellt, um im folgenden gewisse Erweiterungen des *Backpropagation* Algorithmus für sequentielle Modelle und Probleme, die bei *Recurrent Neural Networks* auftreten können, zu beschreiben. Die zwei am weitesten verbreiteten Modelle wurden dann zum Ende des Kapitels dargestellt: Erstens das *Long Short Term Memory* Modell und zweitens dessen Abwandlung, also das *Gated Recurrent Unit (GRU)* Modell.

2.4.1 Sequentielle Modelle

Der Unterschied zwischen einem sequentiellen und nicht sequentiellen Modell ist der, dass, so wie der Name schon sagt, die Reihenfolge der Daten im behandelten Problem mit unter Betracht genommen werden kann. Dies bedeutet, dass in einem normalen *Neural Network* die Sequenz, in der die Daten ins Modell eingegeben und verarbeitet werden, keine größere Bedeutung haben. Bei sequentiellen Modellen ist es besonders wichtig, die historischen Werte in einer chronologischen Reihenfolge zu verwenden, um die Prädiktionen von den aufeinanderfolgenden Ereignissen abhängig zu machen. Die am meisten verbreiteten sequentiellen Daten sind Texte, die entweder als Sequenz von Wörtern oder, so wie in diesem Fall, als Sequenz von Zeichen verstanden werden können (Francois Chollet 2018). Ein Unterschied zu einem *Feedforward* Modell, das es ermöglicht, eine Sequenz als tatsächliche Sequenz zu betrachten, ist die *Memory*, die es erlaubt, die Vergangenheit zu behalten und die nächsten *Time Steps* in Abhängigkeit der vorherigen zu behandeln. Die Daten werden also nicht alle auf einmal ins Modell eingegeben, sondern eins nach dem anderen und werden durch eine rezidive (*Recurrent*) Verknüpfung verbunden, um die *Memory* des Modells zu simulieren. Es ist möglich, eine *Recurrent* Verbindung zu entfalten (dies wurde im späteren **Unterkapitel 2.4.5** beschrieben), um es stattdessen in der äquivalenten *Feedforward* Form darzustellen. Jedoch ist zu beachten, dass bestimmte Anpassungen zum *Backpropagation* Algorithmus beigefügt werden müssen.

Eine wichtige Eigenschaft von *RNNs* ist, dass sie in der Theorie in der Lage sein müssten, Langzeitabhängigkeiten zu erlernen. Es stellt sich jedoch heraus, dass dies in der Praxis unmöglich ist, da solche Modelle unter dem *Vanishing Gradient Problem* (**Unterkapitel 2.4.3**) leiden, dass man auch häufig bei anderen *Deep*-en Architekturen beobachten kann. Ein *RNN* ist also in der Lage, zum Beispiel den Satz “Die Bäume wachsen im ___” mit dem Wort “Wald” zu Ende zu bringen. Wenn es jedoch zu langen Abhängigkeiten kommt, bei denen zwischen zwei Informationen viele *Time Steps* liegen, die nichts mit einer bestimmten Information zu tun haben, wird die korrekte Prädiktion schwierig (z.B “Er kommt aus Deutschland ... Er spricht fließend ___” → “deutsch”). Die Modellierung von langfristigen Abhängigkeiten ist jedoch mit Hilfe von *GRU* und *LSTM* möglich.

2.4.2 (Truncated) Backpropagation through time

Backpropagation through time (BPTT) ist das Äquivalent zu dem *Backpropagation* Algorithmus aus **Unterkapitel 2.3.5** für *RNNs* mit sequentiellen Daten. Man könnte sich den Algorithmus so vorstellen, dass man wie im **Unterkapitel 2.4.5** ein *LSTM* Modell in sequenziell aufeinanderfolgenden Modellen auflöst (**Abbildung 11**), die man sich wie *Layers* in einem *Convolutional Neural Network* vorstellen kann. In *RNNs* wird an jedem *Time Step* eine Observation eingelesen, die zu dem gleichen Zeitpunkt auch einen *Output* liefert (Brownlee 2017*a*). Es ist also möglich, jeden dieser Zeitpunkte zu überprüfen, wie genau man mit der jeweiligen Prädiktion war (Verlusfunktion). Wenn man nach dem Berechnen des Verlustes an jedem Zeitpunkt das Netzwerk wieder zusammenfasst und die Verluste kumuliert, ist es möglich, ein Gradienten Update über alle *Time Steps* durchzuführen. Diese Prozedur wird fortgeführt, bis der Verlust hinreichend verkleinert wurde. Ein Problem das jedoch hierbei auftritt, ist dass ein Gradient *Update* für Sequenzen, die sehr viele *Time Steps* haben, sehr kostspielig sein kann, weshalb man in der Praxis meist den *Truncated Backpropagation through time (TBPTT)* verwendet. Der Unterschied besteht darin, dass man nicht wie zuvor die ganze Sequenz nimmt, sondern eher kleine Teile der Sequenz - die *Updates* auf diesen durchzuführen ist deutlich weniger kostspielig. Im Vergleich würde man in diesem Fall das *RNN* wie zuvor, für eine gewisse vorbestimmte maximale Länge (ein gewisser Teil der ganzen Sequenz) entfalten, und würde, für jeden *Time Step* eine Prädiktion machen und den Verlust berechnen. Anschließend könnte man das Modell wieder zusammenfassen und für diesen Teil der Sequenz ein *Update* durchführen. Dies müsste man für alle Untersequenzen wiederholen und so lange fortführen, bis der Verlust, wie immer, ausreichend minimiert wurde. Zum Beispiel mit einer Sequenz die 150 Zeichen hat und in 3 Untersequenzen der Länge 50 geteilt wurde, würde der Gradient *Update* auf jedem der drei Teile separat durchgeführt. Die Anzahl der *Time Steps*, über die man zurück iteriert, muss nicht notwendigerweise der Anzahl der Länge der Untersequenzen entsprechen, aber in den meisten Implementierungen von *RNNs* ist dies der Fall. Der Nachteil ist, dass es nicht möglich ist, Zusammenhänge zu lernen, die länger als die gebildeten Untersequenzen sind.

2.4.3 Vanishing und Exploding Gradients

Zwei häufige Probleme, die bei *Recurrent Neural Network* auftreten können, sind erstens das *Vanishing Gradient Problem* und zweitens das *Exploding Gradient Problem*. Das *Vanishing Gradient Problem* tritt dann auf, wenn man über viele *Layers* zurück *Propagiert* und der Gradienten *Update* in den ersten *Layers* des Netzes immer kleiner wird. Ungünstig hierbei ist auch, wie schon im **Unterkapitel 2.3.3** beschrieben, dass manche Aktivierungsfunktionen verursachen, dass der berechnete Gradient Richtung Null geht (z.B bei *Sigmoid* und *Tanh* Funktionen). Im folgenden Kapitel wurden die *GRU*, wie auch *LSTM* Modelle, die dieses Problem aufheben, dargestellt.

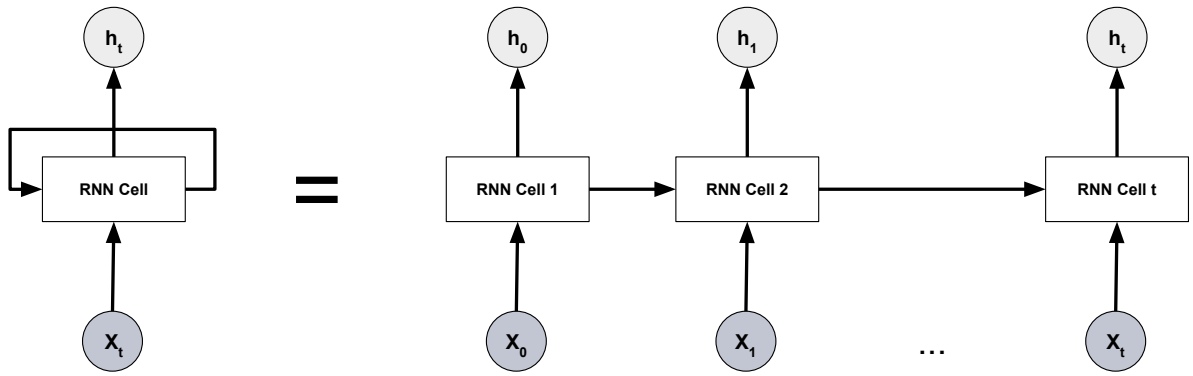
Das zweite Problem das häufig bei *RNNs* vorkommt, ist das Problem des *Exploding Gradient*. Die Prozedur von *Backpropagation through time* wurde im vorherigen Unterkapitel erläutert -

dadurch weiß man, dass der Verlust und somit der Gradienten *Update* über die jeweiligen *Time Steps* kumuliert werden, um das entsprechende *Update* der Gewichte durchzuführen. Das kann dazu führen, dass *Updates* sehr groß werden und im Endeffekt das Lernen aus dem Trainingsdatensatz instabil oder sogar unmöglich wird. Hierfür könnte es verschiedene Lösungen geben: erstens kann man *Gradient Clipping* nutzen, um den *Gradient Update* mit einem Schwellenwert zu begrenzen. Dieser Wert ist also die maximale Größe eines *Updates*, das durchgeführt werden kann. Eine weitere Option ist, wie auch beim *Vanishing Gradient Problem*, das *LSTM* Modell und das Aufteilen der ganzen Sequenz in Untersequenzen.

2.4.4 Simple RNN

In **Abbildung 11** ist auf der linken Seite der Gleichung ein *RNN* Modell mit seiner rezidiven Schleife dargestellt. Die Idee hierbei ist, dass eine *Input* Observation x_t, die an einem *Time Step* eingelesen wird, einen *Output* h_t liefert, der gleichzeitig auch als *Input* der nächsten Zelle verwendet wird. Diese Prozedur kann man auch wie auf der rechten Seite der Abbildung darstellen, in dem die Schleife in eine sequentielle Verbindung aufgelöst wird. Man hat also jetzt mehrere Kopien des gleichen Netzwerkes, aber diesmal hat man anstatt der rezidiven Verbindung eine sequenzielle Verbindung zwischen den jeweiligen Kopien . Das erste Netzwerk erhält sein *Input* x_0 der in der *RNN Cell 1* verarbeitet wird, um letztendlich einen *Output* h_0 zu liefern, der auch als Information an das nächste Netzwerk weitergegeben wird. Das gleiche passiert mit den darauf folgenden *Inputs* $x_1, x_2, ..., x_t$ (dunkelgrau), bei denen entsprechende *Outputs* $h_1, h_2, ..., h_t$ (hellgrau) nach Transformierung in *RNN Cell 2,3,..,t* (weiß) geliefert werden.

Abbildung 11: Unrolled RNN

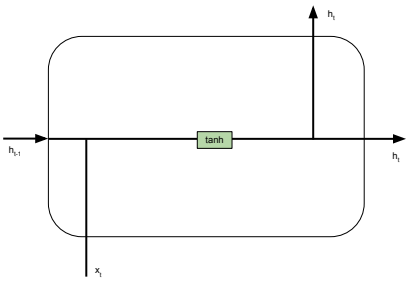


Quelle: Eigene Darstellung (Olah 2015)

In **Abbildung 12** sieht man eine *RNN* Zelle, die den *Output* h_{t-1} aus der letzten Zelle nimmt und es mit dem *Input* x_t an Zeitpunkt t verbindet. In der Zelle befindet sich ein Neuronales Netz

Element (grün) mit einer *Tanh* Aktivierung. Die *Tanh* Aktivierung reguliert den *Output* einer Zelle und presst es in einen Wert zwischen -1 und 1. Ein Netz ohne diese Aktivierung würde schon nach ein paar Zellen große Werte annehmen, wodurch das Problem des *Explodig Gradient* zum Vorschein kommt. Diese Architektur ist jedoch nicht in der Lage lange Abhängigkeiten zwischen den *Time Stemps* zu modellieren, da der berechnete Gradient durch die Kettenregel ausgelöscht wird.

Abbildung 12: RNN Cell



Quelle: Eigene Darstellung

2.4.5 Long Short Term Memory (LSTM)

Nachfolgend wird ein *Recurrent Neural Network* mit *Long Short Term Memory* Zellen beschrieben und dessen Vorteile zu dem vorher dargestellten normalen *RNN* aufgezeigt. Man könnte sagen, dass die *LSTM* Zelle eine Erweiterung zu dem *RNN* ist, in dem versucht wurde, die Nachteile zu umgehen (*Vanishing* und *Explodig Gradient*). *LSTM* Zellen haben darüber hinaus heutzutage eine besondere Beliebtheit erreicht, da alle modernen Übersetzer und Textgenerierungsprozesse mit diesen Modellen aufgebaut wurden.

Eine *LSTM Cell* Struktur ist in **Abbildung 13** dargestellt. Zwei verschiedene Arten von Transformationen können identifiziert werden: erstens die grünen Elemente, die einfache *Neural Network Layers* mit einer gewissen Anzahl an *Hidden Units* darstellen und zweitens die orangefarbenen Elemente, die die Vektoroperationen repräsentieren.

Die Schlüsselfunktion, die es erlaubt Informationen von einer Zelle zur anderen auf einem einfachem Weg zu transferieren, ist die horizontale obere Linie, die direkt durch die Zelle verläuft. Dies ist der so genannte *Cell State* oder *Long Term Memory*, die mit der *Cell State* aus der vorherigen Zelle beginnt und weiter in die darauf folgende Zelle weitergegeben wird (nach Durchführung von zwei Vektoroperationen).

Die Erschließung des *Cell State* sieht man in **Abbildung 13 a)** und wird wie folgt berechnet:

$$C_t = C_{t-1} * f_t + i_t * \hat{C}_t$$

Erstens wird der *Cell State* aus der vorherigen Zelle C_{t-1} mit dem *Forget Gate* f_t multipliziert, der sich wie folgt erschließen lässt:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$

wobei W_f und b_f entsprechend die Gewichte und der *Bias* des *Forgets* Schritts sind, der als Aktivierungsfunktion eine *Sigmoid Activation* hat. Zweitens wird der Term $i_t * \hat{C}_t$ addiert, der entscheiden wird, welche neuen Informationen zu dem *Cell State* hinzugefügt werden. Diese Elemente werden wie folgt errechnet:

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$

mit W_i als Gewichts Matrix dieses *Layers* und b_i als *Bias* Term.

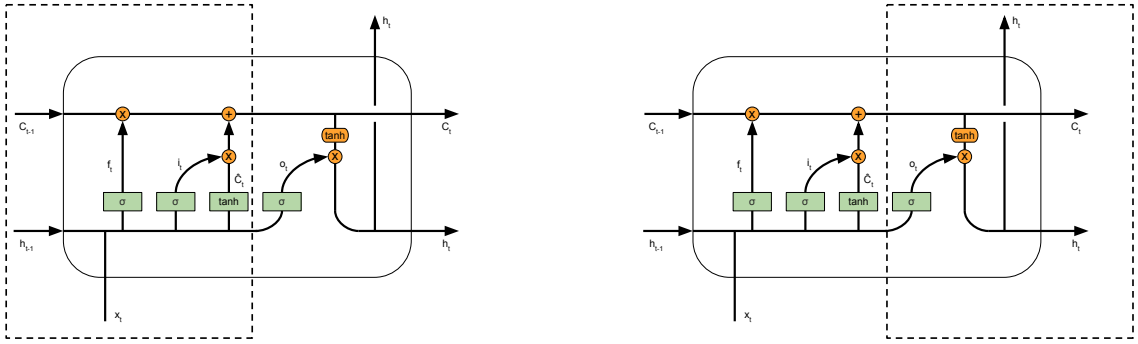
$$\hat{C}_t = tahn(W_C[h_{t-1}, x_t] + b_C)$$

mit entsprechender Gewichte Matrix W_C und *Bias* b_C. Die durchgeführten Transformationen haben folgende Interpretation: Die *Forget* Operation, also die Multiplizierung von C_{t-1} mit f_t sagt aus, wie viel man aus den vorherigen Schritten vergessen sollte. Da f_t das Produkt eines *Layers* mit *Sigmoid* Aktivierung ist, ist es ein Vektor von Zahlen zwischen 0 und 1, bei dem 0 bedeutet, dass diese Information komplett vergessen werden soll und 1, dass die Information im vollem Umfang beibehalten werden muss. Die $\hat{C}_t$ Werte stellen neue Kandidaten von Informationen dar, die in den neuen *Cell State* C_t mit eingebunden werden müssen. Dieser *Layer* wird mit einer *Tanh* Funktion aktiviert, wodurch die Werte in den Bereich -1 bis 1 *gepusht* werden. Diese Kandidaten werden zusätzlich mit Hilfe von i_t skaliert und entscheiden somit, wie die entsprechenden Information im *Cell State geupdated* werden (*Sigmoid* Aktivierung: also Werte von 0 bis 1).

Abbildung 13: LSTM Cell

(a) Forget and Remember

(b) Output



Quelle: Eigene Darstellung (Olah 2015)

In **Abbildung 13 b)** wird zunächst entschieden, wie der *Output* der Zelle aussehen wird. Als erstes folgt ein *Sigmoid Layer* der entscheidet, welcher Teil des *Inputs* ausgegeben wird.

Als nächstes wird der im vorherigem Schritt berechnete *Cell State* durch eine *Tanh* Funktion transformiert (um Werte zwischen -1 und 1 zu erhalten), um ihn im Anschluss mit dem *Output* aus dem *Sigmoid Layer* zu multiplizieren - somit soll erreicht werden, dass nur die Teile ausgeben werden, für die man sich entschieden hat. Mathematisch dargestellt ist der soeben beschriebe Prozess wie folgt zu verstehen:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

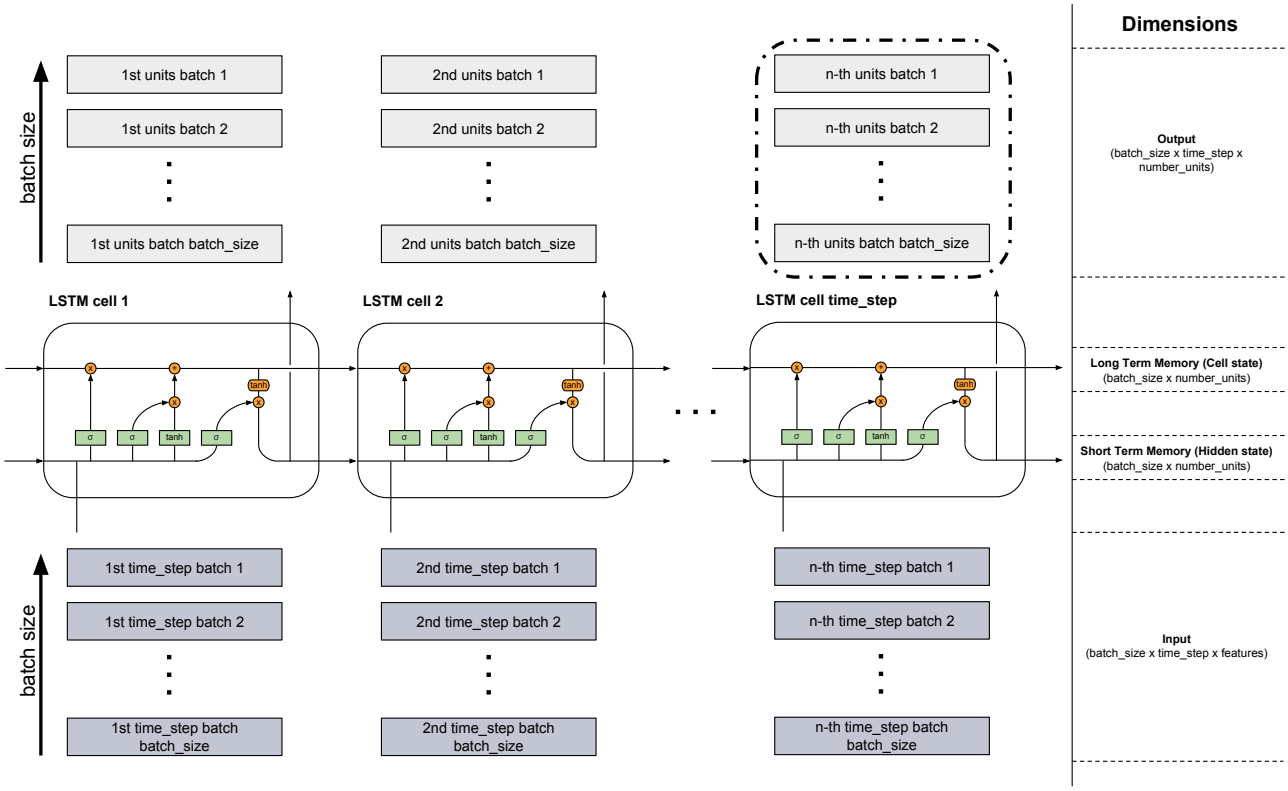
und

$$h_t = o_t * C_t$$

wobei in der ersten Gleichung der *Input* im *Layer* mit den Gewichten W_o und dem *Bias* b_o bearbeitet wird. In der zweiten Gleichung stellt h_t den *Output* und o_t die gewünschten Informationen aus dem *Input* dar. Ein wichtiger Hinweis hier ist, dass der *Output* h_t in zweifacher Ausführung aus der *LSTM* Zelle ausgegeben wird. Einmal wird er als sogenannte *Short Term Memory* an die nächste Zelle geleitet und einmal wird er als tatsächlicher *Output* der aktuellen Zelle zurückgegeben.

Mehrere *LSTM* Zellen bilden ein *LSTM Layer*. Dies wurde in der **Abbildung 14** gezeigt. Da man es meistens mit Daten zu tun hat, die sehr umfangreich sind, ist es nicht möglich, alles an einem Stück in das Modell einfließen zu lassen. Deshalb werden Daten auf kleine Stücke geteilt, als *Batches*, die nacheinander verarbeitet werden, bis der *Batch*, der den letzten Teil enthält, eingelesen wird. Im unteren Teil der **Abbildung 14** sieht man den *Input* (dunkelgrau), bei dem die *Batches* nacheinander, also von *batch 1* bis *Batch batch_size*, eingelesen werden. Die darüber liegenden Zellen *LSTM cell 1* bis *LSTM cell time_step* stellen die vorher beschrieben Zellen des *LSTM* Modells dar. Die Anzahl der Zellen ist gleich der Anzahl der festgelegten *Time Steps*. Wenn man beispielsweise eine Text Sequenz nimmt, die insgesamt 150 Zeichen hat, könnte man sie durch 3 teilen (batch_size 50) und würde je *Batch* eine Sequenz der Länge 50 haben (Anzahl von time_steps in der Untersequenz und somit die Anzahl der *LSTM* Zellen). Wenn man anschließend jedes Zeichen *One-hot encoden* würde, würde jedes Element (dunkelgraue Kästchen des *Inputs*) einen Vektor darstellen, der die Länge des Vokabulars (Anzahl der *Features*) hätte und außer der Position des jeweiligen Buchstabens, an dem eine 1 wäre, nur 0-en hätte. Diese Vektoren würden in den jeweiligen Zellen in die *Neuronal Networks* (grünen Elemente in den Zellen) einfließen und würden ihre Dimension zu der Länge von der Anzahl an *Hidden Units* ändern. Der *Input* hat also die Dimension (batch_size x time_step x features). Die *Long Time Memory (Cell State)* und *Short Time Memory (Hidden State)* haben die gleichen Dimensionen (batch_size x number_units). Die hellgrauen Blöcke, die aus den Zellen entstehen, haben eine andere Dimension, dadurch, dass die Transformationen in den *Neural Networks* (grüne Elemente) mit Hilfe der *Hidden Units* stattgefunden haben (batch_size x time_step x number_units). Der *Output* kann aus jeder Zelle zurückgegeben werden - allerdings ist meistens nur die Information aus dem letzten Block (schwarze Umrandung) relevant (nicht in allen Problemen), da dort alle Informationen aus den früheren *Time Steps* enthalten sind.

Abbildung 14: LSTM Architektur



Quelle: Eigene Darstellung

2.4.6 Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU) ist eine andere Art von *RRNs* die im Vergleich zu *LSTMs* eine deutlich geringere Rechenkraft in Anspruch nimmt. Hier gibt es, wie in **Abbildung 15** zu sehen, anstatt von drei Elementen (wie beim den *LSTM* Zellen), nur zwei. Auf **Grafik a)** wurde das *Reset Gate* und auf **Grafik b)** das *Update Gate* dargestellt. Die Vektoroperationen und *NN* Elemente wurden wie zuvor mit orange bzw. grün markiert. *Reset Gate* entscheidet, wie viel von den vergangenen Informationen von dem Modell vergessen werden sollen:

$$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r)$$

W_r bezeichnet die Gewichte des Netzes, die für die Zusammensetzung von h_{t-1} (Information aus der vorherigen Zelle) und x_t (*Input* zum Zeitpunkt t) zuständig sind. Wie zuvor ist b_r der *Bias* dieses *NN* Elements. Das *Update Gate* entscheidet in dem Modell darüber, wie viele von den vergangenen Informationen (aus früheren Zellen) an die darauf folgenden Zellen übergeben werden. Dies berechnet man wie folgt:

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z)$$

in dem W_z die Gewichte des *Update Gates* mit *Sigmoid* Aktivierung sind und b_z der *Bias* in diesem Netz darstellt. Der *Cell Sate* erschließt sich in diesem Fall aus der Summe des *Inputs*

x_t und des Produkts von dem *Reset Gate* r_t mit der Information aus der vorherigen Zelle h_{t-1}, die dann, durch die nicht lineare *Tanh* Funktion, aktiviert wird. Mathematisch wird das in der folgenden Gleichung dargestellt:

$$C_t = tanh(x_t + r_t \otimes h_{t-1})$$

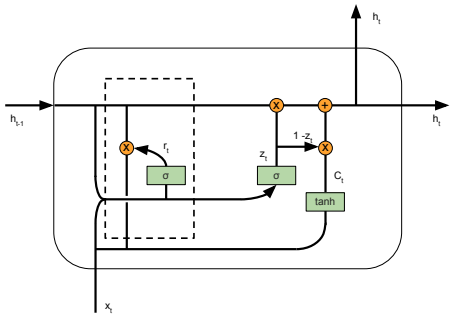
wo $\otimes$ das elementweise Multiplizieren darstellt. Im letzten Schritt wird die neue Information aus der Zelle h_t an Zeitpunkt t berechnet. Dies wird wie folgt gemacht:

$$h_t = z_t \otimes h_{t-1} + (1 - z_t) \otimes C_t$$

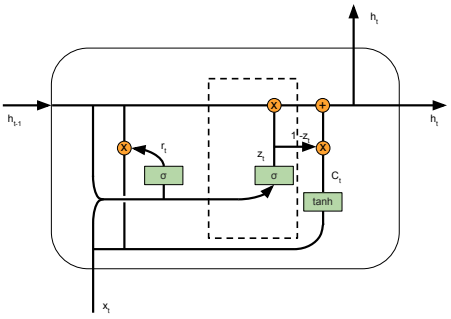
z_t also das *Update Gate* wird elementweise mit der Information aus der letzten Zelle h_{t-1} multipliziert und dann zusammenaddiert mit dem elementweisen Produkt aus $(1 - z_t)$ und dem *Cell State* C_t

Abbildung 15: GRU Cell

(a) Reset Gate



(b) Update Gate



Quelle: Eigene Darstellung (Olah 2015)

3 Analyse

Dieses Kapitel befasst sich damit, wie die im vorherigem Kapitel beschriebenen Methoden auf den gewählten Datensatz angewandt werden. Im ersten Teil wurde die Datenquelle sowie die Charakteristiken der Jahresabschlüsse vorgestellt. Darauf folgend wurde kurz erklärt, wie und mit welchen Hilfsmitteln die Daten gezogen wurden, um anschließend zu zeigen, wie diese Daten aufbereitet worden sind und welche grundlegenden Statistiken für die Textdateien vorliegen. Im zweiten Teil wurde erstens gezeigt, welche Infrastruktur für die darauf folgende Analyse verwendet wurde. Danach wurden die Ergebnisse für unterschiedliche *Parameter Settings* und unterschiedliche Architekturen dargestellt und verglichen. Diese Ergebnisse werden sowohl Charakteristiken der jeweiligen Elemente der Modelle beinhalten sowie Beispiele von generiertem Text aus verschiedenen Epochen. Dies erlaubt es zu beobachten, ob und wie der Lernprozess für die Modelle verlaufen ist.

3.1 Datensatz

Der Datensatz, der verwendet wird, um das vorher beschriebene Problem zu behandeln, muss ausreichend groß sein, da Methoden angewandt werden (*Neuronale Netze - RNN*), die in Hinsicht auf Daten sehr *greedy* sind. Allgemein ist es schwer festzulegen, ab wann man von einem großen Datensatz spricht, aber in der Regel hilft die anwachsende Datenmenge der *Generalisierung* also auch der Verhinderung von *Overfitting* in einem Modell. Ein größerer Datensatz ist allerdings immer mit Aufwand von mehr Rechenkraft verbunden, worauf im späteren Teil des Kapitels noch detaillierter eingegangen wurde.

Jahresberichte sind generell Texte, die über die Tätigkeiten eines Unternehmens im vergangenen Jahr informieren. Die Geschäftsberichte dienen dazu, Aktionären und anderen Parteien einen Einblick in die finanziellen Ergebnisse eines Unternehmen zu geben. Die meisten Rechtsorgane verlangen, dass solche Berichte erstellt und veröffentlicht werden. Ein solcher Jahresbericht wird oft beim Handelsregister der Gesellschaft hinterlegt und muss unter anderen folgende Elemente beinhalten (Oser 2017):

1. Allgemeine Unternehmensinformationen
2. Betriebs- und Finanzübersicht
3. Bericht des Geschäftsführers
4. Informationen zur Unternehmensführung
5. Stellungsnahme des Vorsitzenden
6. Bericht des Wirtschaftsprüfers
7. Jahresabschlüsse, einschließlich:

- Die Bilanz
- Gewinn- und Verlustrechnung
- Cashflow-Rechnung
- Bilanzierungs- und Bewertungsmethoden
- Weitere Merkmale

3.1.1 Daten Quelle

Die verwendeten Daten kommen von der Internet Seite *AnnualReports.com*, die dem Nutzer durch einen kostenlosen Service die Möglichkeit bietet, einfach und schnell auf alle historischen Jahresberichte eines Unternehmens zuzugreifen. Die dort vorhandenen Berichte stellen die größte Ansammlung von Jahresabschlüssen der USA dar. Die gelisteten Unternehmen stellen der Seite jährlich die neusten Abschlüsse zur Verfügung, die als *PDF* oder *HTML* zum *Download* angeboten werden. Der ganze Service ist kostenfrei - nur das Bestellen einer gedruckten Version ist kostenpflichtig. Die Seite ermöglicht es, Investoren, Aktionären und wissenschaftlichen Arbeitern, wie auch Studenten, einen Überblick auf alle relevanten Informationen über bestimmte Unternehmen zu erhalten. Für die geplante Aufgabe der Textgenerierung ist diese Datenbank von besonderem Interesse, da viele Unternehmensinformationen in Textform vorliegen.

3.1.2 Webcrawler

Ein *Webcrawler* erlaubt es, ausgewählte Internet Seiten zu durchsuchen. *Webcraweler* sind Computerprogramme, die in dem *Source Code* einer Seite nach weiteren *URL's* suchen, um im Endeffekt aus allen gesammelten Seiten eine bestimmte Information zu gewinnen. Man beginnt also mit einer oder mehreren *URL's* und arbeitet sich zu anderen verlinkten Seiten vor, bis die gewünschte Information zu finden ist und gespeichert werden kann. Diese Methodik erlaubt es auf einer großen Skala, Daten aus dem *World Wide Web*, ohne das manuelle Durchsuchen mehrerer (oft) hunderter oder tausender Links, zu gewinnen.

In dieser Arbeit wurde die *Python library*, *Scrapy* verwendet. *Scrapy* basiert auf kleinen Programmen, die *spiders* genannt werden. Hierbei handelt es sich um an sich eigenständige *scraper*, die einfachen Regeln folgen, um sich im *World Wide Web* fort zu bewegen. Anhand von sich gegenseitig aufrufenden Funktionen wird die Idee von *don't repeat yourself framework* eingehalten. Diese Eigenschaft macht *Scrapy* zu einem höchst effizienten *Webscraper library*, weshalb sie auch in dieser Arbeit verwendet wurde (Scrapinghub 2008).

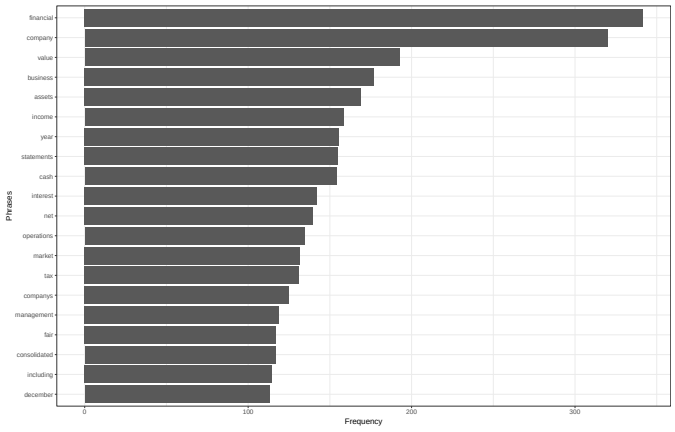
3.1.3 Grundlegende Statistiken

In **Abbildung 16** sind zwei Grafiken erkennbar, die die Häufigkeiten der Phrasen in allen analysierten Texten darstellen, die mit Hilfe der *N-Gram* Methode zerlegt wurden. *N-Grams* ist eine Methode aus der Familie von *Bag-of-words*, die aus einem Text Gruppen von Wörtern (oder Zeichen) mit n oder weniger Wörtern bildet. Ein Beispiel würde wie folgt aussehen:

Satz - “Im Wald stehen Bäume” und die entsprechenden 2-Grams sind [“Im”, “Im Wald”, “Wald”, “Wald stehen”, “stehen”, “stehen Bäume”, “Bäume”]. Dieses Beispiel könnte man auch als *Bag-of-2-grams* bezeichnen. In der **Grafik a)** sieht man ein Balkendiagramm, dass das durchschnittliche Auftreten der häufigsten Phrasen darstellt. In **Grafik b)** ist eine sogenannte *WordCloud* bzw. *PhraseCloud* zu sehen, da nicht nur einzelne Wörter unter Betracht genommen wurden, sondern Phrasen, die aus einem bis drei Wörtern bestehen. Beide Grafiken stellen eine ähnliche Information dar - jedoch in unterschiedlichen Formen. In der rechten Grafik lassen sich schnell die häufigsten Phrasen erkennen, die man dann wiederum leicht in der linken Grafik mit dem dazugehörigen Durchschnitt wiederfinden kann. Da es sich um finanzielle Texte handelt, sind die häufigsten Phrasen auch mit diesem Gebiet verbunden. Die am meisten vorkommenden Wörter sind *financial*, *company* und *value*, mit einem entsprechenden Durchschnitt von 341, 318 und 194. Die einzigen zwei Phrasen, also die Kombination von mehr als einem Wort, die in den Top 100 auftreten, sind *financial statements* und *fair value*, wobei das erstere auch das Thema der zu analysierenden Texte ist und somit einen Sinn ergibt.

Abbildung 16: Phrasen Häufigkeiten

(a) Durchschnittliche Häufigkeiten von Phrasen



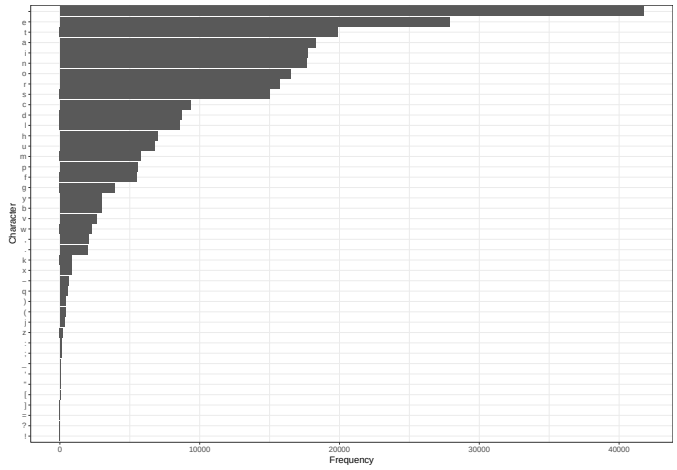
(b) Phrasen Cloud

Quelle: Eigene Darstellung

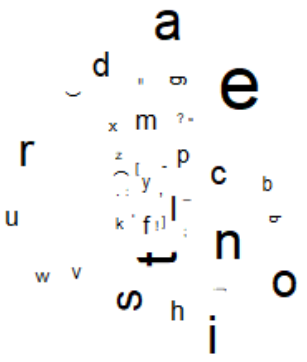
Eine ähnliche Abbildung zu der oben beschriebenen, ist die **Abbildung 17**, welche allerdings nicht die Häufigkeit ganzer Wörter darstellt, sondern die Häufigkeiten einzelner Zeichen. Die Texte wurden in einzelne Buchstaben und Sonderzeichen unterteilt und danach aufsummiert und durch die Anzahl aller Dokumente geteilt, um das durchschnittliche Auftreten einzelner Zeichen zu bestimmen. Da die später verwendeten Methoden auf Zeichen-Basis angewandt werden, sind diese Informationen von großer Bedeutung um im Nachhinein den generierten Text besser verstehen zu können. Das Zeichen, dass im Schnitt am meisten auftritt, ist das Leerzeichen - es tritt im Schnitt ~42 tausend mal auf, was um 10 Tausend mehr ist, als der häufigste Buchstabe *e* und mehr als doppelt so viel, wie der zweithäufigste Buchstabe *t*. Diese drei Zeichen sind somit die am häufigsten auftretenden Zeichen und sind in allen Texten am zahlreichsten repräsentiert.

Abbildung 17: Zeichen Häufigkeiten

(a) Durchschnittliche Häufigkeiten von Zeichen



(b) Zeichen Cloud

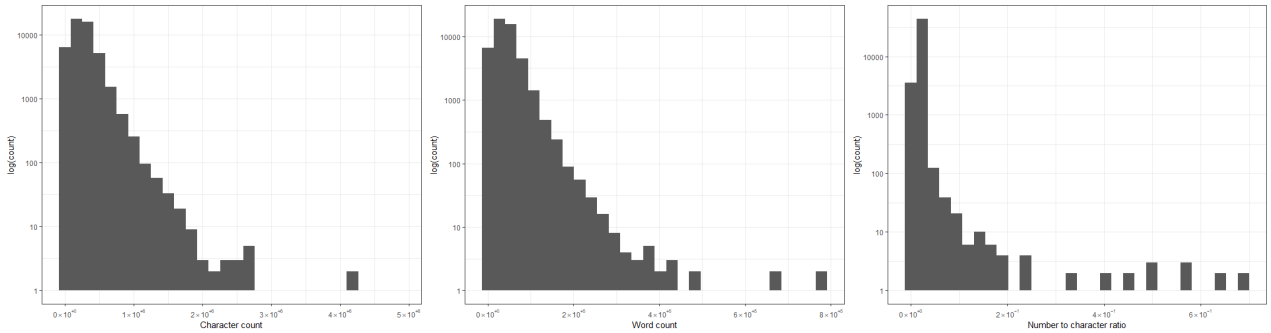


Quelle: Eigene Darstellung

Abbildung 18 enthält Informationen über die *Metadaten* der analysierten Texte. In **Grafik a)** wurde die Frequenz der Zeichenanzahl in den verschiedenen Dokumenten als Histogramm dargestellt. Die Streubreite liegt zwischen 0 und 4.5 Millionen Zeichen, wobei die Hauptmasse der Dokumente eine Anzahl von 0 bis 2 Million Zeichen besitzt. Die Dokumente, die in den höheren Bereichen der Zeichenanzahl liegen (4 Millionen), treten weniger häufig auf und können damit als Ausreißer anerkannt werden, die jedoch für die Durchführung der Analyse kein Hindernis darstellen, da diese meist von größeren Unternehmen erstellt wurden und dadurch auch einen deutlich größeren Umfang haben als Dokumente, die von kleineren Unternehmen verfasst wurden. Beunruhigend werden Dokumente sein, die eine deutlich zu kleine Anzahl an Zeichen haben - dies würde nämlich darauf hindeuten, dass ein Fehler beim *Downloaden* oder Verarbeiten der Dokumente aufgetreten ist. Die Spitze des Histogramms liegt bei 250 Tausend Zeichen. In der zweiten **Grafik b)** sieht man, dass die Häufigkeitsverteilung der Anzahl der Wörter in den unterschiedlichen Dokumenten ähnlich aussieht, wie die der Zeichenanzahl, was bedeutet, dass voraussichtlich eine größere Zeichenanzahl auch mit einer größeren Anzahl von Wörtern zusammenhängt. Die Streubreite dieser Variable reicht von 0 bis 800 Tausend Wörter und die Spitze liegt bei 40 Tausend Wörtern. Die Hauptmasse der Texte befindet sich zwischen 0 und 400 Tausend Wörtern. Das dritte Histogramm, welches auf **Grafik c)** zu sehen ist, stellt die Verhältnisse von Zahlen zu allen Zeichen in den jeweiligen Texten dar. Dies wurde berechnet, um festzustellen, dass man nicht nur Texte hat, die hauptsächlich Jahresbilanzen zeigen, sondern Texte, die entsprechend viel geschriebenen Text enthalten. Dies ist in dieser Arbeit von besonderem Interesse. Hierbei liegt die Streubreite zwischen 0 und 0.6 (60%). Die Hauptmasse befindet sich auf dem Intervall von 0 bis 0,2 (20%). Es handelt sich also überwiegend um Texte, in denen Zahlen weniger als 20% ausmachen und somit mehr geschriebenen Text als Jahreszahlen etc. enthalten. Im Nachhinein werden Texte bei der Transformation von *PDF* Dateien in Textdokumente, Absätze, bei denen dieser Parameter größer als 0.2 (20%) ist, gelöscht um somit die darin enthaltenen Tabellen aus den reinen Textdateien loszuwerden.

Abbildung 18: Histogramme der Metadaten

(a) Zeichenanzahl **(b)** Wörteranzahl **(c)** Zahlen zu Zeichen Ratio



Quelle: Eigene Darstellung

In **Tabelle 4** wurden die deskriptiven Statistiken aller Texte zusammengefasst. Es wurden die Lagemaße für die Zahlen zu Zeichenverhältnissen, Zeichenanzahl und Wörteranzahl berechnet. Die analysierten Texte haben im Schnitt ein Zahlen- zum Zeichenverhältnis von 0.018 (also 1.8%), wobei die Texte mit jeweils dem kleinsten und größten Verhältnis, entsprechend 6% und 69%, haben. Der kürzeste Text, wenn es um Wörter geht, hat auch die kleinste Anzahl an Zeichen (218 Wörter und 9221 Zeichen). Dieser Jahresabschluss enthält nur eine kurze Beschreibung des vergangenen Fiskaljahres und einleitende Wörter des Geschäftsführers. Im Durchschnitt liegt die Zahl der Wörter bei 42282 und die Anzahl der Zeichen bei fast 273 Tausend. Der längste Jahresabschluss hat über 778 Tausend Wörter und der Text mit der höchsten Anzahl von Zeichen hat mehr als 4.8 Millionen Zeichen. Wenn man die durchschnittliche Anzahl der Zeichen durch die durchschnittliche Zahl der Wörter teilt, dann erhält man Wörter der durchschnittlichen Zeichenlänge ~6 haben.

Tabelle 4: Grundlegende Statistiken

	Ratio	**Characters**	**Words**
Min.	0.06	9221	218
1st Qu.	0.015	146537	22764
Median	0.017	248540	38322
Mean	0.018	273134	42282
3rd Qu.	0.020	357018	55095
Max.	0.689	4858544	778254

Quelle: Eigene Darstellung

3.1.4 Datenverarbeitung

Die im vorherigen Schritt erhaltenen *PDF's* sind in einer solcher Form nicht zu gebrauchen. Zur Verarbeitung der Dokumente und zur Durchführung des Experiments muss der Text aus den *PDF's* extrahiert und gesäubert werden. Dies wurde mit Hilfe eines R-Skripts durchgeführt, welches reinen Text aus den Dokumenten zieht und als *.txt* Datei abspeichert. In diesem Verfahren wurde auch das Verhältnis der Anzahl von Zahlen zu allen Zeichen, in den jeweiligen Sätzen, berechnet. Sätze, in denen diese Verhältnisse 0.2, also 20% überstieg, werden aus dem

extrahierten Text ausgeschlossen, da die Chance, dass ein solcher Satz eine extrahierte Tabelle darstellt, sehr hoch ist. Zur Erinnerung: Tabellen mit Kennzahlen über Unternehmen sind in diesem Fall irrelevant und können ohne Probleme ignoriert werden. Im Nachhinein wurden auch noch alle Zahlen aus den Texten ausgeschlossen, jedoch ist der vorherige Schritt trotzdem notwendig, um den entsprechenden Text, der in den Tabellen enthalten war, zu finden und im zweiten Schritt zu löschen.

Die gesammelten *PDF's* sind außerdem in verschiedenen Programmen und Versionen verfasst worden, weil sie erstens von unterschiedlichen Unternehmen geschrieben wurden und zweitens aus einem Zeitraum von mehr als 10 Jahren stammen. Das führt dazu, dass die Dokumente unterschiedlich gut aufgelöst und verarbeitet werden können. Hinzu kommt, dass manche Zeichen schlecht entschlüsselt wurden oder einfach einem anderen Zeichen zugeschrieben worden sind. Das führt dazu, dass viele merkwürdige Zeichen im Text auftreten, an Stellen, an denen in den *PDF's* Zeichen, Bilder, Seitenaufteilungen, etc., aufgetreten sind. Diese Sonderzeichen oder schlecht entschlüsselten Zeichen sind nicht von Interesse, wenn es um das Erlernen von geschriebenen finanziellen Texten geht, weshalb die Zeichen und Spezialzeichen auf die folgenden begrenzt wurden: ', -, , !, \, (,), „ ., :, ;, ?, [,], _, =, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z. Darüber hinaus wurden alle Buchstaben, die in den Texten als großgeschriebene Buchstaben auftreten, zu deren entsprechenden Darstellung in Kleinschrift transformiert, damit das verwendete Vokabular und damit die interessierenden Klassen des Modells entsprechend klein gehalten wird.

Zusammenfassend:

1. *.pdf* Dateien wurden in *.txt* Dateien transformiert.
2. Tabellen wurden aus dem Text anhand der Zahlen zu Zeichenverhältnissen entfernt.
3. Großbuchstaben wurden zu Kleinbuchstaben transformiert.
4. das Vokabular wurde auf folgende Zeichen begrenzt: ', -, , !, \, (,), „ ., :, ;, ?, [,], _, =, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z.
5. Zahlen wurden aus den Texten ausgeschlossen.

3.1.5 Datenaufbereitung (Inputspace)

Nach der Datenverarbeitung ist es noch notwendig, die Daten in die entsprechende Form für *Deep Learning* Probleme zu bringen. Hierfür wird das theoretische Wissen aus **Kapitel 2** über *Tensoren* und das *One-hot Encoding* benötigt. Da die Aufgabe darin bestehen wird, ein bestimmtes Zeichen anhand einer vorausgehenden Zeichensequenz vorauszusagen, muss der Text in entsprechende Komponenten zerteilt werden:

1. Zeichen Sequenz der Länge *maxlen*, die im Folgenden *input sequence* genannt wird.

2. Das auf die *input sequence* folgende Zeichen (*following character*).

Das folgende Beispiel dient der Visualisierung des Vorgangs:
Ein Ausschnitt aus dem Jahresbericht der Firma Kelloggs aus dem Jahr 2016

"we have experienced, and expect to continue to experience, intense competition for sales of all of our principal products in our major product categories, both domestically and internationally." (Kellogg 2016)

Der oben zitierte Ausschnitt wurde nach den Regeln aus **Unterkapitel 3.1.4** verarbeitet und enthält somit nur auserwählte Zeichen. In der **Tabelle 5** wird dargestellt, wie der Text in Untersequenzen aufgeteilt wird. Es werden Sequenzen gebildet, die eine maximale Länge des Parameter *maxlen* sind (in diesem Beispiel ist *maxlen* 35). Das bedeutet, dass der Text in gleichlange Abschnitte der Länge von 35 Zeichen geteilt wird. Die Aufgabe, die zu erlernen ist, besteht darin, den 36-sten Buchstaben korrekt vorauszusagen. Es werden nicht alle möglichen Zerteilungen des Textes in die Analyse mit aufgenommen. Darüber entscheidet ein Verschiebungs-Parameter, der weiterhin *skipp* genannt wird. In diesem Beispiel beträgt *skipp* 3 und bedeutet, dass nach jeder Aufteilung in Sequenzen der Länge *maxlen*, die darauf folgende Sequenz nicht einen Buchstaben danach beginnt, sondern mit zwei. Diese Vorgehensweise verhindert, dass während des Lernprozesses eines Algorithmus genaue Strukturen des Textes erlernt werden können. Zur Erinnerung: es ist nicht das Ziel, Text aus vorherigen Dokumenten abzubilden, sondern die Erlernung textgenerierender Prozesse.

Tabelle 5: Text Aufbereitung

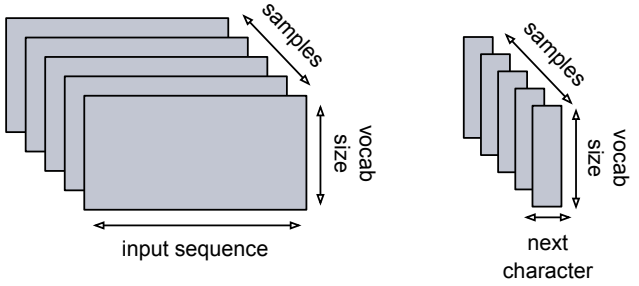
sample	input sequence	following character
1	we have experienced, and expect to	c
2	have experienced, and expect to con	t
3	e experienced, and expect to contin	u
4	xperienced, and expect to continue	t
5	rienced, and expect to continue to	e
6	nced, and expect to continue to exp	e
...	...	...
52	es, both domestically and internati	o
53	both domestically and internationa	l
54	th domestically and internationally	.

Quelle: Eigene Darstellung

Wenn man im Nachhinein die im vorherigen Schritt erhaltenen Sequenzen so entschlüsselt, wie es in dem **Unterkapitel 2.1.3** (*One-hot Encoding*) erläutert wird, erhält man eine Ansammlung von Matrizen (2D *Tensoren*), die bei Aufstockung einen 3 D *Tensor* bilden. Das gleiche geschieht, wenn man die *Following Characters* entschlüsselt. Der Effekt dieser Prozedur ist in **Abbildung 19** dargestellt. Auf der linken Seite sieht man die aufgestockten Matrizen der *Input Sequence*, die die Dimension *(maxlen x vocab size)* haben. Das Tupel dieser Matrizen ist dann ein *Data Qube* (3 D *Tensor*), der zusätzlich zu den zwei Dimensionen noch die Dimension

beinhaltet, die die Anzahl der *Samples* repräsentiert. Auf der rechten Seite der Abbildung sieht man auch einen 3 D Tensor, allerdings für die *Following Characters*. Die *Sample* Anzahl und das *Vocab Size* bleibt hierbei gleich, nur das *maxlen* ist in diesem Fall 1, da es sich um einen Buchstaben handelt, den man voraussagen will.

Abbildung 19: 3 D Tensor als Input Space



Quelle: Eigene Darstellung

3.2 Deep Learning Infrastruktur und Ergebnisse

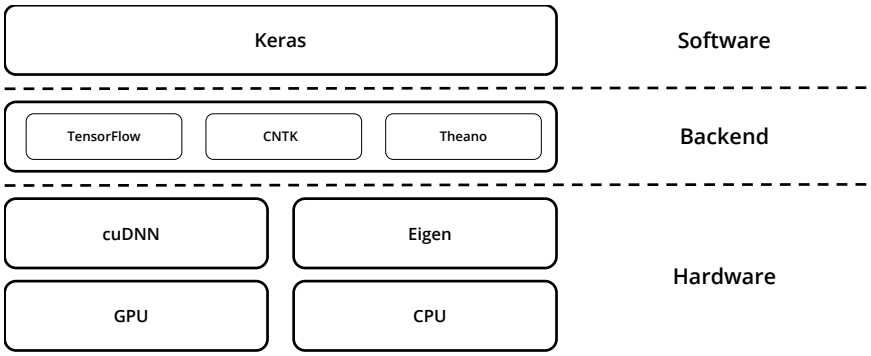
In diesem Unterkapitel wird zuerst die Hard- und Software, die verwendet wurden, dargestellt. Hierzu wurden *Keras* und *TensorFlow* verwendet (**Unterkapitel 3.2.1**). Im zweiten Schritt wurden die Ergebnisse der Analyse gezeigt, wobei genauer auf die Verteilung der Werte im Neuronalen Netz sowie die Entwicklung des Verlustes und der *Accuracy* für verschiedene Parametersettings eingegangen worden ist (Ergebnisse für alle Modelle gibt es im **Anhang**). Darüber hinaus wurde auch das erlernte *Embedding* für das beste Parametersetting dargestellt. Die Grafiken wurden teilweise in dem *R package ggplot* und teilweise in *TensorFlow* vorbereitet.

3.2.1 *Keras* und *TensorFlow*

Zur Durchführung der Analyse wird *Tensorflow* als *back-end* und *Keras* als *Interface* in *R* verwendet. *Keras* ermöglicht es, relativ komplexe Modelle in einer übersichtlichen Form zusammenzusetzen und mit dem gleichen *Code* die Analyse auf einem CPU oder GPU anzuwenden. Das vereinfacht es, Prototypen von Modellen zu erstellen und zu testen, um im Anschluss den funktionierenden Prozess auf einem entsprechend leistungsstarken Gerät auszuführen. Darüber hinaus sind in *Keras* auch Netzarchitekturen, wie zum Beispiel *RNNs* oder *Convolutional Neural Networks* für *Computer Vision* implementiert. *Keras* wurde anfangs mit dem Gedanken entwickelt, Forschern die Möglichkeit zu bieten, schnelle *Deep Learning* Experimente durchzuführen und zu testen (Francois Chollet 2018). Heutzutage ist es eines der meist verbreitetsten *Frameworks*, um die vorher genannten Probleme zu lösen. Zusammenfassend ist *Keras* eine *high-level library*, mit derer Hilfe man separate Blöcke (die unterschiedlichen Aufgaben nachkommen)

zusammensetzen kann, um komplexe *Deep Learning* Strukturen zu entwickeln (Chollet et al. 2015). *Tensorflow* hingegen ist eine *open source* Software, die es erlaubt, leistungsstarke nummerische Berechnungen auf unterschiedlichen Prozessoren (CPU oder GPU) durchzuführen (Francois Chollet 2018). Es ist somit eine *back-end* Lösung für *low-level Tensor* Operationen, die es *Keras* ermöglicht, effizient und schnell Berechnungen durchzuführen. *Tensorflow* ist nicht die einzige *back-end* Lösung, welche mit *Keras* kombiniert werden kann, da *Keras* das Problem modular angeht und es damit ermöglicht, verschiedene *tensor-basierte back-end* Lösungen zu integrieren (siehe **Abbildung 20**). *TensorFlow* wiederum ruft *Eigen* bei einem CPU und *cuDNN* bei einem GPU auf. *Eigen* und *cuDNN* stellen die Parallel Computing Plattformen für die jeweiligen Prozessoren dar.

Abbildung 20: Hardware und Software



Quelle: Eigene Darstellung (Francois Chollet 2018)

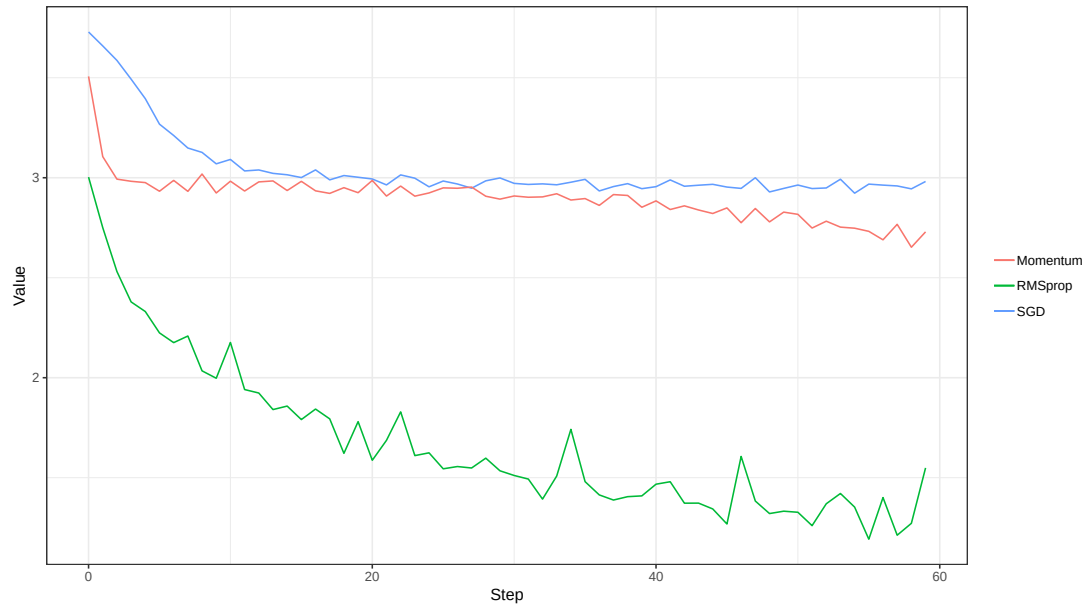
3.2.2 Ergebnisse

In **Tabelle 6** (siehe **Anhang**) wurden die Modelle mit ihren dazugehörigen *Parametersettings* angegeben. Darüber hinaus wurden auch die Metriken für die Iteration (Epoche) dargestellt, in denen der niedrigste Verlust für die jeweiligen *Settings* berechnet wurde. Es wurden sowohl unterschiedliche Parameter des Netzes ausprobiert: die Anzahl der *Hidden Units*, die maximale Länge der *Input* Sequenzen, wie auch verschiedene Optimierungsalgorithmen, die in dieser Arbeit vorgestellt wurden.

In den folgenden Abbildungen wurden die für die in **Tabelle 6** vorgestellten Architekturen, die entsprechenden Metriken grafisch über die Iterationen (Epochen - ein vollständiger Durchlauf aller *Input* Daten) dargestellt. Die Metriken, die für diese Arbeit von größtem Interesse sind, sind erstens der Verlust und zweitens die *Accuracy*. In erster Linie will man den Verlust minimieren und die *Accuracy* wurde als extra Information dargestellt. Dies ist deswegen der Fall, weil nicht das genaue Voraussagen des nächsten Zeichens von Interesse ist (dies würde zur genauen Abbildung von historischen Texten führen), sondern die Erlernung von Regeln für einen gewissen Schreibstil. Der Trainings-Datensatz stellt nur einen Bruchteil aller Jahresabschlüsse dar. Bessere Ergebnisse könnte man durch die Skalierung auf einen größeren Teil des Datensatzes erreichen. Man müsste allerdings eine leistungsstärkere Maschine verwenden, um eine akzeptierbare Lernzeit beizubehalten. Die Ergebnisse für alle Architekturen befinden sich im Anhang.

Abbildung 21 stellt die ersten drei getesteten Modelle dar. Diese haben die gleichen Parameter, bis auf den Optimierungsalgorithmus, der die Verlustfunktion minimieren soll. Diese Algorithmen wurden in **Unterkapitel 2.2** vorgestellt und detailliert beschrieben. Insgesamt wurden für die jeweiligen Modelle 60 Epochen durchgeführt, auf denen man beobachten kann, wie sich der gesamte Verlust über die Zeit entwickelt. Am schlechtesten schneidet *Stochastic Gradient Descent (SGD)* ab (model_1 - blaue Linie) - er ist von der ersten bis zu letzten Epoche mit den Verlustwerten über den beiden anderen und scheint bei Epoche 10 zu konvergieren und nicht mehr zu lernen. Nicht viel besser ist der *SGD mit Momentum* (model_2 - rote Linie), da er sich ähnlich wie der normale *SGD* verhält. Hierbei verringert sich der Verlust im Gegenteil zum vorherigen Algorithmus nach Epoche 40 wieder und könnte bei einer größeren Anzahl von Iterationen zu besseren Ergebnissen führen, als in den ersten 60 Epochen. Der Algorithmus, der sich am besten entwickelt, ist der *RMSprop* (model_3 - grüne Linie), der sowohl nach der ersten Epoche den niedrigsten Verlust hat und auch am schnellsten den Verlust über die Epochen minimiert.

Abbildung 21: Verlust von Modellen mit verschieden Optimierungsalgorithmen

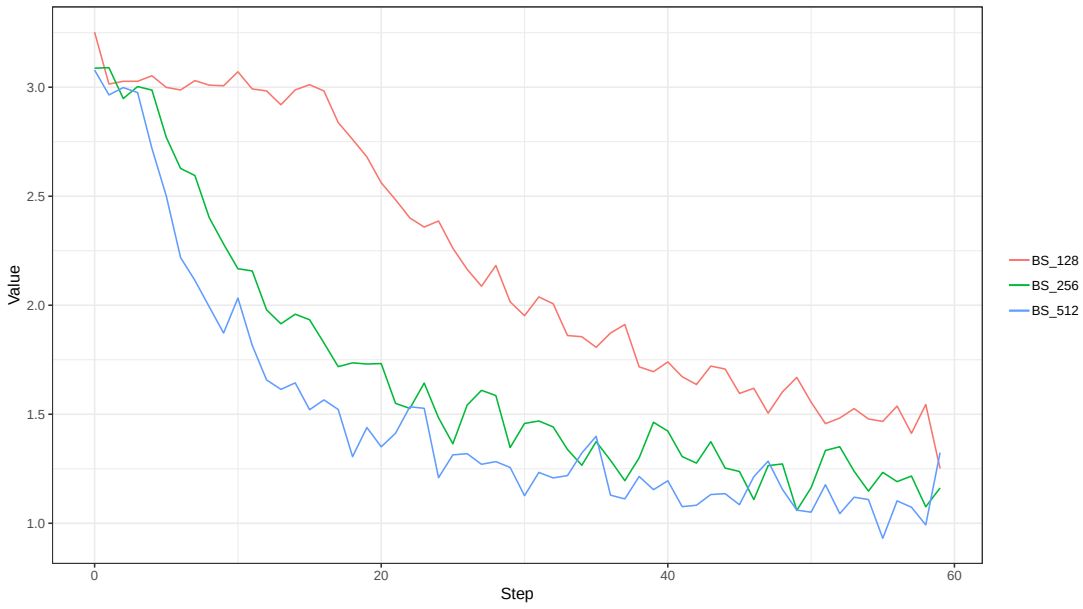


Quelle: Eigene Darstellung

In **Abbildung 22** sieht man wieder drei Modelle. Diesmal haben alle drei den gleichen Optimierungsalgorithmus und zwar den *RMSprop*, da der zu den besten Ergebnissen im vorherigen Fall geführt hat. Diese drei Modelle unterscheiden sich diesmal durch die *Batch Size* (genauer: *Mini Batch Size*). Das Modell mit der *Batch Size* 512 (model_6 - blaue Linie) schneidet am besten ab. Dies könnte deswegen der Fall sein, dass das Modell mit einer solchen Anzahl an *Hidden Units* in der Lage ist, komplexe Strukturen (Zusammenhänge) aus den Texten zu lernen. Das Modell mit *Batch Size* 128 (model_4 - rote Linie) fängt erst ab Epoche 18 an schneller den Verlust zu minimieren und nähert sich sogar in den letzten Epochen dem Wert des Verlustes von den Modellen mit *Batch Size* 256 (model_256 - grüne Linie) und 512. Das Modell mit

Batch Size 256 ist über alle Iterationen leicht schlechter als das komplexere Modell. Im weiteren Schritt wird erst die *Batch Size* 256 genommen (da diese weniger Parameter zum trainieren hat) und später wird noch das komplexere Modell mit 512 *Hidden Units* in Kombination mit anderen Parametern ausprobiert.

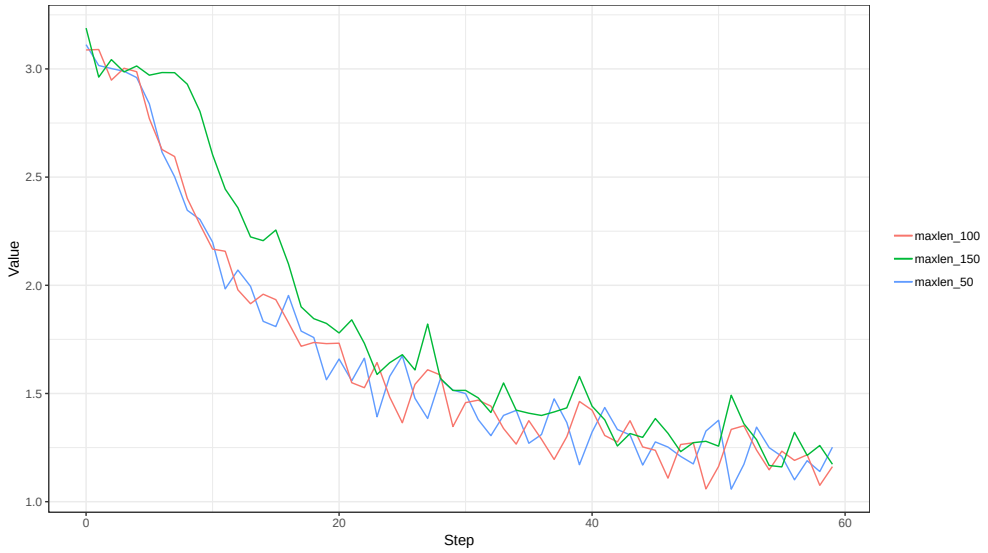
Abbildung 22: Verlust von Modellen mit verschieden Batch Size Parametern



Quelle: Eigene Darstellung

Als nächstes wurden verschiedene Werte für den *maxlen* Parameter getestet. Drei Werte: 50 (model_7 - blaue Linie), 100 (model_5 - rote Linie) und 150 (model_8 - grüne Linie). **Abbildung 23** stellt die Entwicklung des Verlustes über die 60 Epochen für diese drei Modelle dar. Man kann erkennen, dass es keine großen Unterschiede in Abhängigkeit des *maxlen* Parameter gibt. Der einzige Unterschied ist, dass man eine gewisse Verschiebung in den Ergebnissen des Verlustes für das Modell mit *maxlen* 150 (grüne Linie) erkennt, was dadurch zustande kommt, dass dieses Modell eine größere Anzahl an Parametern hat und deswegen schwerer zu trainieren war. Im Endeffekt haben aber alle drei Modelle ähnliche Ergebnisse und deswegen wäre es egal mit welchem Wert des Parameters man fortfahren möchte. Allerdings ist es schwerer, Abhängigkeiten im Text zu lernen ,die länger sind als die maximale Länge der Sequenz, weswegen man sich im weiteren auf den Wert 150 für den *maxlen* entscheidet, um eine längere Abhängigkeit modellieren zu können.

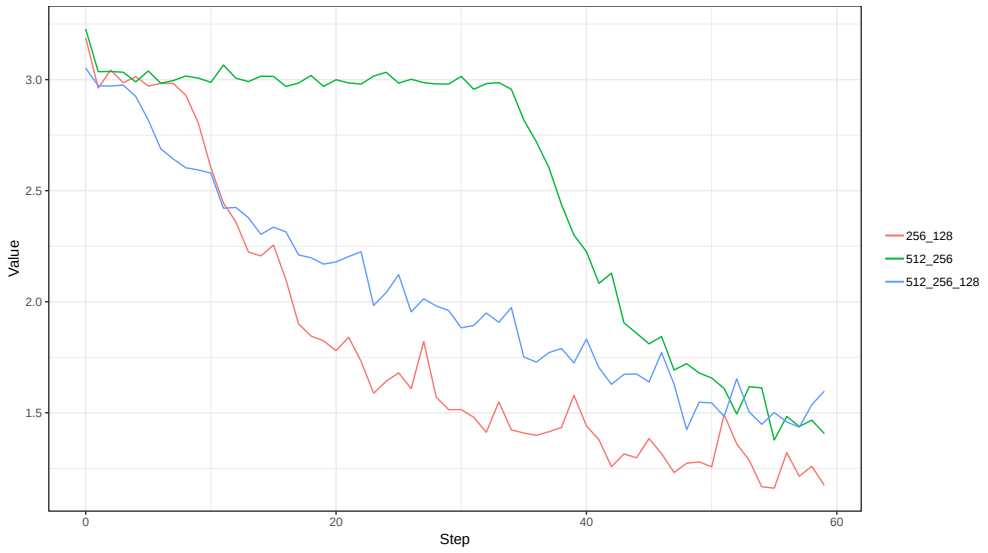
Abbildung 23: Verlust von Modellen mit verschieden maxlen Parametern



Quelle: Eigene Darstellung

Die folgenden drei Modelle zeigen unterschiedliche Grade der Komplexität der Architekturen der Neuronalen Netze. Das einfachste Modell mit zwei *Layern* (*Hidden Units* in *Layer 1* 256 und *Layer 2* 128) hat bis auf die ersten paar Epochen den niedrigsten Verlust (model_8 - rote Linie). Das zweite Modell mit zwei *Layern*, aber mit einer größeren Anzahl an *Units* (512 und 256), ist anfangs zu komplex und das Modell schafft es erst nach *Epoche* 30 den Verlust zu minimieren(model_9 - grüne Linie). Bei *Epoche* 60 ist dieses Modell aber fast gleich auf mit den beiden anderen Modellen. Das letzte Modell, dass dieses mal drei *Layer* hat (512, 256 und 128), wurde mit einer niedrigeren Lernrate initialisiert, um den Verlust früher minimieren zu können (model_13 - blaue Linie). Der Verlust wird allerdings sehr langsam minimiert, da die Lernrate am Anfang kleiner war.

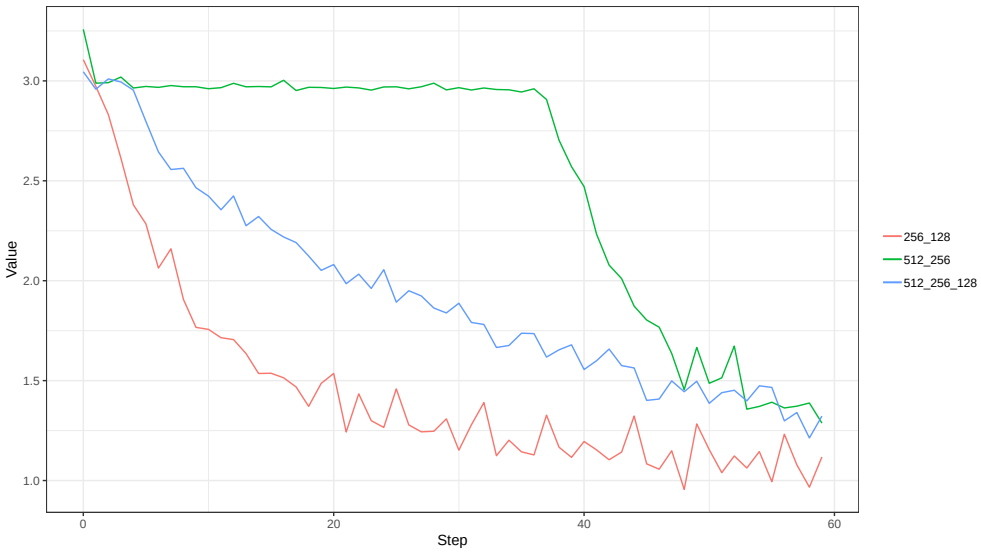
Abbildung 24: Verlust von Modellen mit verschieden Komplexitätsgraden (Batch Size 256)



Quelle: Eigene Darstellung

Abbildung 25 zeigt die gleichen Modelle wie in der vorherigen Abbildung - nur dieses mal mit einer größeren *Batch Size* - 512. Man sieht, dass das Modell mit zwei *Layern* (512 *Hidden Units* und 256 *Hidden Units* - model_11 - grüne Linie) anfangs wieder zu komplex war, wodurch sich der Verlust bis Epoche 37 nicht verändert hat. Danach ist der Sprung in der Minimierung des Verlustes jedoch groß und er passt sich den der zwei anderen Modelle an. Das Modell mit drei *Layern* (512 *Hidden Units*, 256 *Hidden Units* und 128 *Hidden Units* - model_12 - blaue Linie) hingegen, wurde so wie im Fall des Modells mit *Batch Size* 256, mit einer niedrigeren Lernrate initialisiert, wodurch die Minimierung des Verlustes schneller beginnen konnte. Das einfachste Modell (256 *Hidden Units* und 128 *Hidden Units* (model_10 - rote Linie) hat über alle 60 Epochen den kleinsten Wert des Verlustes erzielt - es sieht jedoch aus, als ob der Wert konvergieren würde und deswegen besteht der Verdacht, dass die beiden komplexeren Modelle bei einer höheren Anzahl von Epochen den Verlust mehr minimieren könnten.

Abbildung 25: Verlust von Modellen mit verschieden Komplexitätsgraden (Batch Size 512)

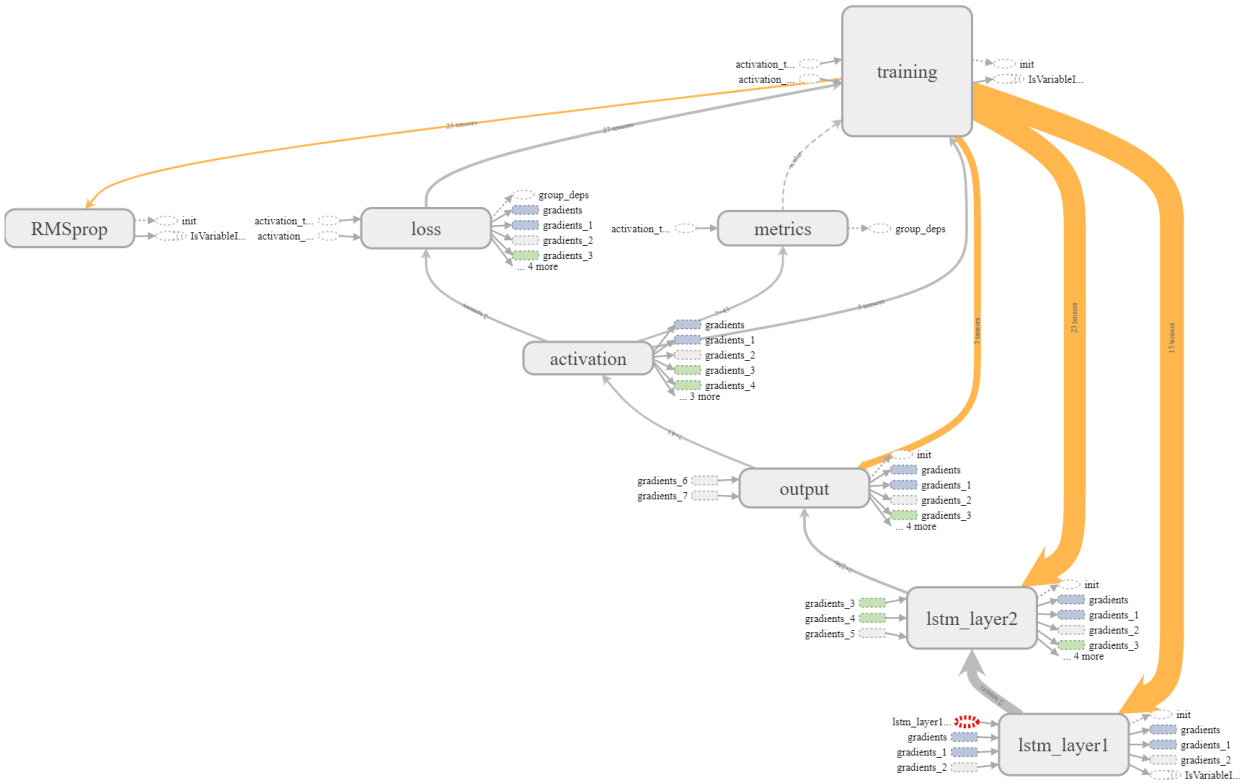


Quelle: Eigene Darstellung

Die folgende **Abbildung 26** (Modell mit: zwei *LSTM Layern*, RMSprop und *Softmax* Aktivierung) zeigt ein Beispiel eines Aufbaus von einem der Modelle (model_11). Man sieht die Elemente, die in den vorherigen Kapiteln beschrieben wurden: *Layers*, *Activations*, Verlustfunktion, Optimierungsalgorithmus und die berechneten Metriken. Man sollte diese Grafik von unten nach oben lesen. Zuerst kommt *lstm_layer1* dann *lstm_layer2*. Diese stellen die jeweiligen *LSTM Layers* im Netzwerk dar und beinhalten alle *LSTM Cells* mit den dazugehörigen Elementen. Als nächstes folgt der *Dense Layer* (eine lineare Transformation, bei der jeder *Input* mit jedem *Output* über ein Gewicht verbunden wird) und die Aktivierungsfunktion, die in diesem Fall die *Softmax* darstellt, und für jeden *Output* (Größe des Vokabular - 43) eine sich zu 1 aufsummierende Wahrscheinlichkeit des Auftretens berechnet. In *Metrics* werden die unterschiedlichen Metriken berechnet, an denen man interessiert ist und im *Loss* Element (Verlust), wird der Wert des Verlustes, der minimiert werden soll, berechnet. Das *Training* Element stellt den Lernprozess dar, der alle erhaltenen Informationen aufnimmt (zusammen mit dem Opti-

mierungsalgorithmus und seinen Parametern) und die entsprechenden Gradienten berechnet sowie die Gewichte im ganzen Netzwerk *Updated*.

Abbildung 26: Modell mit id model_11



Quelle: Eigene Darstellung

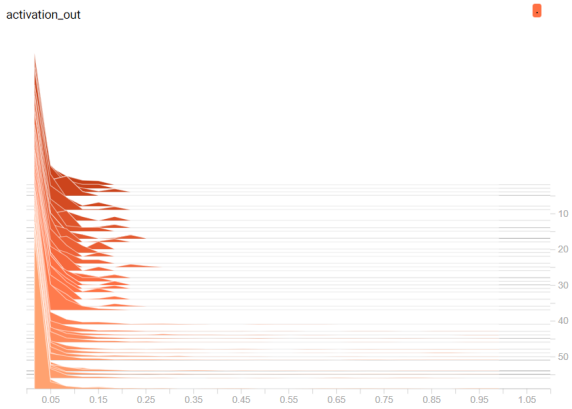
In den folgenden Abbildungen sieht man Histogramme der jeweiligen Elemente des Neuronalen Netzes. Dank ihnen kann man erkennen, ob und wie sich die Werte über die aufeinander folgenden Iterationen verändern. Die Histogramme werden in zwei verschiedenen Arten vorgestellt: Erstens ein 3D *Plot*, der die Histogramme über die Iterationen anzeigt und zweitens ein 2D *Plot*, der die Verteilung des entsprechenden 3D *Plots* darstellt. Die Histogramme der Merkmale aus den ersten Epochen sind in den jeweiligen Grafiken ganz hinten und sind in helleren orange markiert. Umso weiter man sich nach vorne bewegt, desto spätere Epochen werden sichtbar und umso roter wird der Farbton des Histogramms (das vorderste Histogramm stellt also die letzte Epoche des Lernprozesses dar). Zwei Merkmale der Histogramme sind über den Zeitraum verfolgbar: Die Spitze, die den Mittelwert repräsentiert und die Varianz der Werte des jeweiligen Elements. Man kann also beobachten, wie sich die Werte über die Iterationen verändern. Falls man keine Verschiebung des Mittelwerts und keine Veränderung in der Varianz der Werte sieht, bedeutet das, dass das Modell im Trainingsprozess nichts gelernt hat. Die Elemente des Neuronalen Netzes. die in diesen Grafiken dargestellt wurden, sind: der *Output* aus den jeweiligen *LSTM Layern*, der *Output* des letzten *Fully connected Layers* (*Dense Layer*) und der *Output* der *Softmax* Aktivierung. All diese Elemente ermöglichen es zu beobachten, ob irgendwelche der vorher beschriebenen Probleme aufgetreten sind (*Exploding* oder *Vanishing*

Gradients). Oft werden diese Darstellungen als Histogramme und Verteilungen zum *Debugging* der Architekturen verwendet.

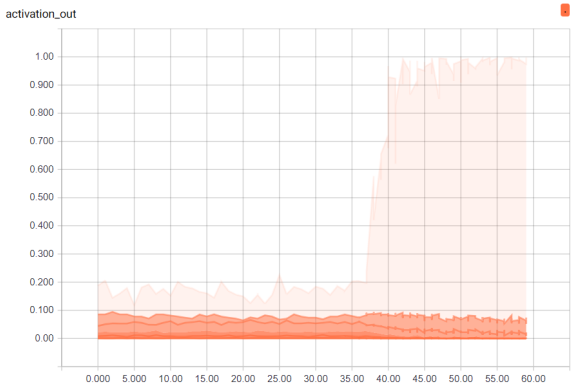
Abbildung 27 stellt insgesamt 8 Grafiken dar (für Modell model_11). Man sollte die Abbildung so betrachten, dass auf der linken Seite sich das 3D Histogramm befindet und die dazugehörige Verteilung auf der rechten Seite. Von unten gesehen kann man also die Werte des *RNN* über die jeweiligen *Layer* bis zum *Output* und die Aktivierung beobachten. Zuerst kommt also *LSTM Layer I* dann *LSTM Layer 2* und so weiter. Hier wurden die Elemente nur für ein Modell betrachtet, da die Beschreibung und Darstellung dieser Grafiken platz- und zeitaufwendig sind (die Histogramme und Verteilungen für alle Modelle befinden sich im **Anhang**). In **Grafik g) & h)** sieht man also ein Histogramm und die Verteilung der Werte, die aus dem ersten *LSTM Layer* kommen. In **g)** sind auf der y Achse die Epochen von 1 bis 60 aufgezeigt und auf der x Achse die entsprechenden Werte. Die z Achse, also die, die den 3D Effekt ermöglicht, stellt die Häufigkeiten eines bestimmten Wertes dar (Histogramm). Dies bedeutet, dass wenn man nur einen Ausschnitt von y betrachtet, erhält man ein 2D Histogramm der *Oputput* Werte für eine bestimmte Epoche. Die Werte, die der *Output* in diesem Fall haben kann, befinden sich zwischen -1 und 1, da die letzte Transformation die *Tanh* Funktion beinhaltet (**Unterkapitel** 2.4.5). Was man gut erkennen kann, ist, dass bis zur Epoche 35 sich grundsätzlich nichts im *Output* dieses *Layers* ändert (dies sehen wir auch in **Abbildung 25** grüne Linie - model_11). Erst nach dieser Epoche fängt das Modell an zu lernen und ordnet die meisten *Outputs* nah an die Null und nur wenige in Richtung -1 und 1, was gut ist in der Hinsicht, dass nur manche Informationen aktivieren werden (die kann man besonders gut auf **Grafik h)** beobachten). In **Grafik e) & f)** erkennt man, so wie in den vorher beschriebenen Grafiken, den *Output* für *LSTM Layer 2*. Der Unterschied hier ist, dass die Werte des *Outputs* anfangs stärker Richtung 1 gestreut sind und die Varianz der Daten ab Epoche 35 größer wird, im Vergleich mit dem erstem *Layer*. Als nächstes folgt der *Output* des *Fully Connected Layers* (*Dense Layers*), der keine Aktivierungsfunktion beinhaltet und deswegen keine begrenzten Werte ausgibt. Hier kann man beobachten, dass sich die links steile Verteilung nicht viel in den ersten 30 Epochen verändert und danach sowohl Änderungen im Mittelwert, der sich ins Negative verschiebt und Veränderungen in der Varianz, die größer wird, aufweist. Darüber hinaus ändert sich die ganze Verteilung über die Iterationen. In den ersten Epochen sieht man eine Verteilung die zwei Spitzen hat und in den letzten sieht man eine Verteilung mit nur einer Spitze, die rechts schief ist. In den beiden letzten **Grafiken a) & b)** sieht man den *Output* aus der *Softmax* Aktivierungsfunktion. Wieder fängt das Modell erst nach den ersten 35 Epochen an, *Outputs* aus dem *Dense Layer* zu aktivieren (Wert 1) hält jedoch die meisten Aktivierungen nah an der Null. Dies ist sehr gut, weil das Modell gelernt hat, nur wenigen Klassen eine höhere Wahrscheinlichkeit zuzuordnen.

Abbildung 27: Histogramme und Verteilungen der *RNN* Elemente

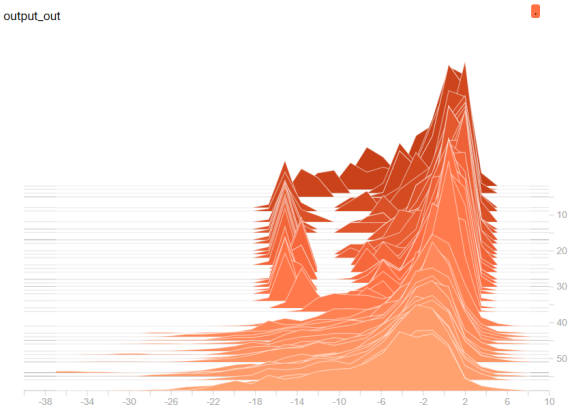
(a) Histogramm *Activation Output*



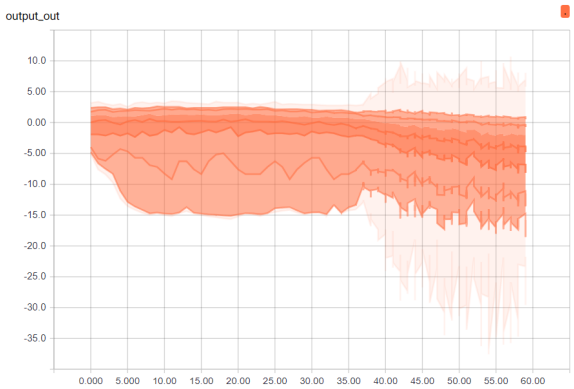
(b) Verteilung *Activation Output*



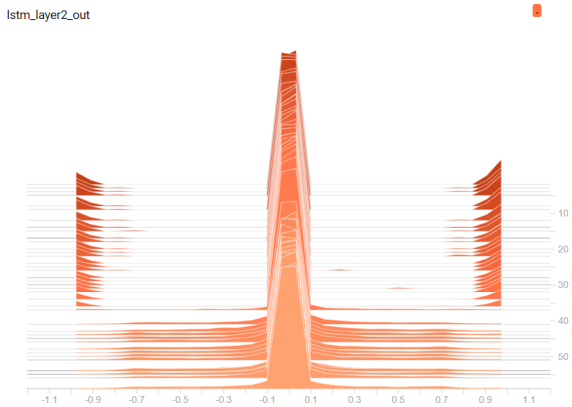
(c) Histogramm *Dense Layer Output*



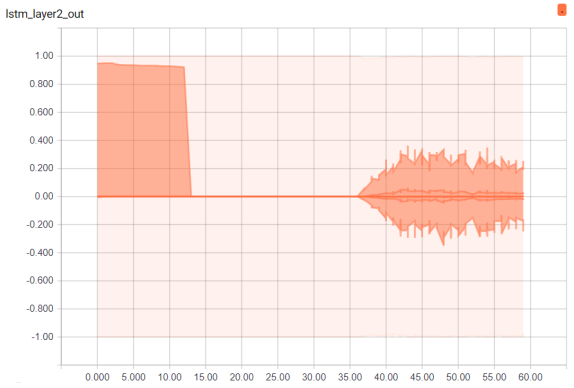
(d) Verteilung *Dense Layer Output*



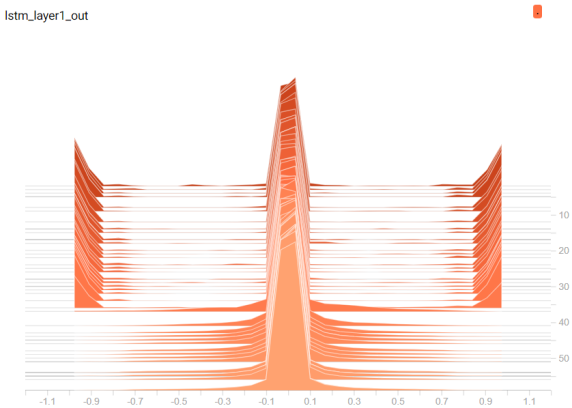
(e) Histogramm *LSTM Layer II Output*



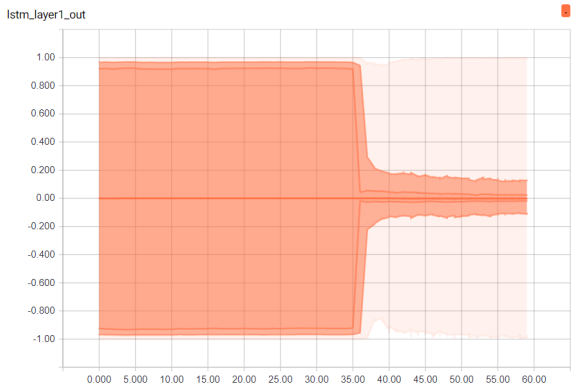
(f) Verteilung *LSTM Layer II Output*



(g) Histogramm *LSTM Layer I Output*



(h) Verteilung *LSTM Layer I Output*



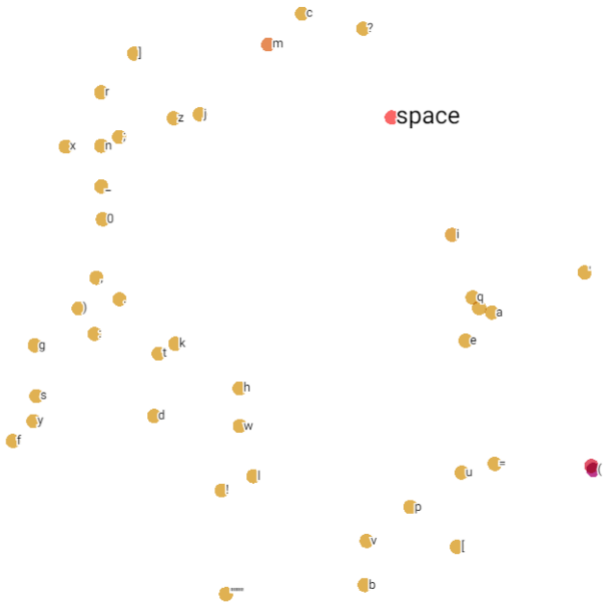
Quelle: Eigene Darstellung

In **Abbildung 28** sieht man die Vektor Darstellung der Zeichen aus dem Vokabular. Die *Embedding Space* wurde mit Hilfe von zwei Methoden auf 3 Dimensionen reduziert, um die Zusammenhänge zwischen den Zeichen visuell darstellen zu können. Die zwei Methoden sind *Principal Component Analysis (PCA)* und *t-Distributed Stochastic Neighbor Embedding (t-SNE)* (Wattenberg et al. 2016). Hieraus kann man erkennen welche Zeichen sich ähnlich sind, wenn es um das Auftreten in einer Sequenz geht. Dies wurde nur für eine Modellarchitektur dargestellt, da das Berechnen von *Embedding* aus den Daten besonders rechenaufwendig ist. Dadurch, dass die *Embedding Space* eigentlich für eine höhere Dimension berechnet wurde, ist die Interpretation in niedrigeren Dimensionen schwer, weil die spezifischen Distanzen im Original-Raum (zum Beispiel 10 Dimensionen) nicht immer erhalten werden kann.

Es ist aber möglich, unterschiedliche Gruppen von Zeichen zu identifizieren:

1. *space* (Leerzeichen) ist das meist vorkommende Zeichen.
2. Spezial Zeichen kommen nicht so oft vor und treten oft zusammen auf.
3. Manche Buchstaben treten oft in Kombination miteinander auf.

Abbildung 28: Embedding Clusters



Quelle: Eigene Darstellung

Als nächstes wurde der generierte Text für verschiedene Epochen und Architekturen für eine zufällig aus dem Trainingsdatensatz *gesampelte* Sequenz (die als Anfangssequenz genutzt wird) dargestellt. Außerdem wurden verschiedene *Perplexties* verwendet, um den Algorithmus unterschiedliche Freiheiten zum Generieren von Texten zu geben. Die *Perplexity* entscheidet darüber, wie die Wahrscheinlichkeitsverteilung des vorausgesagten Zeichens aussehen wird. Mit diesem Parameter kann man also die Kreativität des geschriebenen Textes beeinflussen. Ein Text mit

einer niedrigen *Perplexität* wird versuchen, immer das Zeichen mit der größten Wahrscheinlichkeit vorauszusagen. Indem man den Parameter höher setzt, erreicht man, dass der Algorithmus auch Zeichen voraussagt, die weniger wahrscheinlich sind. Ein zu hoher Parameter führt dazu, dass Buchstaben gewählt werden, die im Endeffekt Wörter kreieren, die nicht existieren (Die Buchstaben werden vollkommen zufällig gewählt). Die Wahl der optimalen Größe dieses Parameters ist nicht einfach und kann nicht anhand eines Kriteriums festgelegt werden, sondern muss für jedes individuelle Problem nach Gefühl gewählt werden. Das folgende Beispiel zeigt den generierten Text für das Modell mit der id *model_6* (Das Modell mit dem niedrigstem Verlust von allen) für verschiedene *Perplexity* Parameter und in unterschiedlichen Epochen. Die Gewichte des Modells sind anfangs zufällig initialisiert und müssen erst über die verschiedenen Iterationen (Epochen) trainiert werden. Deswegen erkennt man, dass mit ansteigender Anzahl der Epochen der Text immer besser wird. Anfangs werden nur Texte generiert, die keinen Satz oder Wort bilden. Für Epoche 1 und *Peplexity* 0.2, erhält man einen Text, der nur den Verlust so klein wie möglich halten möchte und deswegen Zeichen voraussagt, die für diesen Zeitpunkt des Modells am wahrscheinlichsten sind - also: Leerzeichen, e, t etc.. Wenn man dem Modell mehr Freiheit gibt, erhält man einen Text der Zeichen voraussagt, die unwahrscheinlicher sind. Bei Epoche 30 hat das Modell schon genug gelernt, um tatsächlich Wörter zu bilden, die existieren. Das erste Wort, dass das Modell nach den ersten *Epochen* gelernt hat, war das Wort "the" (dies ist das am häufigsten vorkommende Wort in allen Texten). Bei *Peplexity* 0.2 will sich das Modell, wie zuvor, sehr sicher mit dem vorausgesagten Zeichen sein. Diesmal aber mit dem Unterschied, dass das Modell mehr Wissen besitzt als in *Epoche* 1. Das Ergebnis daraus ist also ein Text, der allgemein oft vorkommende Wörter enthält, wie: *the*, *of*, *in* oder *company*. Bei höheren *Perplexity* Parametern ist der Text reicher an Wortschatz, allerdings entstehen auch Wörter die gar nicht existieren (aber plausibel klingen). Bei *Perplexity* 1 wird ein Text generiert, der über keine logischen Zusammenhänge mehr verfügt. In der *Epoche* 55, in der der niedrigste Verlust erreicht worden ist, sieht man bei allen Parametereinstellungen, dass der generierte Text Wörter enthält, die es gibt und dass das Modell versucht, durch den ganzen Text eine gewisse Information weiterzuführen, auch wenn es an diesem Zeitpunkt nicht gelungen ist, einen sinnvollen Text zu erschaffen.

Modell id: model_6
Perplexity: 0.2
Epoch: 1

```
eo  e     e  e e        i    te      e    e    e  eee   e   i   e   e e     t
        e   e              e   t    e e ei e  t  e ee         i  ete
                e  t  e  e                e et  e        ai        e   e   e     t
       te        e   e  e    eei    ne      i   o    e  ee        e     e ie
         e e      e e ee ee ee e   e   i      e    e   e   eee ee
```

Modell id: model_6

Perplexity: 0.5
Epoch: 1

```
idt oa ia e chtoot ii nueu cot s sio s r a ie nea e u eee nhtt t heotn ote
    oantetos n rdele uettts evpee e t to eese inihn ee o eee e i titsitaetet
    oiset eea ueae iol e n i iegieii deiti s reaen ti etden tooe utwe noe e
    tteivc adi l n s tr e eeeteeo a e ee td a safeyich eeeeua saeie e eie
    eunetenaai riees p ee tte o din t t ttitte r e uu s issete ei aei yoin a
```

Modell id: model_6
Perplexity: 1
Epoch: 1

```
alna oe msylttiat btagrefeso sce snle r ivailwu iuiisihpccpeitrpthjdsne.tseis
    uttsreptt esi oysfdnuireteeoyepohei hncs iesitoae oardtoosp wsl eeeft i ttsc
    onfsiufeersaperpfeam erhet diaaisa u nmwbwobdreiy muy ssueygleitdciidsetfuto
    ae t istud saieetimeutr ttserq de i cursbamftscaocdnseiwofbtrdtuu iaasevedu
    otwane:kio nits eat.tehs nfu upsceiisnso nf pr tsyens ehpevnogn rituf
    tsssrkneoits si_ a
```

Modell id: model_6
Perplexity: 0.2
Epoch: 30

```
y as we have tor and an ant the services the companys in the company in the
    companys and and our structure of the production of the structured and
    services and production to the results of the production and production and
    products and and and and and and and our stock exchange on and executive
    offices) () has been subject to such filing requirements for the past days.
    yes no no indicate by chec
```

Modell id: model_6
Perplexity: 0.5
Epoch: 30

```
the the company in the annual results in the based on the results with a companys
     in the result and the despore our financial statements in the companying the
    group the company of the are and an executive officers in results in the
    management in the mortgage success commitment to a results income of directors
     report solution of the group of the forward interest in the results in a
    success of exp
```

Modell id: model_6

Perplexity: 1
Epoch: 30

```
supmotile of (mosic and this reporting and monthlision antice vased exceles on
    industry part symert every intexests wyut dolly. incurecreductured strents
    nonuc-as ata of the registrant has subs and sell of proment sides grouthor
    for our thries and goodban ricox, chort uls degrowth, thatficon other ,
    amital brong lrading , product of has progresss sher financial simiaral hire
    management position t
```

Modell id: model_6
Perplexity: 0.2
Epoch: 56

```
include the company and securities registered pursuant to section (g) of the act:
    none indicate by check mark if the registrant is not required to file reports
    pursuant to rule of regulation s-t (. of this chapter) during the preceding
   months (or for such shorter period that the registrant was required to file
   reports pursuant to rule of regulation s-t (. of this chapter) during the
   preceding
```

Modell id: model_6
Perplexity: 0.5
Epoch: 56

```
pection, factors of a customers group statement and investor responsibility in
    simple of financial performance and provide increase consolidated statements
    directory and post shareholder maintained assets for a company for a market
    and securities registered pursuant to rule of regulation s-t (. of this
    chapter) and deliver stock exchange committen value chaingant company (trie
    group in the stre
```

Modell id: model_6
Perplexity: 1
Epoch: 56

```
nd and applications) must has manufacturing projetces to shareholders table and
    project. was definition and entruse will todselver, the cruss an
    accismonctrated providing ide for over customers of during the world
    managements western, and exceed shareholders maintainancins, while liment by
    ausitions estate pertents products materials resoliration, identiate
    acquisition and group business intorp
```

4 Zusammenfassung

Die vorliegende Masterarbeit ist ein Versuch oder auch erster Schritt in Richtung eines voll automatisierten Textgenerierungprozesses. Viel Zeit wird in verschiedenen Bereichen für das Schreiben unterschiedlichster Texte investiert, wobei viele davon eine meist formelle und strikte Struktur haben und deswegen automatisiert werden können. Neue Methoden erlauben es, das Problem der Textgenerierung neu zu betrachten. Neue Übersetzungsmethoden und ein automatisches Verfassen von E-Mails wird heutzutage von großen Firmen, wie z.B. *Google*, in deren Produkte, eingeführt. Diese basieren immer öfter auf *Deep Learning* Methoden, wie *Convolutional Neural Networks* oder *Recurrent Neural Networks*, die noch vor kurzem, aus technologischen Gründen, nicht weit verbreitet waren. Die Entwicklung neuer Prozessoren und die Erfindung neuer Algorithmen haben es ermöglicht, das volle Potenzial dieser Methoden zu nutzen und mit deren Errungenschaften die Forschung und das Interesse in diesem Gebiet weiter voranzutreiben. Die Entwicklung und Hindernisse dieser Methoden wurden genauer in der Arbeit besprochen und deren heutiges Aussehen grundlegend beschrieben. Da die Aufgabe im Erlernen von Schreiben von Texten basiert, die wiederum eine sequenzielle Form von Daten darstellen, wurde der Schwerpunkt der Arbeit auf die Erklärung von *Recurrent Neural Networks* (bzw. *Long Short-Term Memory* Modelle) gelegt. *LSTM* Modelle erlauben es, die größten Probleme von *RNNs* zu beheben, was deren wachsende Popularität fördert. Um die durchaus komplexe und oft nicht tiefgehend beschriebenen (in der Literatur) *LSTM* Zellen zu verstehen, wurde eine Analyse von den Grundlagen der *Neural Networks* bis zu den fortgeschrittenen Methoden in *Recurrent Neural Networks* der jeweiligen Elemente durchgeführt. Die Wahl dieser Elemente kann von den Nutzern dieser Methoden frei gewählt werden, weshalb die Arbeit in einer Modularen Struktur aufgebaut wurde.

Der erste Teil der Arbeit befasst sich mit der allgemeinen Beschreibung des Problems der Textgenerierung, sowie mit einem Überblick der Literatur, die es zu diesem Thema gibt. Der zweite Teil hingegen befasst sich mit den Grundlagen der *Neural Networks*, sowie mit dem Problem des *Encodings* der Daten und den Hilfsalgorithmen, die es einem Netz ermöglichen, dessen Prädiktion zu optimieren. Zunächst wurden deshalb der *Gradient Descent* Optimierungs Algorithmus und seine Erweiterungen und Abwandlungen vorgestellt. Diese werden zur Optimierung von Verlustfunktionen verwendet, deren Funktion im Trainingsprozess auch dargestellt wurden. Die Verlustfunktion, die von besonderer Relevanz für die Aufgabe der Texgenerierung ist, ist die *Cross-Entropy* Funktion, deren Gleichung aus der *Kullback Leibler Divergenz* hergeleitet wurde. Diese ermöglicht es verschiedene Wahrscheinlichkeitsverteilungen miteinander zu vergleichen. Zusätzlich wurde in diesem Teil noch die Bedeutung der unterschiedlichen Aktivierungfunktionen beschrieben, die es ermöglichen, nichtlineare Zusammenhänge aus den Daten zu erlernen, um im Endeffekt mit Hilfe vom *Backpropagation* Algorithmus die Parameter (Gewichte) des Modells in einer rückwirkender Art zu transformieren. Als nächstes wurde der Hauptteil der Arbeit beschrieben, also die *Recurrent Neural Networks* sowie die Erweiterung - in diesem Fall

der vorher vorgestellten Algorithmen. Erstens wurden mögliche Hürden wie *Vanishing* und *Exploding Gradients* in Hinsicht auf *RNN* beschrieben, um zweitens zu zeigen, wie *Truncated Backpropagation Through Time* und *Long Short-Term Memory* Zellen diese überwinden können. Im Vergleich zum Mechanismus der *LSTM* Zelle wurde die *Gated Recurrent Unit* Zelle dargestellt, die eine Alternative zu dem Erstgenannten ist. Der dritte Teil der Arbeit fängt mit einer Darstellung der Gewinnung und Verarbeitung der Daten an, die verwendet wurden, um das Erlernen des Textgenerierungsprozesses möglich zu machen. Der Datensatz der hier verwendet worden ist, sind Jahresabschlüsse von Unternehmen, da dieses Gebiet von Zeitersparnissen besonders profitieren könnte und die Beschaffung der Daten relativ einfach und zugänglich ist. Im weiteren Schritt wurden dann verschiedene Modelle (mit unterschiedlichen *Parameter Settings*) ausprobiert, um letztlich ein nützliches Modell zu bekommen. Die Qualität der Modelle wurde anhand von verschiedenen Kriterien verglichen:

1. die Minimierung des Verlustes, der es erlaubt zu verfolgen, ob sich das Modell über die fortschreitenden Epochen verbessert hat.

2. die Histogramme und Verteilungen der verschieden Elemente der *RNN*, die auf Probleme hinweisen können

3. der generierte Text, der das Endprodukt des Modells ist und deswegen der beste Maßstab ist, ob das Ziel erreicht wurde oder nicht.

Insgesamt wurden ungefähr 48 Tausend PDF Dokumente mit Jahresabschlüssen für die Durchführung der Analyse vorbereitet. Aus technischen- und Kostengründen war es nicht möglich, alle Dokumente mit ins Modell aufzunehmen. Weitaus bessere Ergebnisse könnten durch den größeren Datensatz erreicht werden und deswegen würde sich eine Skalierung auf einer Maschine mit mehr Rechenkraft (*RAM* oder *VRAM*) lohnen. Aus demselben Grund wurde das Parameter *Tuning* manuell durchgeführt und nur einige Architekturen und *Parameter Setting* wurden ausprobiert. Hierbei würden verschieden *Hyperparameter tuning* Methoden sehr hilfreich sein, die aber wie zuvor voraussetzt, dass die Rechenzeit in einem angemessenen Zeitrahmen bleibt. Darüber hinaus wird immer mehr Aufmerksamkeit den *Attention Based Models* geschenkt, da diese nicht unter den *Hardware* Problemen leiden wie *RNNs* (Culurciello 2018). Des weiteren ist die *Memory* des Modells bei *LSTM* oder *GRU* Strukturen auf *Time Steps* begrenzt, die in die hunderte gehen, aber nicht in die tausende oder mehr. Erste Forschungsarbeiten zeigen, dass *Attention Based Models* durch ihre hierarchische Architektur dazu fähig sind, längere Abhängigkeiten zu erlernen, wodurch sie in vielen Problemstellungen herkömmliche *RNN* Strukturen übertreffen. Die Zukunft dieser Modelle ist zum jetzigen Zeitpunkt allerdings noch unklar und im Gegenteil zu *RNN* sind dies noch Dinge, die am Anfang ihrer Forschung stehen (Culurciello 2018).

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J. & and, D. M. (2015), 'TensorFlow: Large-scale machine learning on heterogeneous systems', http://tensorflow.org/. Software available from tensorflow.org.

Alpaydin, E. (2010), *Introduction To Machine Learning second edition*, MIT Press.

Baoyu Jing, Pengtao Xie, E. X. (2018), 'On the automatic generation of medical imaging reports'. Online; accessed 27 December 2018.

Bischl, B. (2016), 'Fcim / predictive modeling chapter 7: Boosting'.

Brownlee, J. (2017*a*), 'A gentle introduction to backpropagation through time'. Online; accessed 27 December 2018.

Brownlee, J. (2017*b*), 'A gentle introduction to exploding gradients in neural networks'. Online; accessed 27 December 2018.

Chollet, F., Allaire, J. et al. (2017), 'R interface to keras', `https://github.com/rstudio/keras`.

Chollet, F. et al. (2015), 'Keras', `https://keras.io`. Online; accessed 14 December 2018.

Culurciello, E. (2018), 'The fall of rnn / lstm', `https://towardsdatascience.com/the-fall-of-rnn-lstm-2d1594c74ce0`. Online; accessed 01 January 2019.

Falbel, D. (2017), 'Tensorflow for r: Word embeddings with keras', `https://blogs.rstudio.com/tensorflow/posts/2017-12-22-word-embeddings-with-keras`.

Francois Chollet, J. J. A. (2018), *Deep Learning with R*, Manning Publications; 1 edition.

Hao, Z. (2017), 'Loss functions in neural networks', `https://isaacchanghau.github.io/post/loss_functions/`. Online; accessed 29 August 2018.

Kapur, R. (2017), 'Neural networks & the backpropagation algorithm, explained', `https://ayearofai.com/rohan-lenny-1-neural-networks-the-backpropagation-algorithm-explained-abf4609d4f9d`. Online; accessed 29 August 2018.

Karpathy, A. (2015), 'The unreasonable effectiveness of recurrent neural networks', `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`. Online; accessed 01 August 2018.

LeCun, Y. (1988), 'A theoretical framework for back-propagation', `http://yann.lecun.com/exdb/publis/pdf/lecun-88.pdf`.

LeCun, Y. & Ranzato, M. (2013), 'Deep learning tutorial', http://www.cs.nyu.edu/~yann/talks/lecun-ranzato-icml2013.pdf (visited 26.10.2017).

Olah, C. (2015), 'Understanding lstm networks', http://colah.github.io/posts/2015-08-Understanding-LSTMs/. Online; accessed 03 October 2018.

Oser, P. (2017), 'Jahresabschluss'. Online; accessed 29 August 2018.

retrescro (2018), 'Flächendeckende fußballberichterstattung', https://www.retresco.de/textgenerierung/sport/. Online; accessed 27 December 2018.

Ruder, S. (2016), 'An overview of gradient descent optimization algorithms', *CoRR* **abs/1609.04747**. http://arxiv.org/abs/1609.04747.

Scrapinghub, L. (2008), 'Scrapy', https://doc.scrapy.org/en/latest/topics/spiders.html. Online; accessed 29 August 2018.

Sepp Hochreiter, J. S. (1997), 'Long short-term memory', https://www.bioinf.jku.at/publications/older/2604.pdf.

Sharma, S. (2017), 'Activation functions: Neural networks', https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. Online; accessed 14 December 2018.

Souppouris, A. (2015), 'I taught a computer to write like engadget', https://www.engadget.com/2015/12/02/neural-network-journalism-philip-k-dick/. Online; accessed 22 July 2018.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. (2014), 'Dropout: A simple way to prevent neural networks from overfitting'.

Tushar, A. A. (2017), '3 types of gradient descent algorithms for small large data sets', https://www.hackerearth.com/blog/machine-learning/3-types-gradient-descent-algorithms-small-large-data-sets/. Online; accessed 22 July 2018.

Walter Pitts, W. M. (1943), 'A logical calculus of the ideas immanent in nervous activity', http://www.cse.chalmers.se/~coquand/AUTOMATA/mcp.pdf.

Wattenberg, M., Viégas, F. & Johnson, I. (2016), 'How to use t-sne effectively', *Distill* . **URL:** *http://distill.pub/2016/misread-tsne*

Werbos, P. J. (1990), 'Backpropagation through time: what it does and how to do it'. Online; accessed 27 December 2018.

Williams, R. J. & Peng, J. (1990), 'An efficient gradient-based algorithm for on-line training of recurrent network trajectories'. Online; accessed 27 December 2018.

Anhang

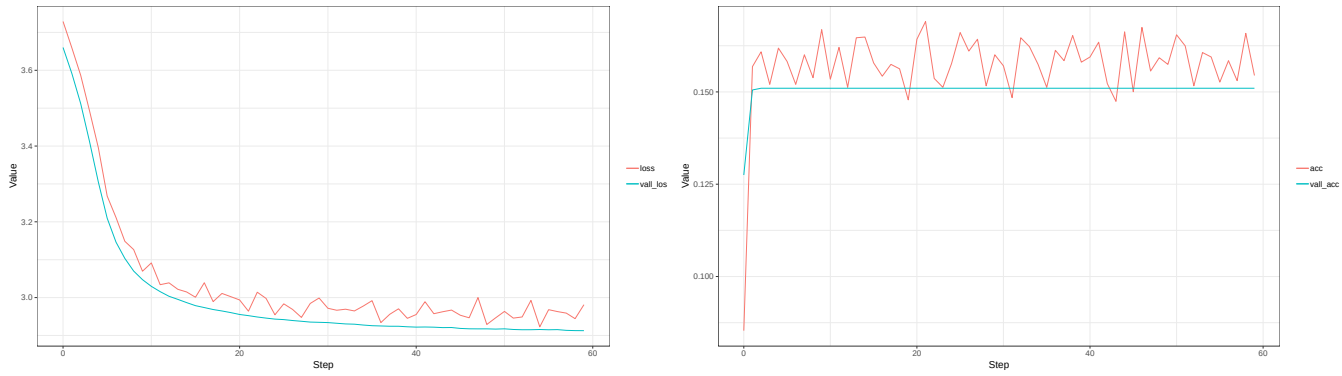
Tabelle 6: Parameter Setting

model id	optimizer	learn rate	number of files	batch size	mini batch size	maxlen	L1 units	L2 units	L3 units	epochs	min loss	test loss	acc
model_1	SGD	0.01	478	12	128	100	128	-	-	60	2.92	2.91	0.16
model_2	Momentum	0.01	478	12	128	100	128	-	-	60	2.65	2.62	0.25
model_3	RMSprop	0.01	478	12	128	100	128	-	-	60	1.19	1.8	0.66
model_4	RMSprop	0.01	478	12	128	100	256	128	-	60	1.25	1.85	0.64
model_5	RMSprop	0.01	478	12	256	100	256	128	-	60	1.06	1.65	0.69
model_6	RMSprop	0.01	478	12	512	100	256	128	-	60	0.93	1.5	0.73
model_7	RMSprop	0.01	478	12	256	50	256	128	-	60	1.01	1.67	0.69
model_8	RMSprop	0.01	478	12	256	150	256	128	-	60	1.16	1.64	0.66
model_9	RMSprop	0.01	478	12	256	150	512	256	-	60	1.38	1.65	0.6
model_10	RMSprop	0.01	478	12	512	150	256	128	-	60	0.95	1.52	0.72
model_11	RMSprop	0.01	478	12	512	150	512	256	-	60	1.28	1.58	0.62
model_12	RMSprop	0.001	478	12	512	150	512	256	128	60	1.21	1.67	0.66
model_13	RMSprop	0.001	478	12	256	150	512	256	128	60	1.42	1.78	0.6

Abbildung 29: Verlust und Accuracy für Trainings und Validierungs Datensatz - model_1

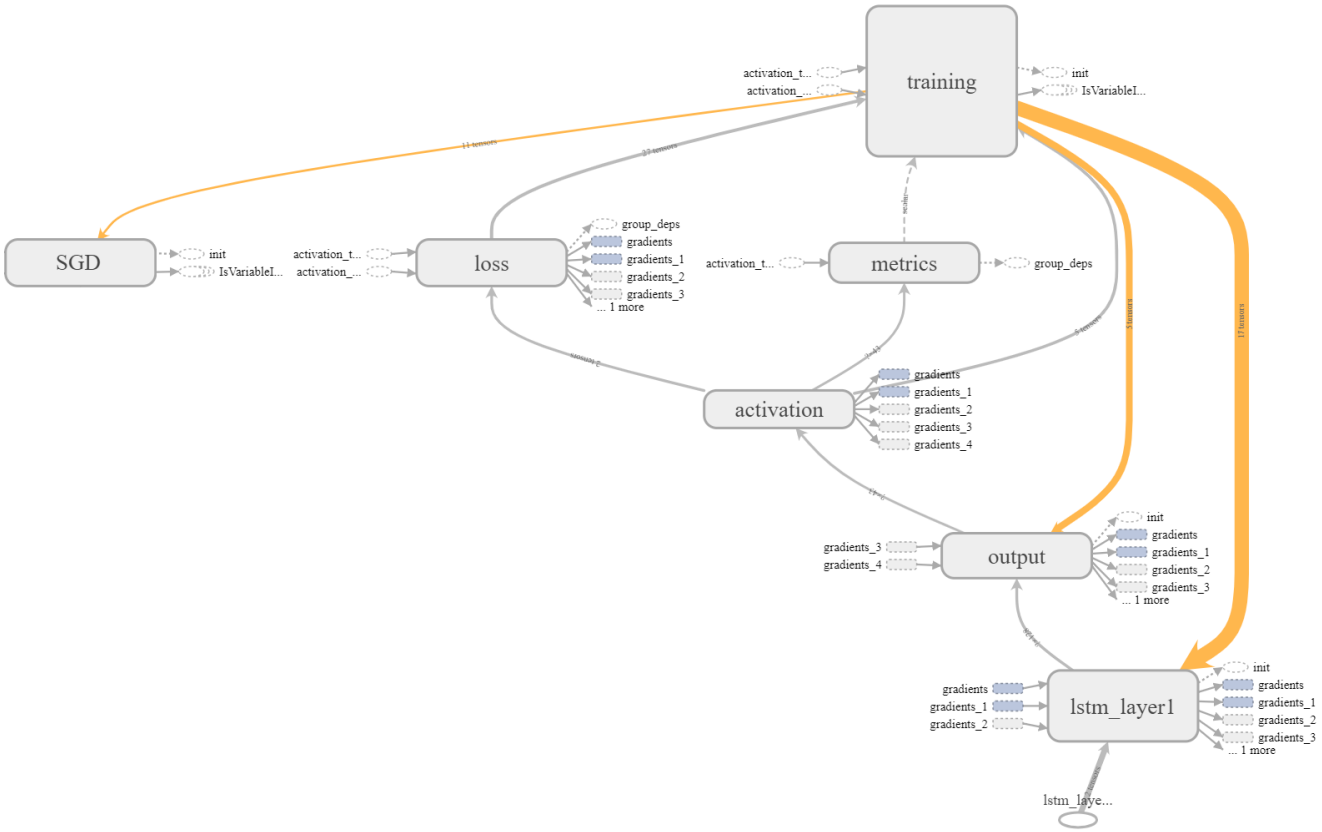
(a) Verlust

(b) Accuracy



Quelle: Eigene Darstellung

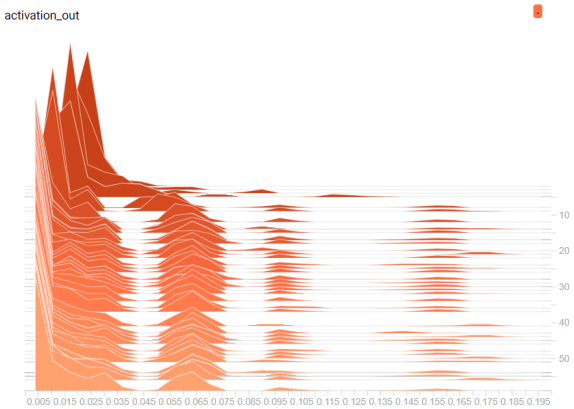
Abbildung 30: Graph model_1



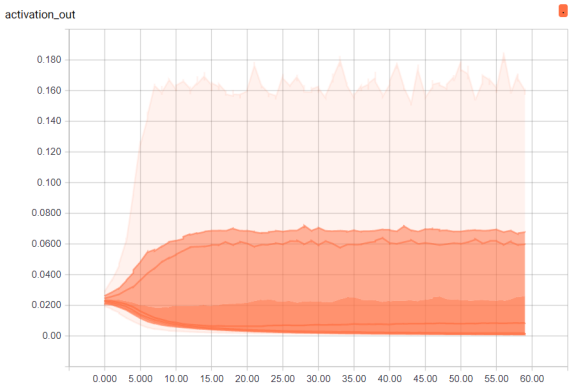
Quelle: Eigene Darstellung

Abbildung 31: Histogramme und Verteilungen der *RNN* Elemente Modell 1

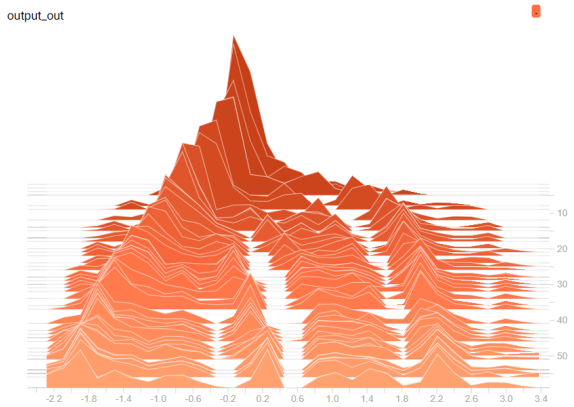
(a) Histogramm *Activation Output*



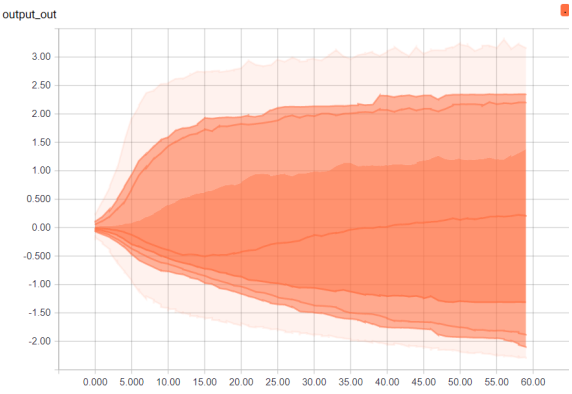
(b) Verteilung *Activation Output*



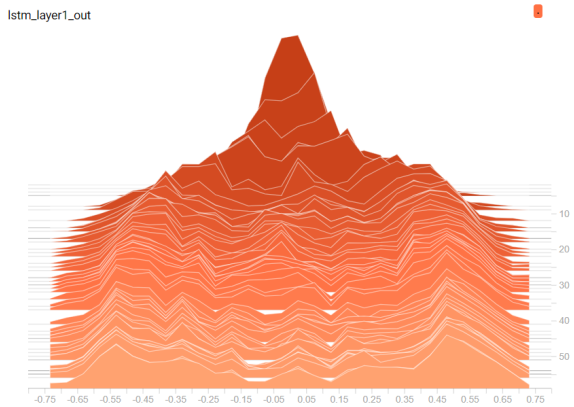
(c) Histogramm *Dense Layer Output*



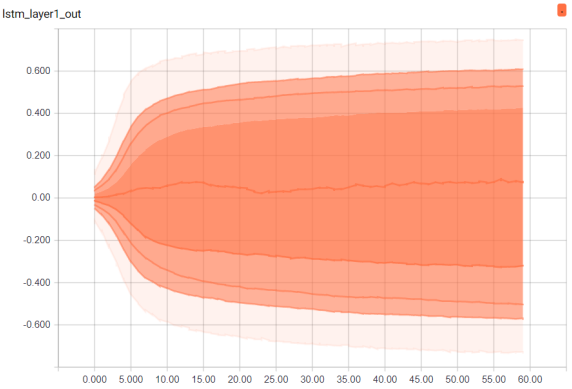
(d) Verteilung *Dense Layer Output*



(e) Histogramm *LSTM Layer I Output*



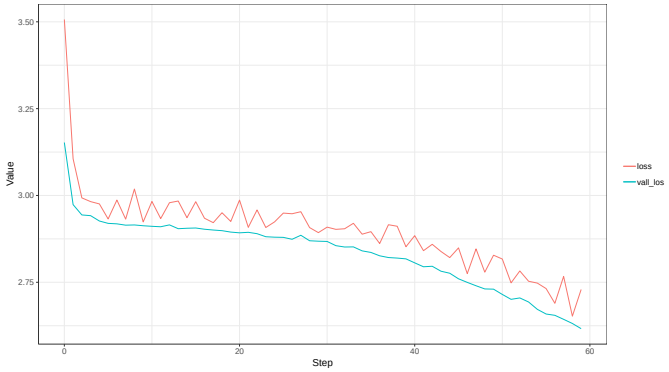
(f) Verteilung *LSTM Layer I Output*



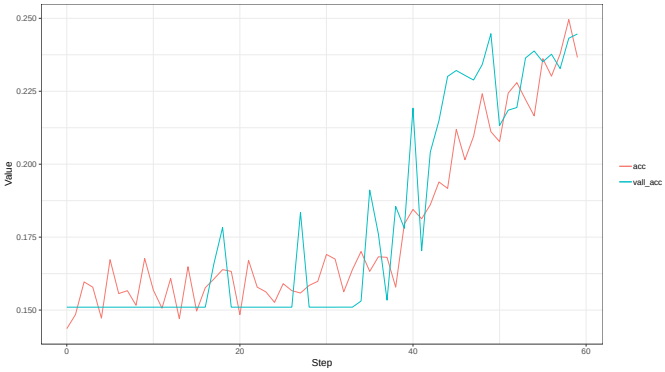
Quelle: Eigene Darstellung

Abbildung 32: Verlust und Accuracy für Trainings und Validierungs Datensatz - model_2

(a) Verlust

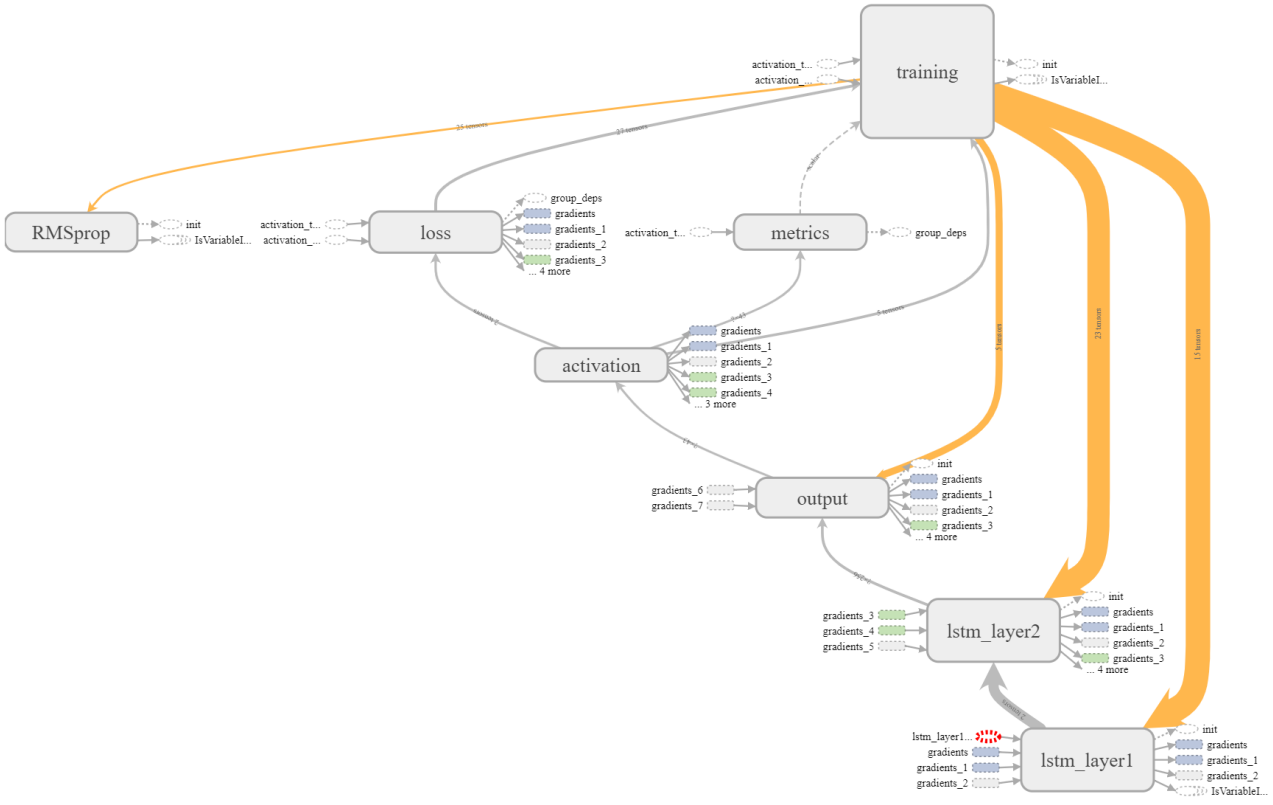


(b) Accuracy



Quelle: Eigene Darstellung

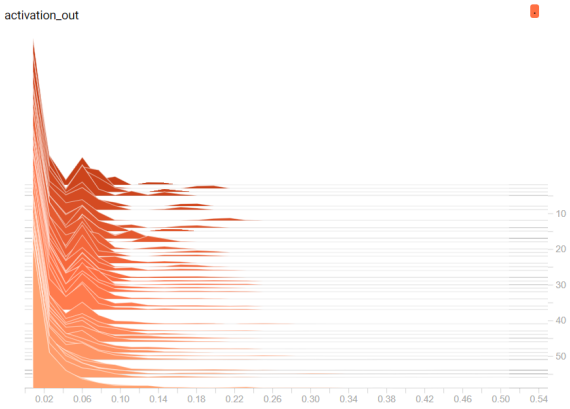
Abbildung 33: Graph model_2



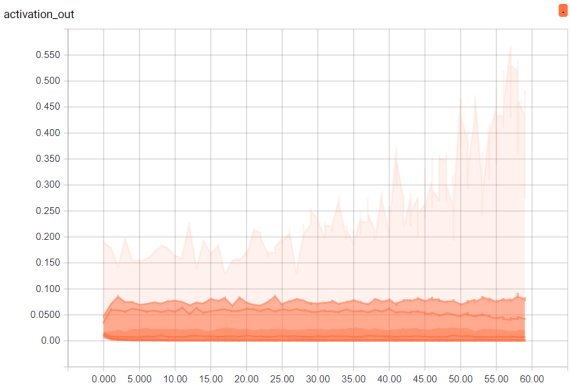
Quelle: Eigene Darstellung

Abbildung 34: Histogramme und Verteilungen der *RNN* Elemente Modell 2

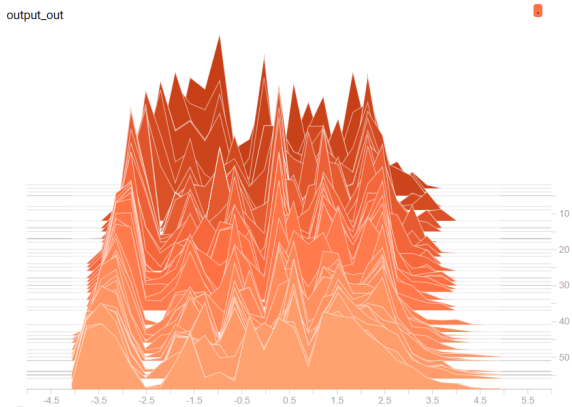
(a) Histogramm *Activation Output*



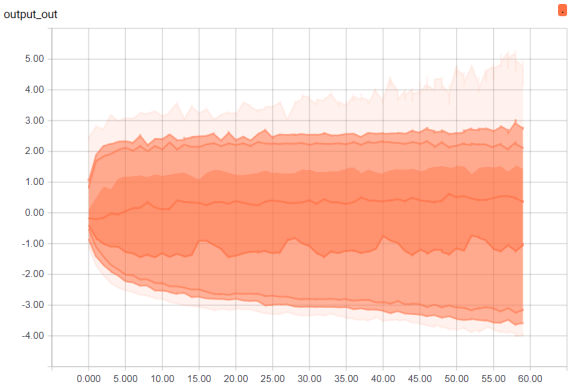
(b) Verteilung *Activation Output*



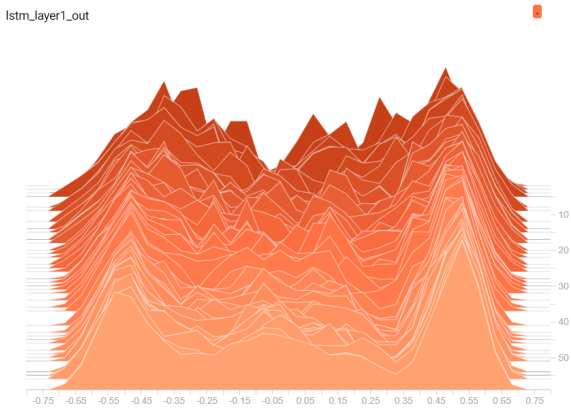
(c) Histogramm *Dense Layer Output*



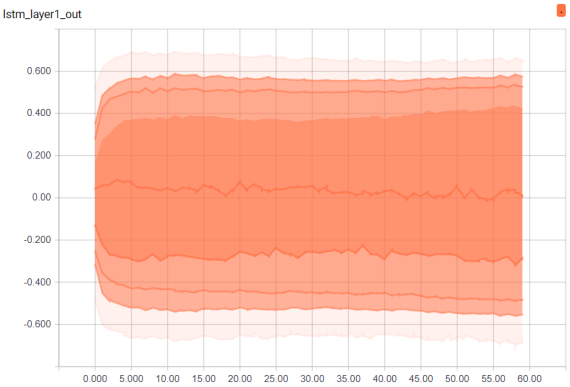
(d) Verteilung *Dense Layer Output*



(e) Histogramm *LSTM Layer I Output*



(f) Verteilung *LSTM Layer I Output*

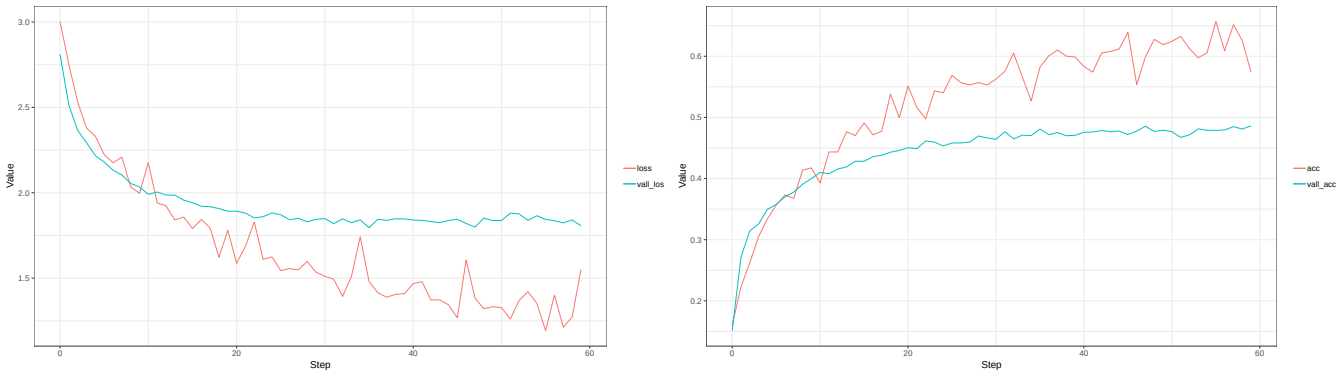


Quelle: Eigene Darstellung

Abbildung 35: Verlust und Accuracy für Trainings und Validierungs Datensatz - model_3

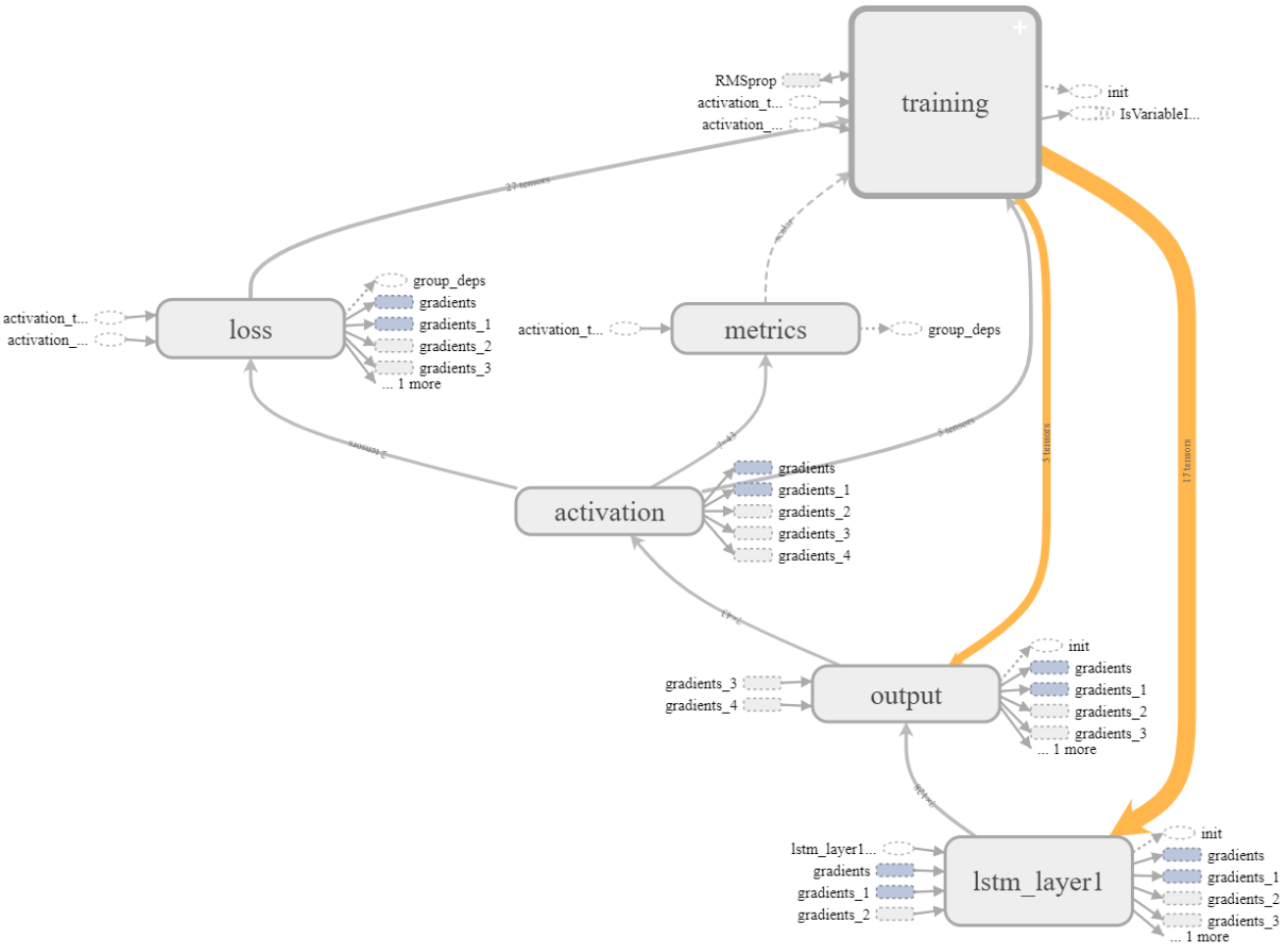
(a) Verlust

(b) Accuracy



Quelle: Eigene Darstellung

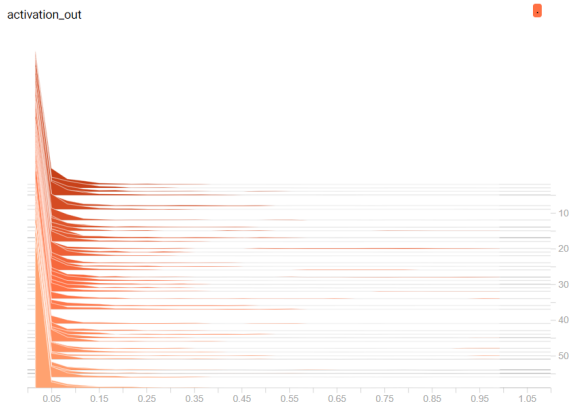
Abbildung 36: Graph model_3



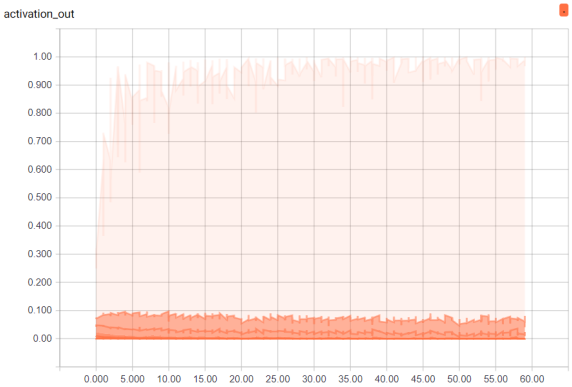
Quelle: Eigene Darstellung

Abbildung 37: Histogramme und Verteilungen der *RNN* Elemente Modell 3

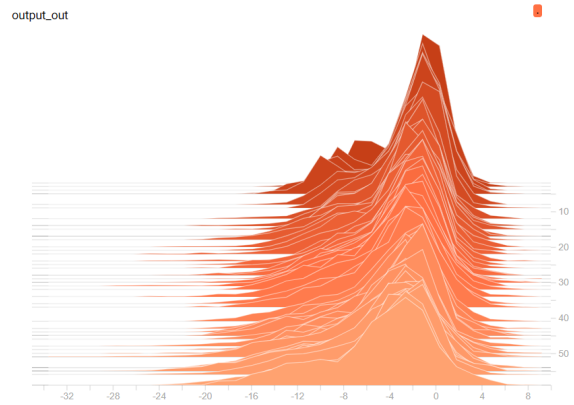
(a) Histogramm *Activation Output*



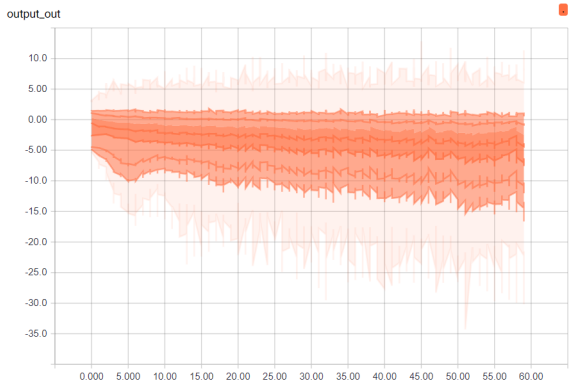
(b) Verteilung *Activation Output*



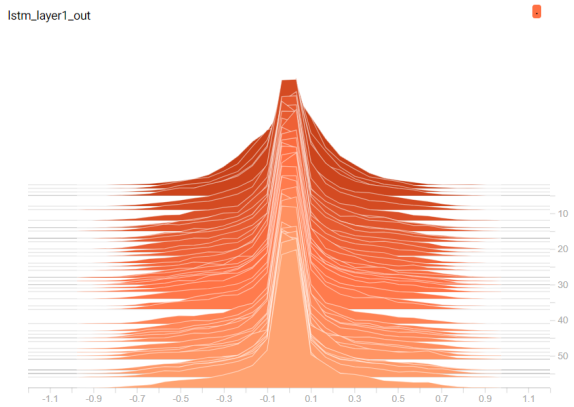
(c) Histogramm *Dense Layer Output*



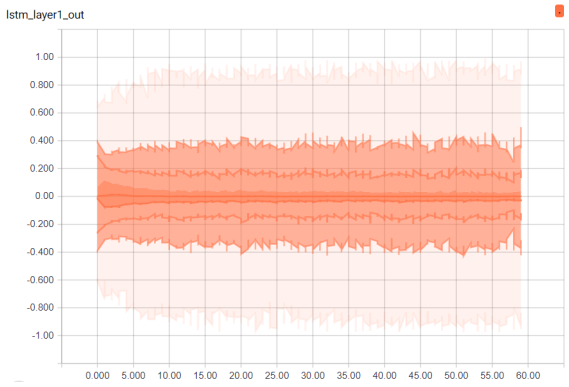
(d) Verteilung *Dense Layer Output*



(e) Histogramm *LSTM Layer I Output*



(f) Verteilung *LSTM Layer I Output*

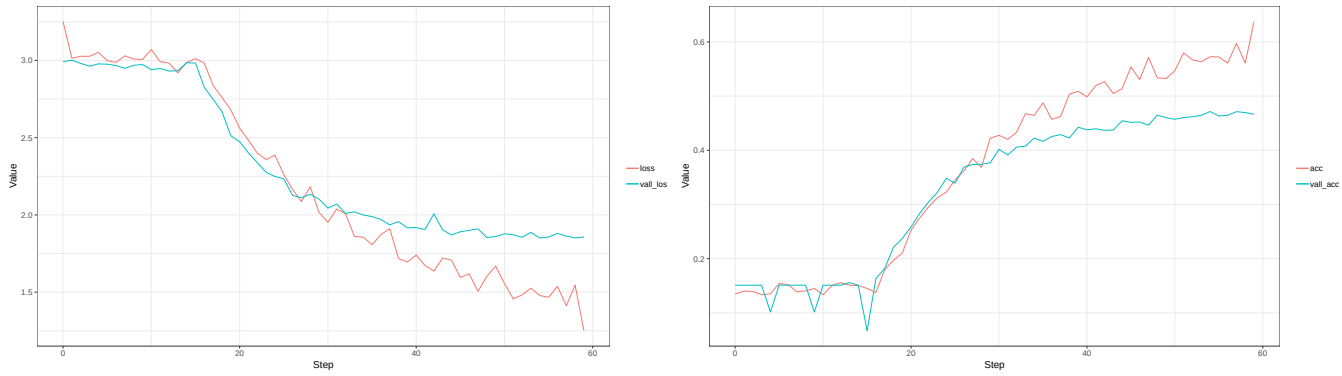


Quelle: Eigene Darstellung

Abbildung 38: Verlust und Accuracy für Trainings und Validierungs Datensatz - model_4

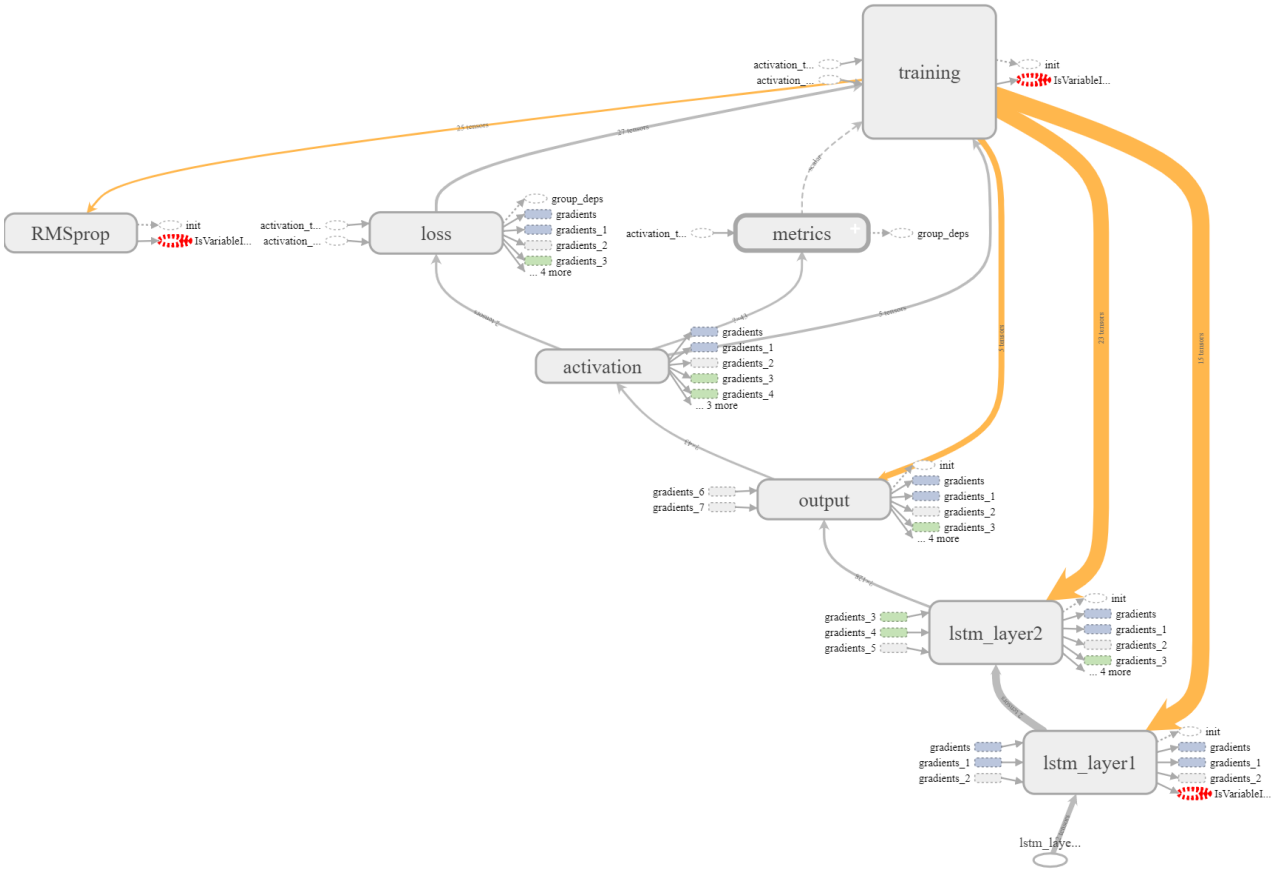
(a) Verlust

(b) Accuracy



Quelle: Eigene Darstellung

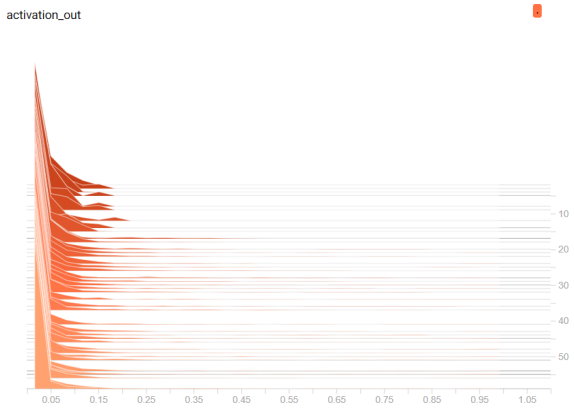
Abbildung 39: Graph model_4



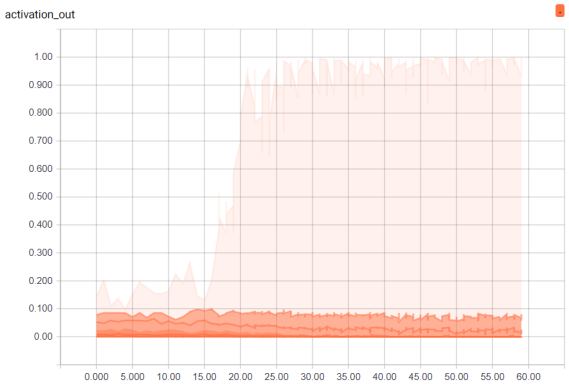
Quelle: Eigene Darstellung

Abbildung 40: Histogramme und Verteilungen der *RNN* Elemente Modell 4

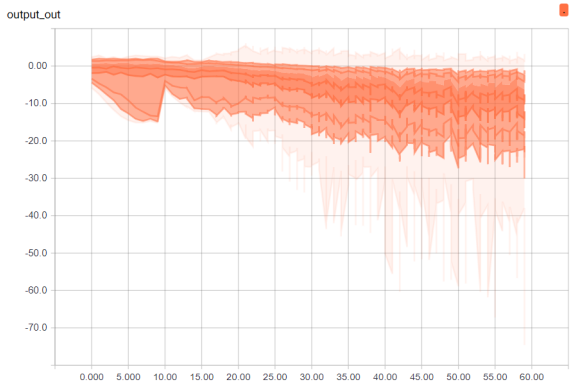
(a) Histogramm *Activation Output*



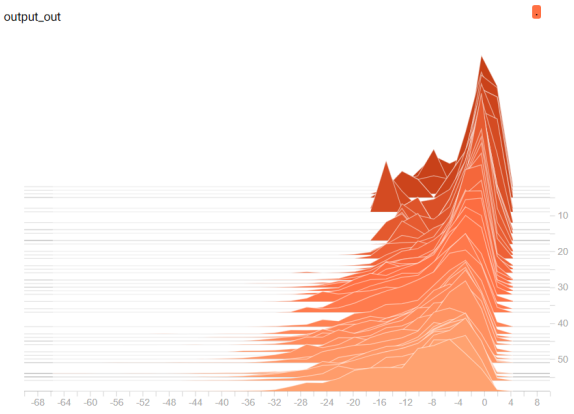
(b) Verteilung *Activation Output*



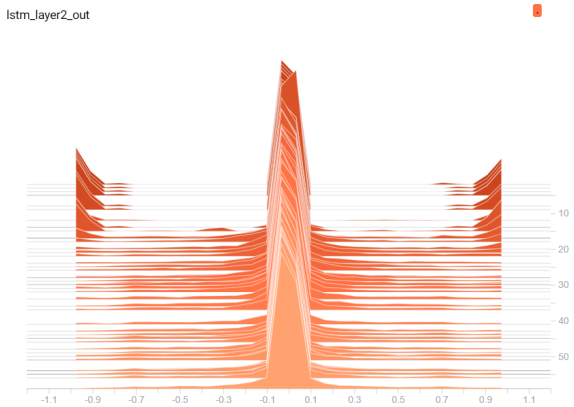
(c) Histogramm *Dense Layer Output*



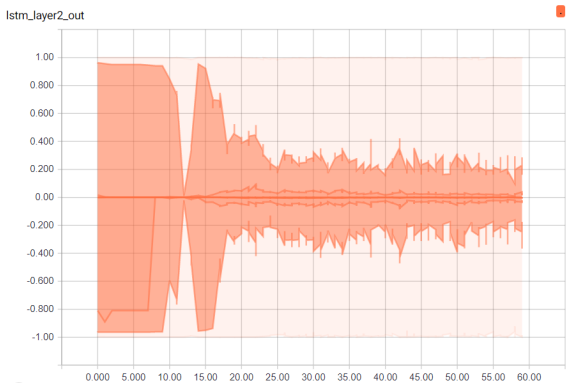
(d) Verteilung *Dense Layer Output*



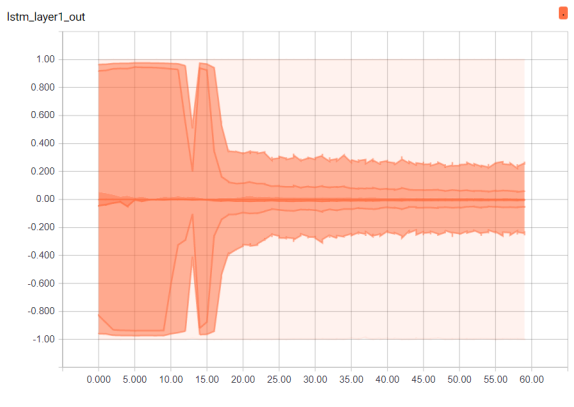
(e) Histogramm *LSTM Layer II Output*



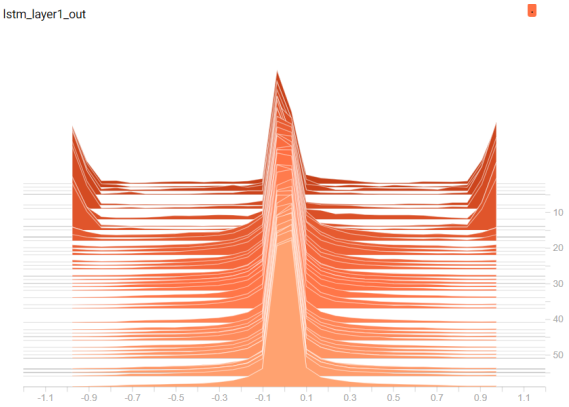
(f) Verteilung *LSTM Layer II Output*



(g) Histogramm *LSTM Layer I Output*



(h) Verteilung *LSTM Layer I Output*

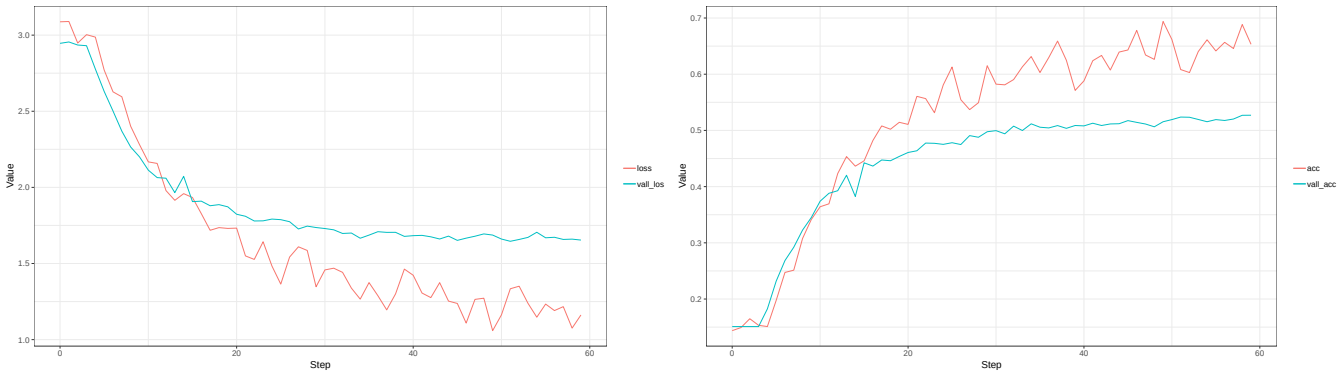


Quelle: Eigene Darstellung

Abbildung 41: Verlust und Accuracy für Trainings und Validierungs Datensatz - model_5

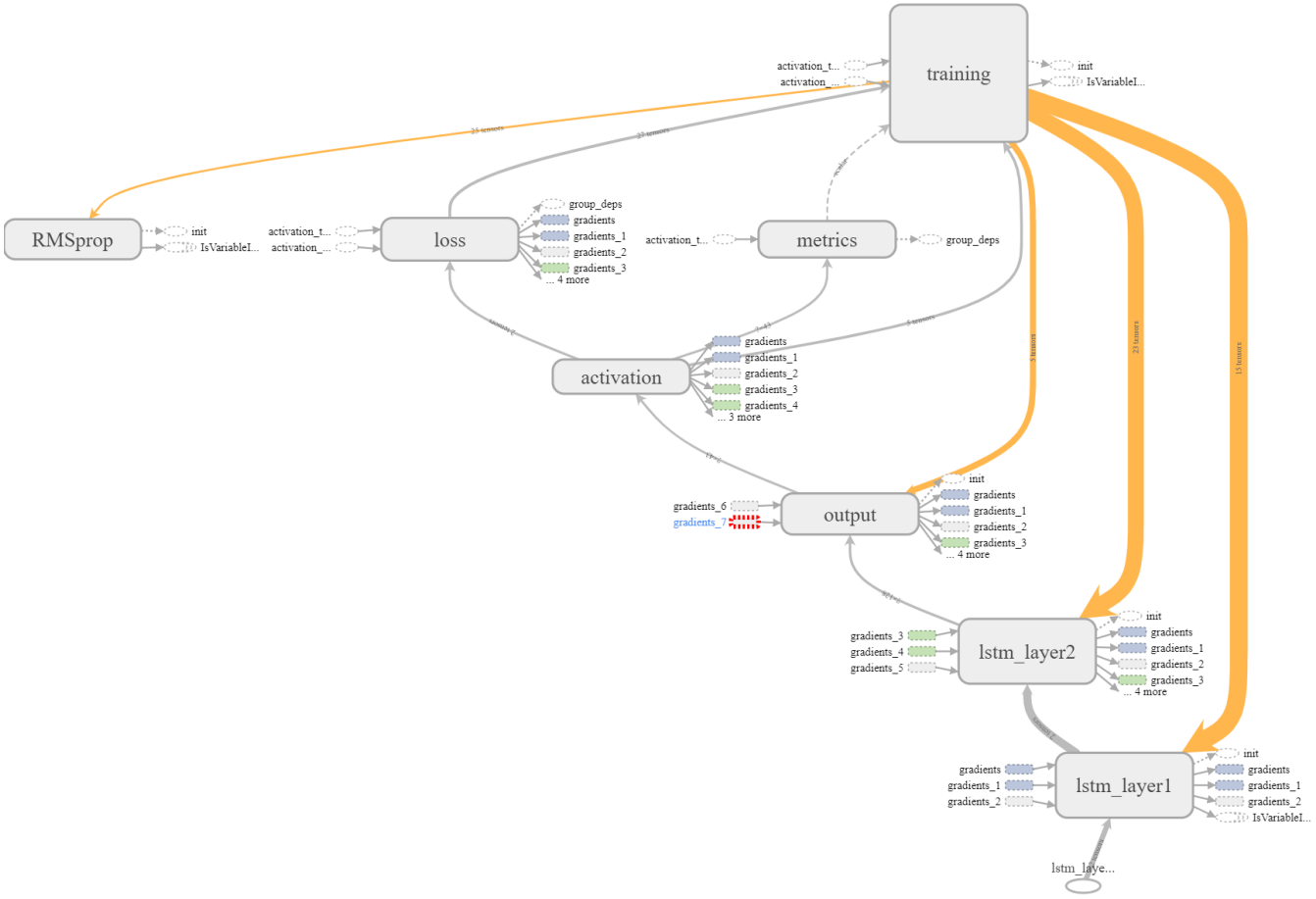
(a) Verlust

(b) Accuracy



Quelle: Eigene Darstellung

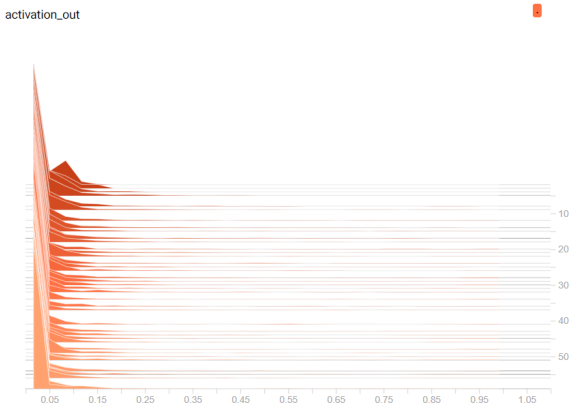
Abbildung 42: Graph model_5



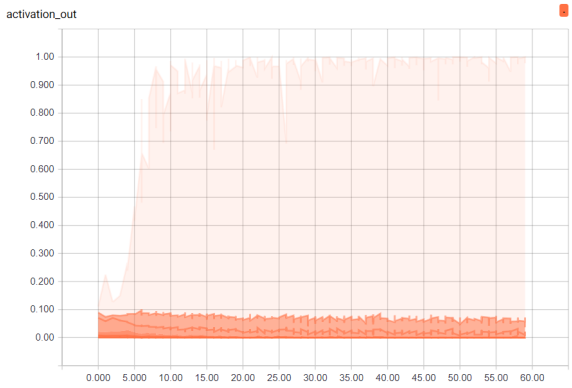
Quelle: Eigene Darstellung

Abbildung 43: Histogramme und Verteilungen der *RNN* Elemente Modell 5

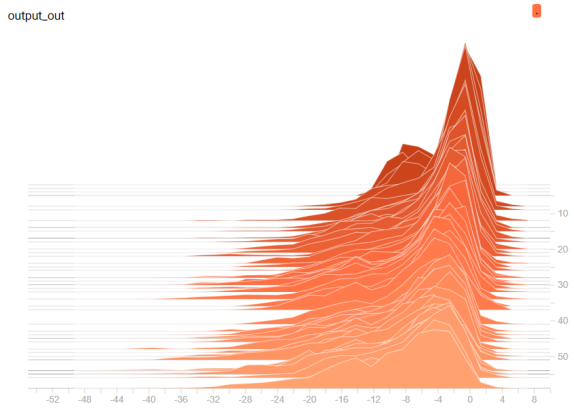
(a) Histogramm *Activation Output*



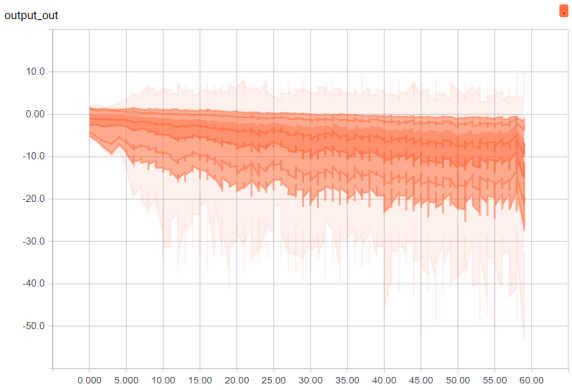
(b) Verteilung *Activation Output*



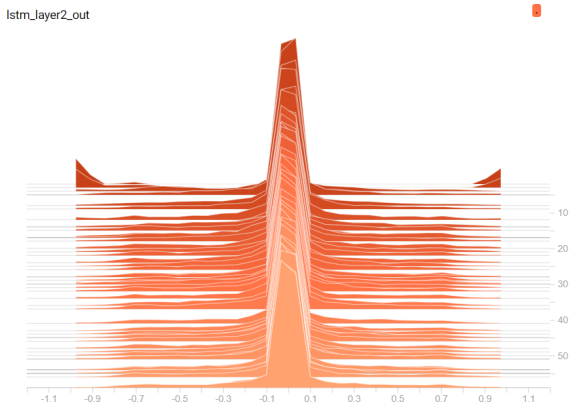
(c) Histogramm *Dense Layer Output*



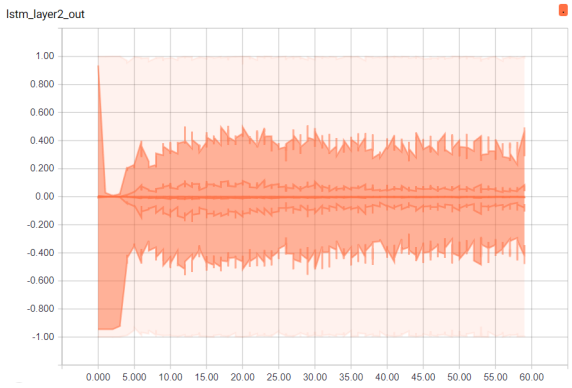
(d) Verteilung *Dense Layer Output*



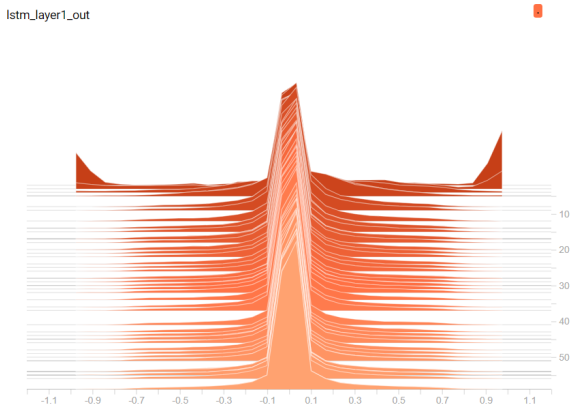
(e) Histogramm *LSTM Layer II Output*



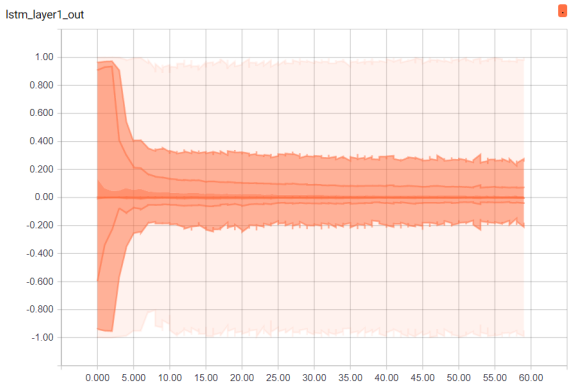
(f) Verteilung *LSTM Layer II Output*



(g) Histogramm *LSTM Layer I Output*



(h) Verteilung *LSTM Layer I Output*

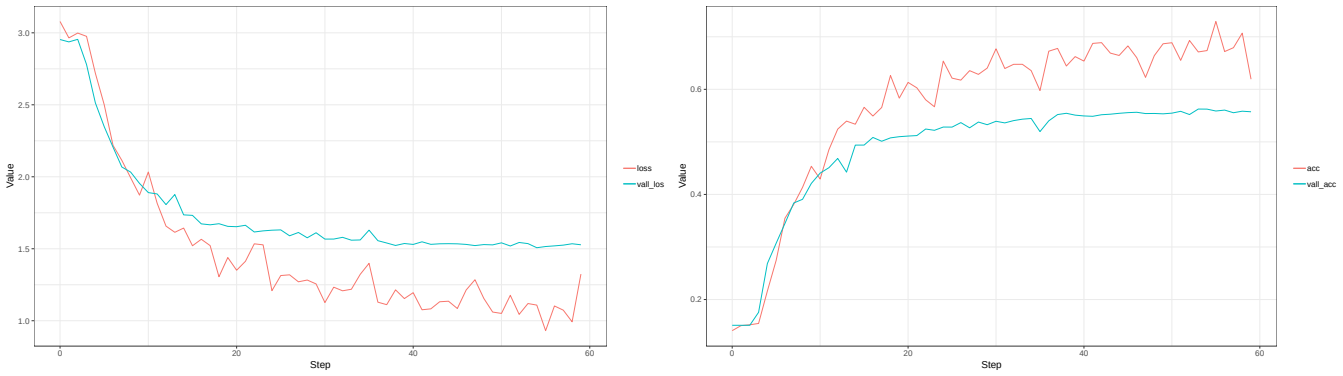


Quelle: Eigene Darstellung

Abbildung 44: Verlust und Accuracy für Trainings und Validierungs Datensatz - model_6

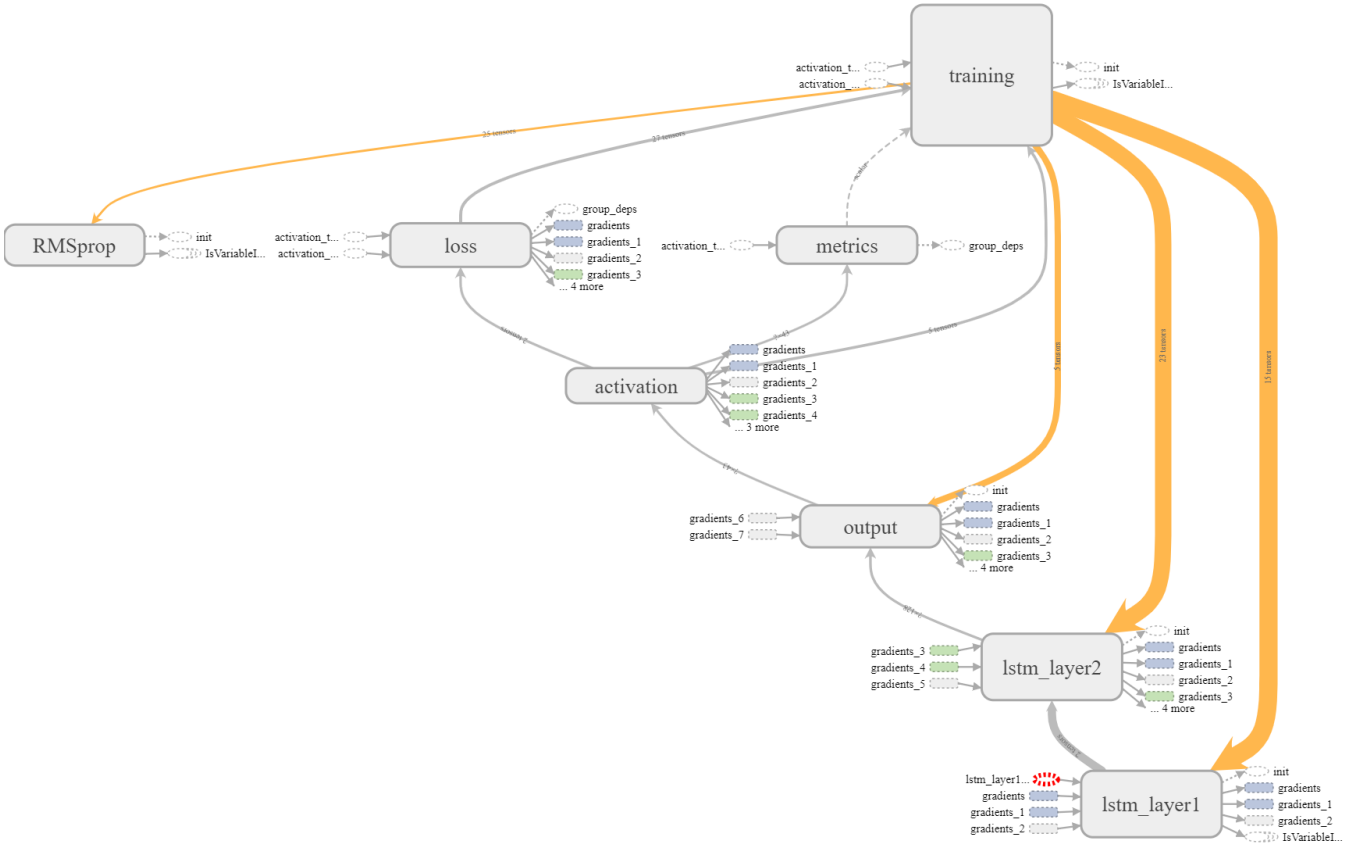
(a) Verlust

(b) Accuracy



Quelle: Eigene Darstellung

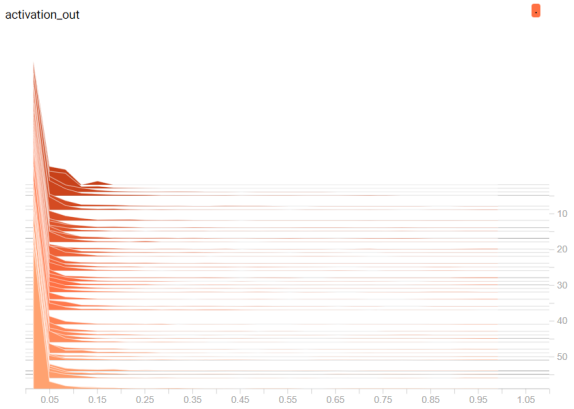
Abbildung 45: Graph model_6



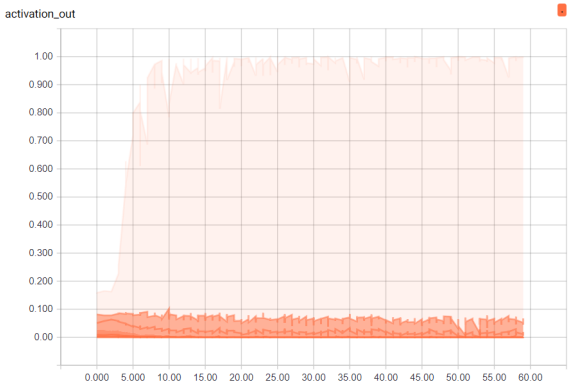
Quelle: Eigene Darstellung

Abbildung 46: Histogramme und Verteilungen der *RNN* Elemente Modell 6

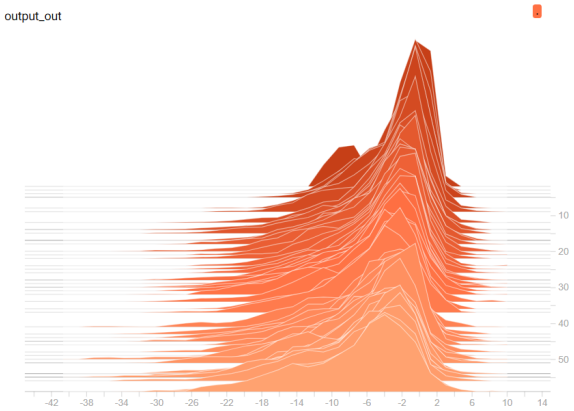
(a) Histogramm *Activation Output*



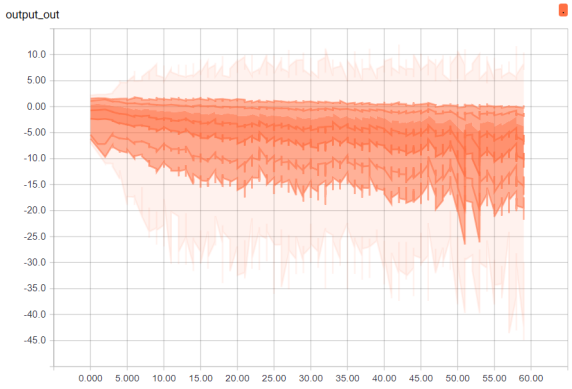
(b) Verteilung *Activation Output*



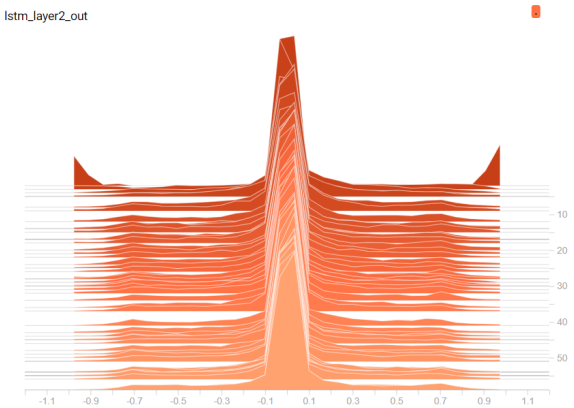
(c) Histogramm *Dense Layer Output*



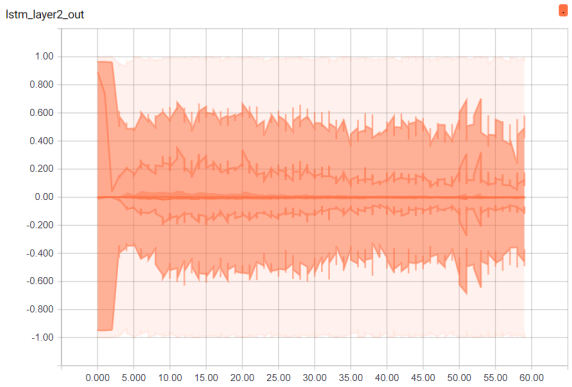
(d) Verteilung *Dense Layer Output*



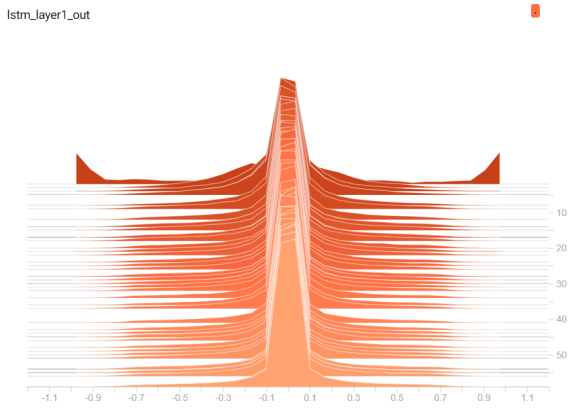
(e) Histogramm *LSTM Layer II Output*



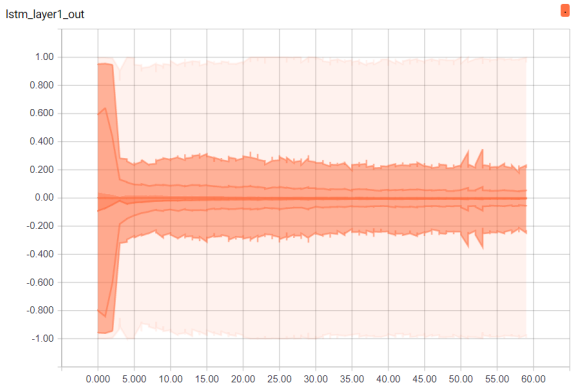
(f) Verteilung *LSTM Layer II Output*



(g) Histogramm *LSTM Layer I Output*



(h) Verteilung *LSTM Layer I Output*

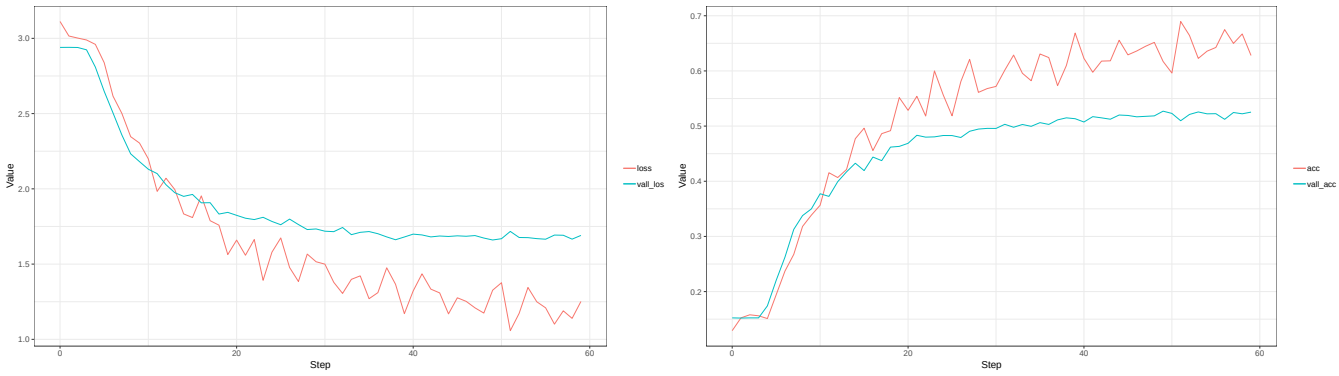


Quelle: Eigene Darstellung

Abbildung 47: Verlust und Accuracy für Trainings und Validierungs Datensatz - model_7

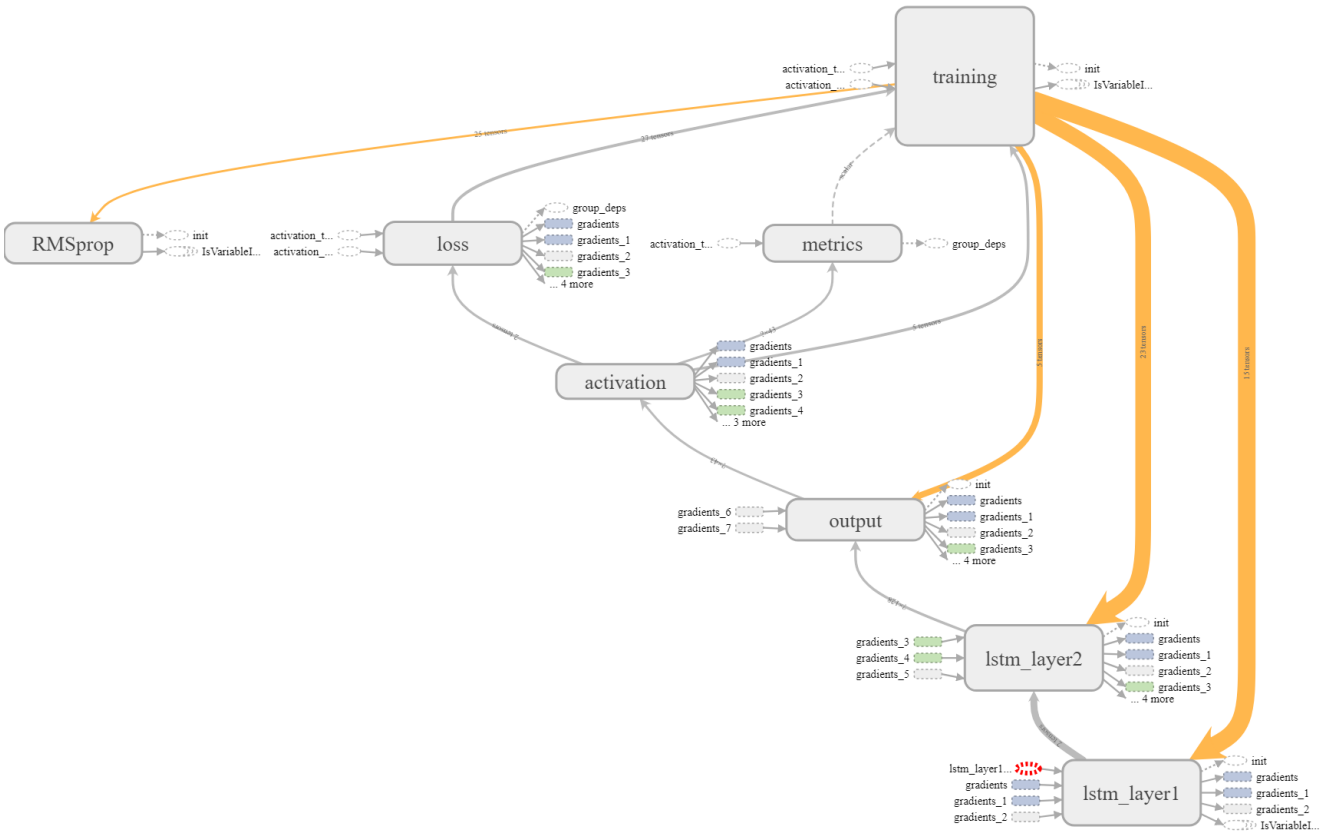
(a) Verlust

(b) Accuracy



Quelle: Eigene Darstellung

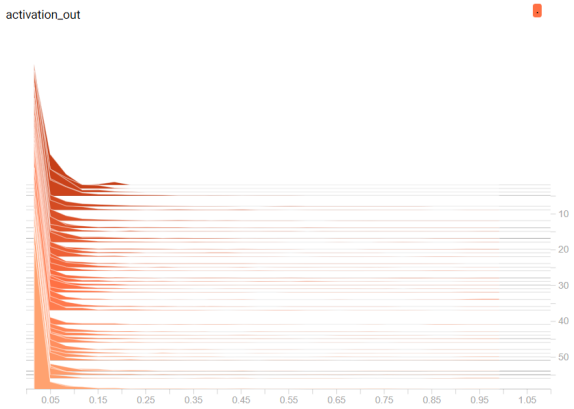
Abbildung 48: Graph model_7



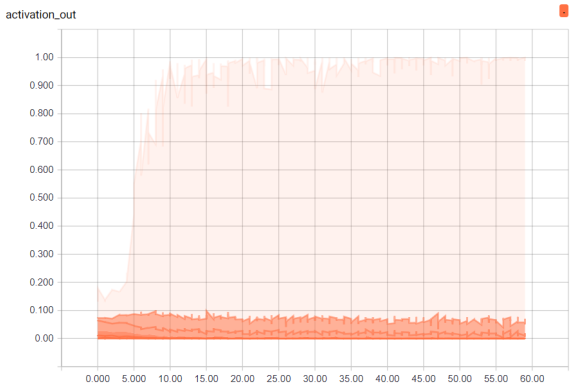
Quelle: Eigene Darstellung

Abbildung 49: Histogramme und Verteilungen der *RNN* Elemente Modell 7

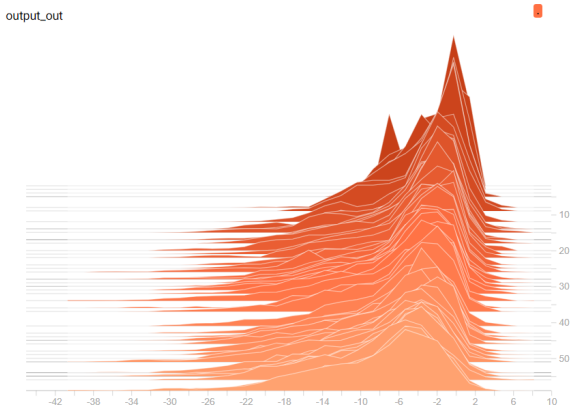
(a) Histogramm *Activation Output*



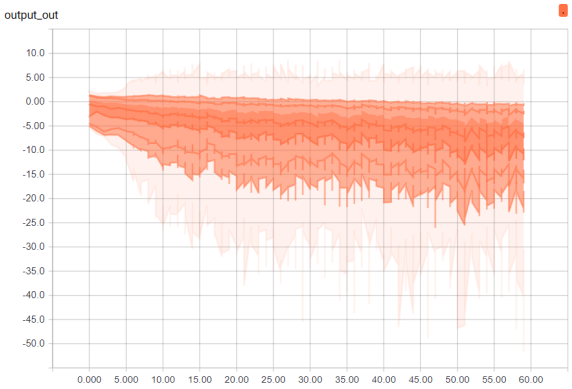
(b) Verteilung *Activation Output*



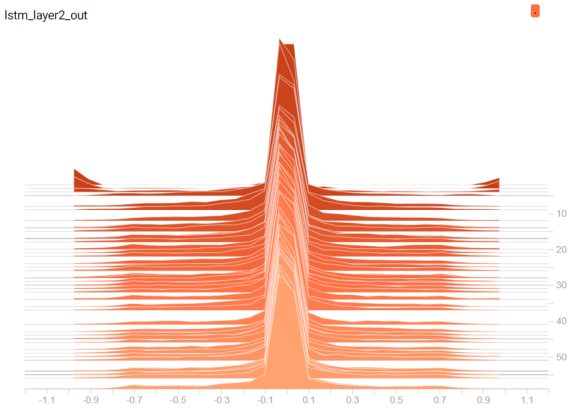
(c) Histogramm *Dense Layer Output*



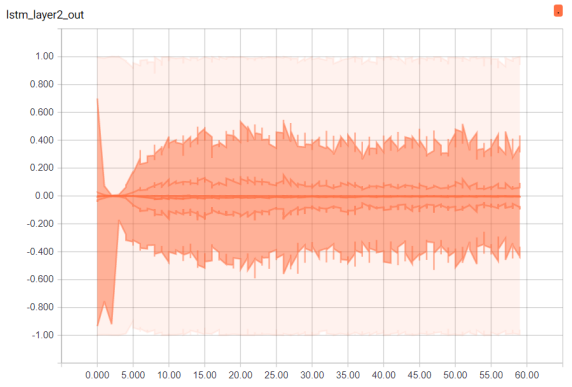
(d) Verteilung *Dense Layer Output*



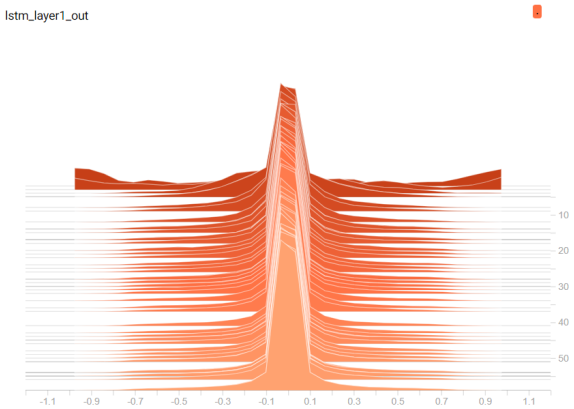
(e) Histogramm *LSTM Layer II Output*



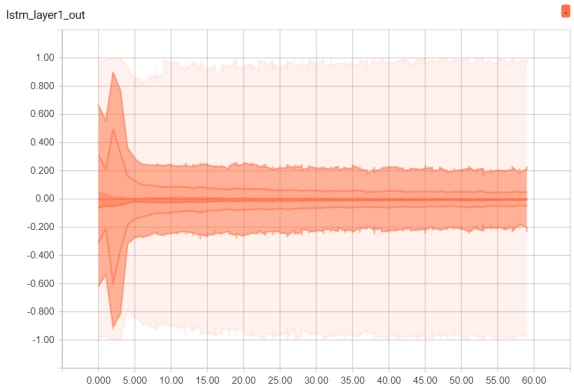
(f) Verteilung *LSTM Layer II Output*



(g) Histogramm *LSTM Layer I Output*



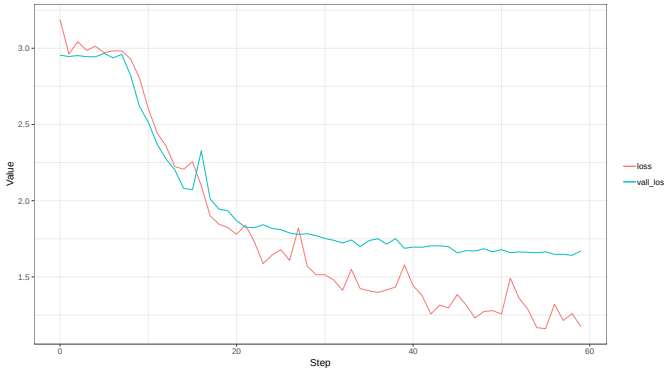
(h) Verteilung *LSTM Layer I Output*



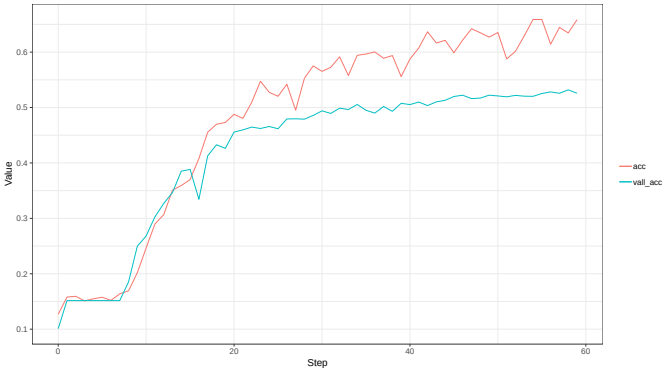
Quelle: Eigene Darstellung

Abbildung 50: Verlust und Accuracy für Trainings und Validierungs Datensatz - model_8

(a) Verlust

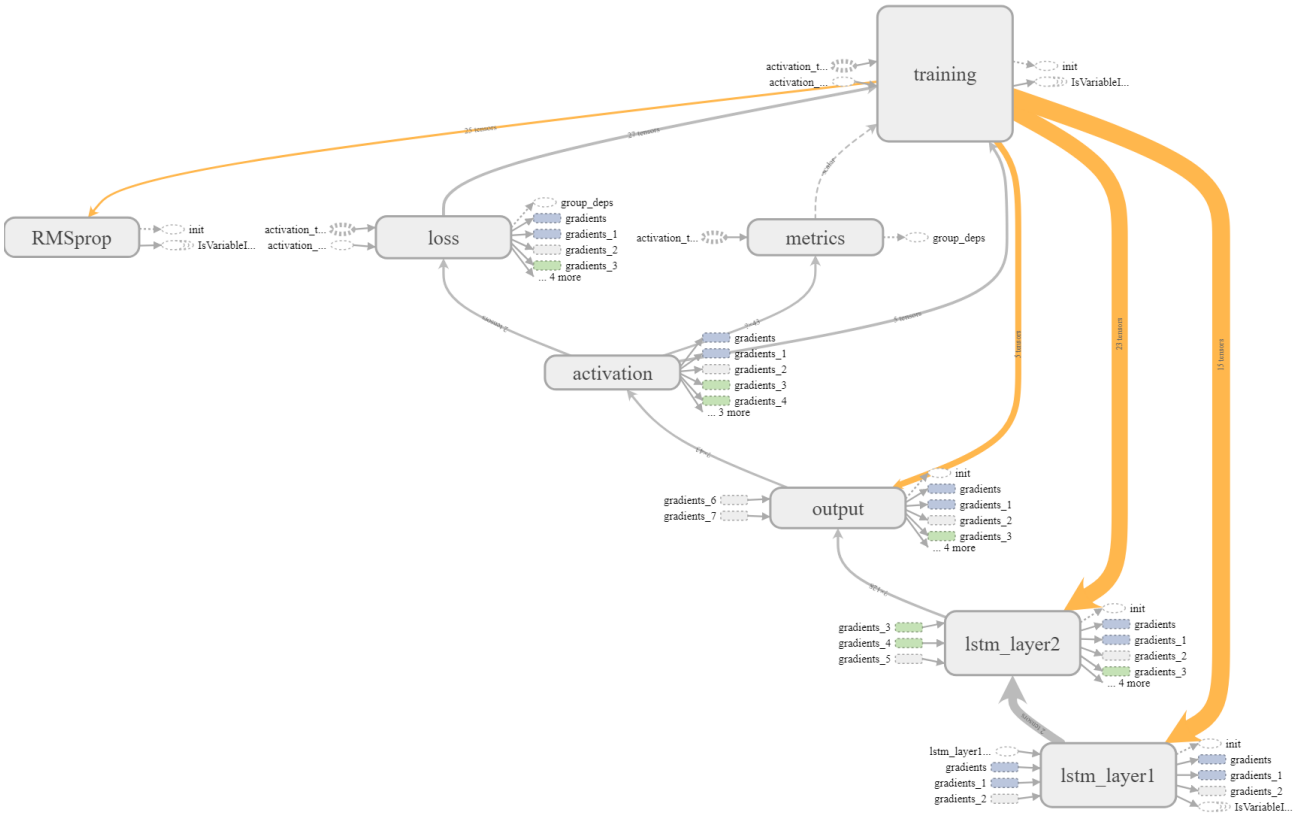


(b) Accuracy



Quelle: Eigene Darstellung

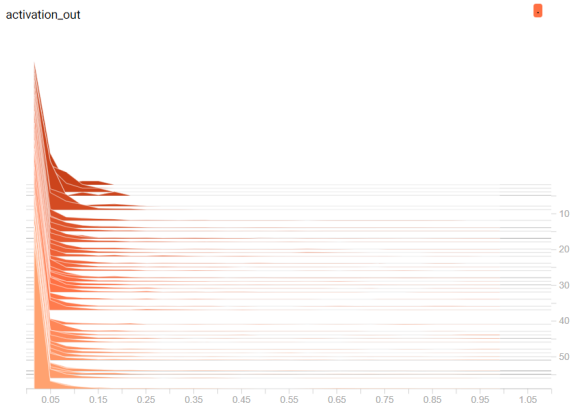
Abbildung 51: Graph model_8



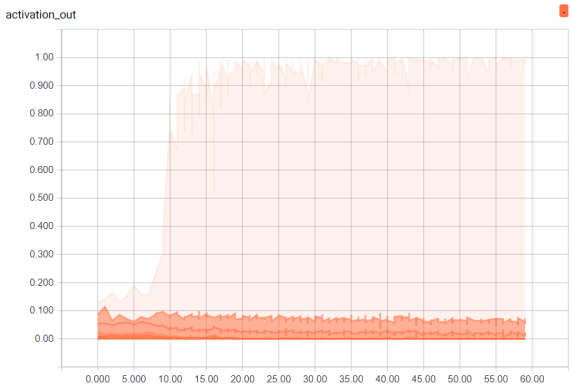
Quelle: Eigene Darstellung

Abbildung 52: Histogramme und Verteilungen der *RNN* Elemente Modell 8

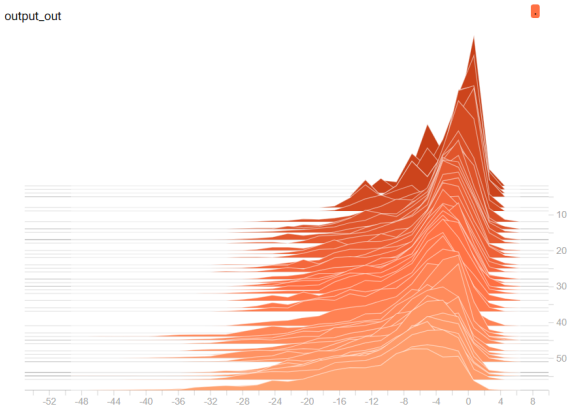
(a) Histogramm *Activation Output*



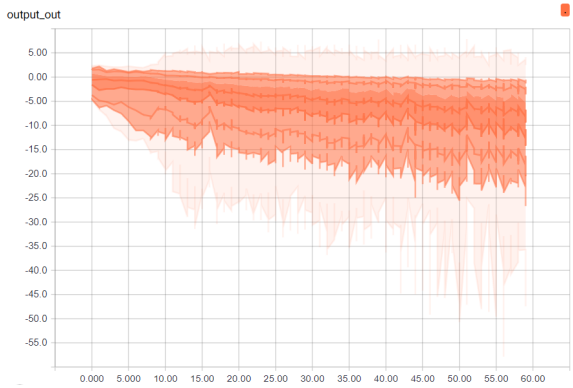
(b) Verteilung *Activation Output*



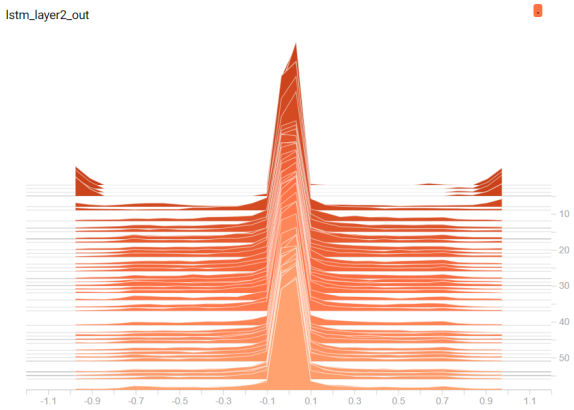
(c) Histogramm *Dense Layer Output*



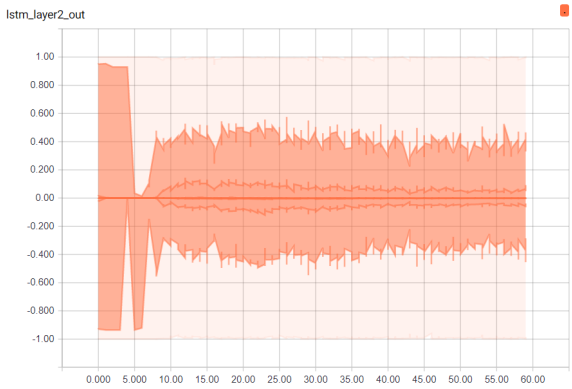
(d) Verteilung *Dense Layer Output*



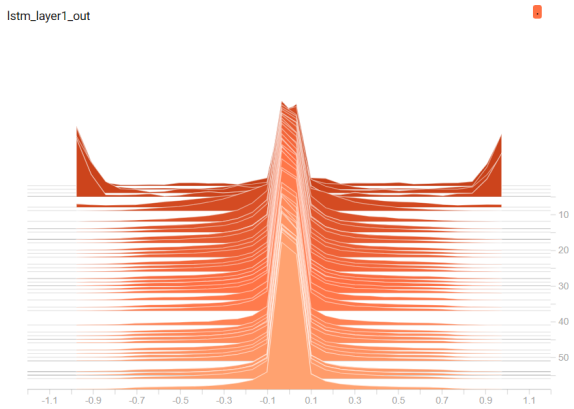
(e) Histogramm *LSTM Layer II Output*



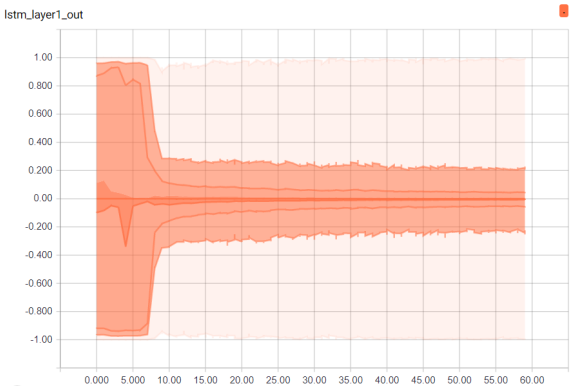
(f) Verteilung *LSTM Layer II Output*



(g) Histogramm *LSTM Layer I Output*



(h) Verteilung *LSTM Layer I Output*

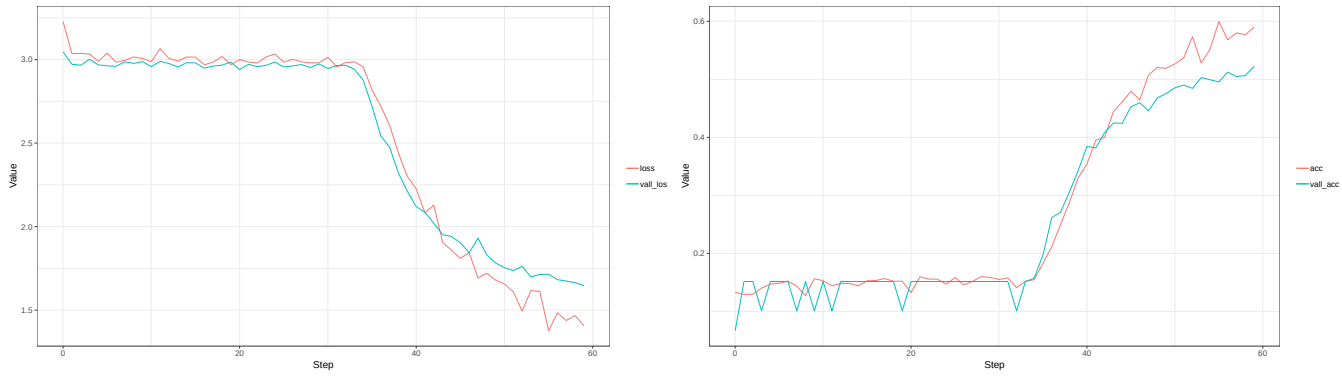


Quelle: Eigene Darstellung

Abbildung 53: Verlust und Accuracy für Trainings und Validierungs Datensatz - model_9

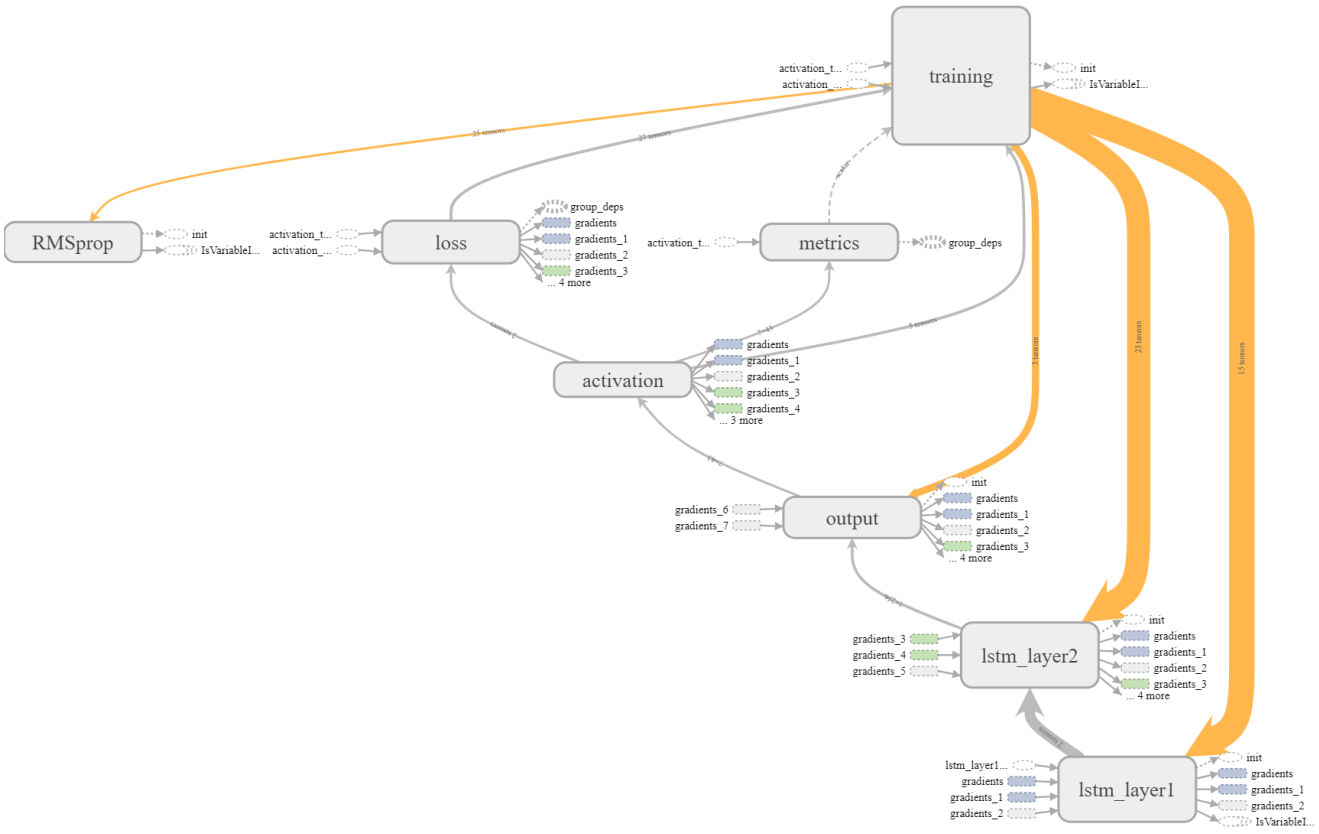
(a) Verlust

(b) Accuracy



Quelle: Eigene Darstellung

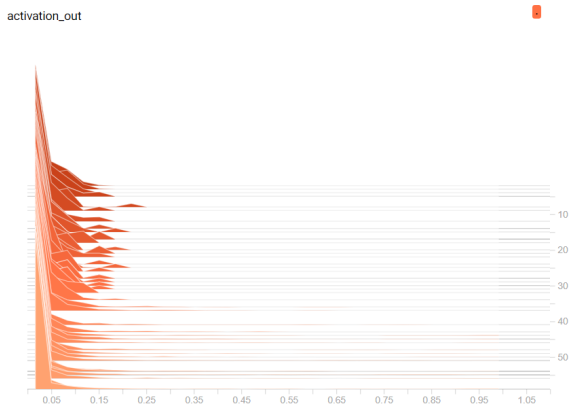
Abbildung 54: Graph model_9



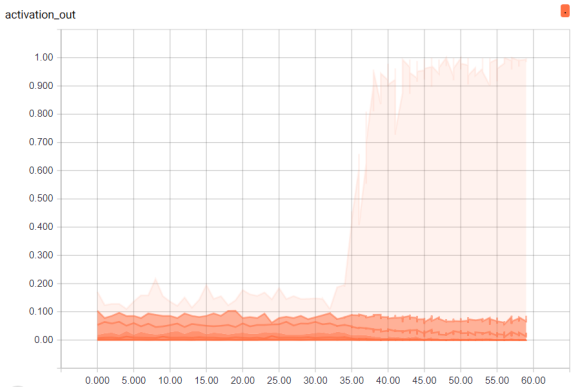
Quelle: Eigene Darstellung

Abbildung 55: Histogramme und Verteilungen der *RNN* Elemente Modell 9

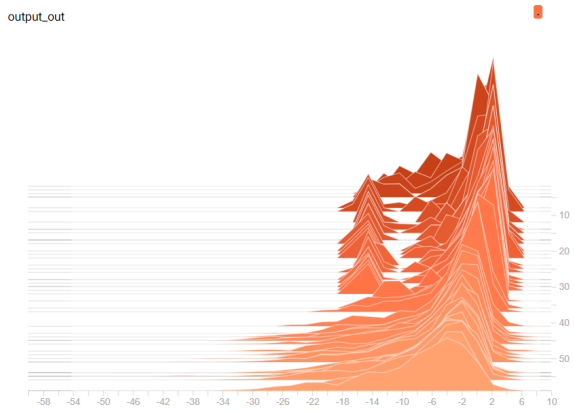
(a) Histogramm *Activation Output*



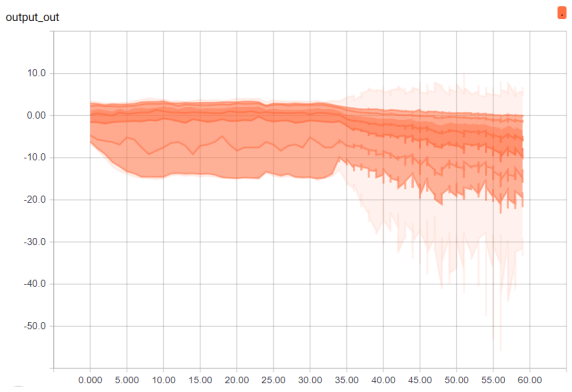
(b) Verteilung *Activation Output*



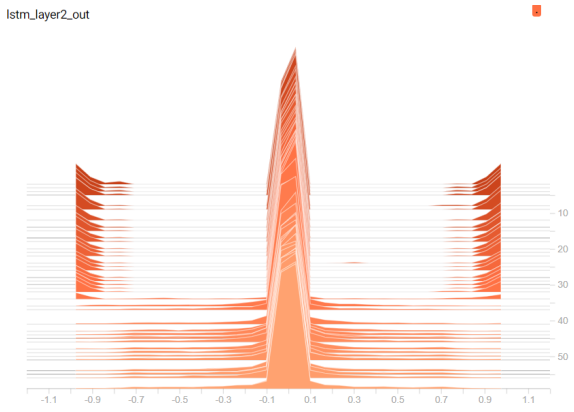
(c) Histogramm *Dense Layer Output*



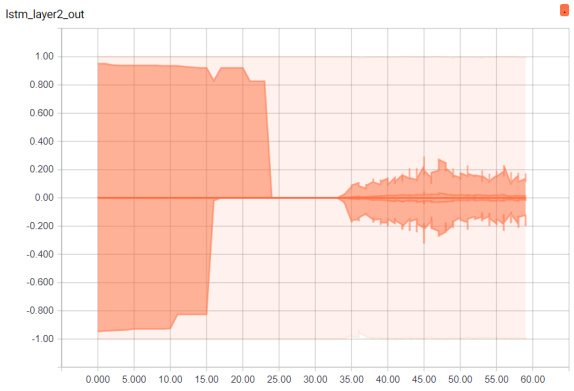
(d) Verteilung *Dense Layer Output*



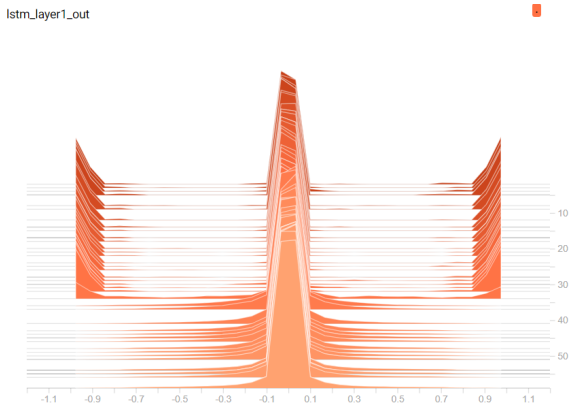
(e) Histogramm *LSTM Layer II Output*



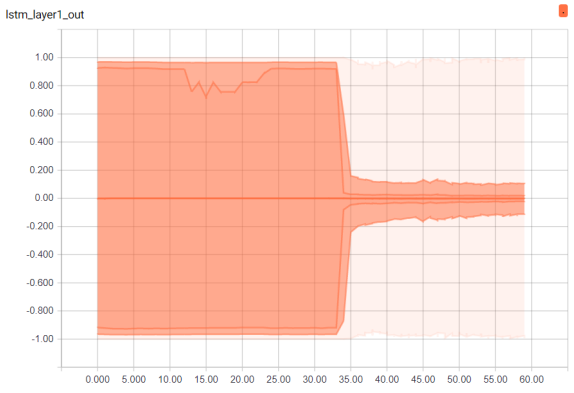
(f) Verteilung *LSTM Layer II Output*



(g) Histogramm *LSTM Layer I Output*



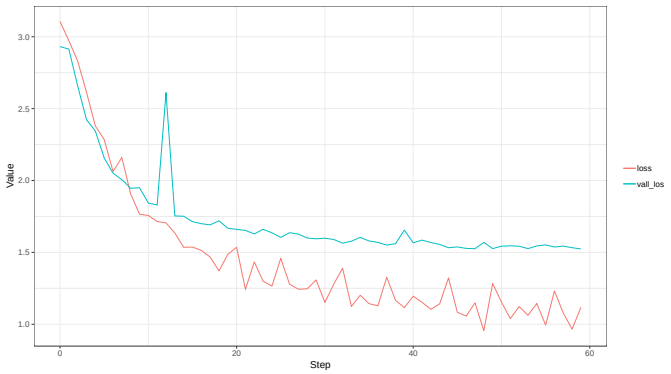
(h) Verteilung *LSTM Layer I Output*



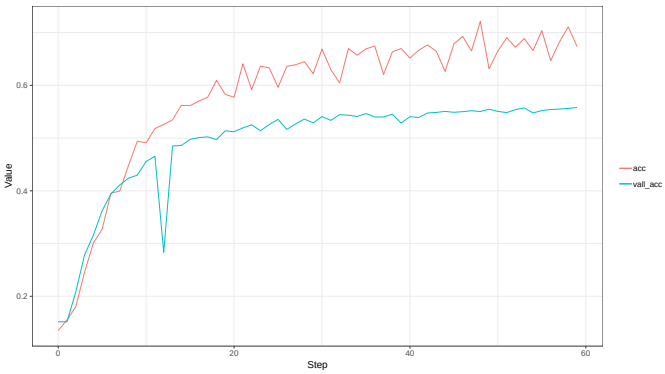
Quelle: Eigene Darstellung

Abbildung 56: Verlust und Accuracy für Trainings und Validierungs Datensatz - model_10

(a) Verlust

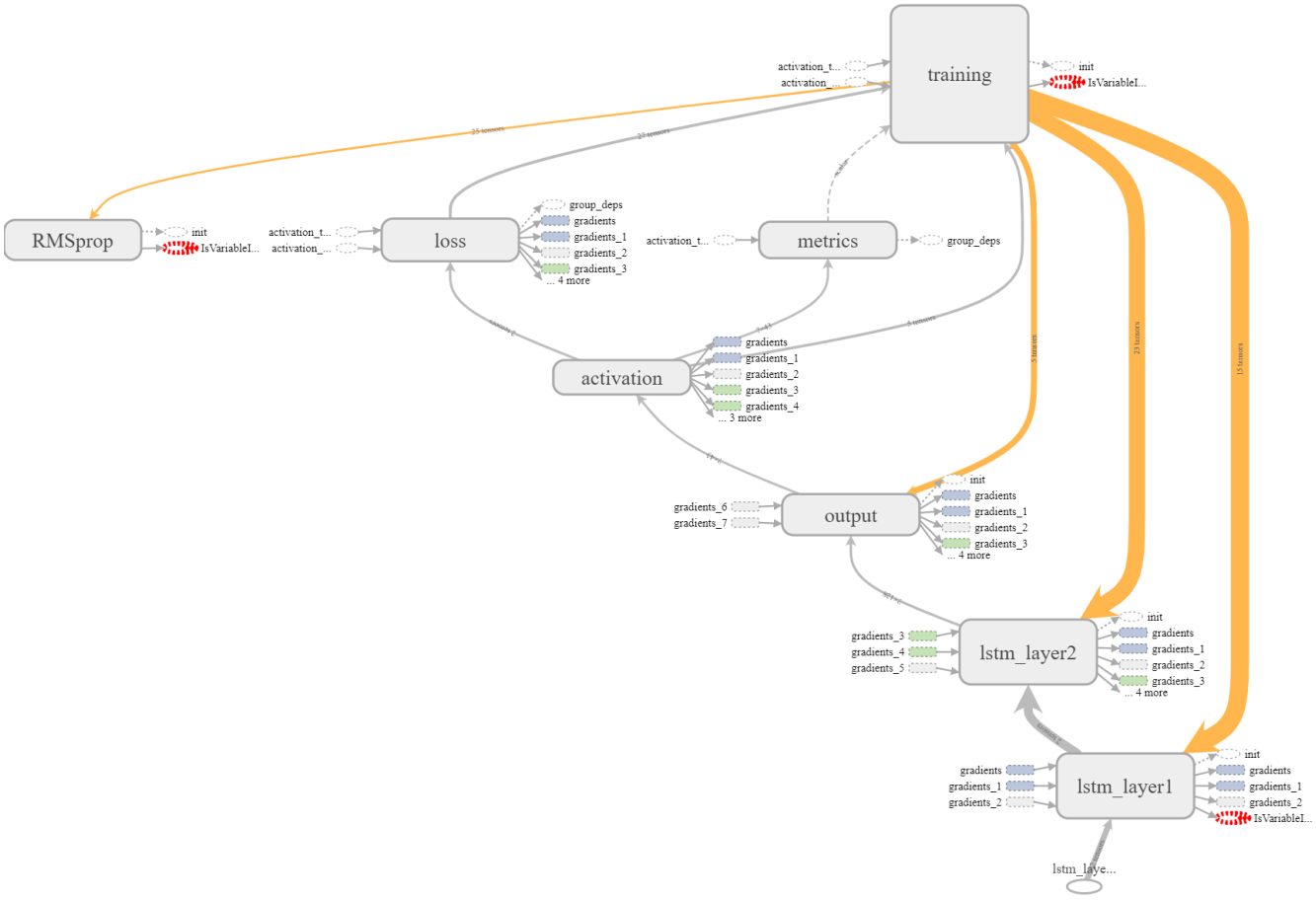


(b) Accuracy



Quelle: Eigene Darstellung

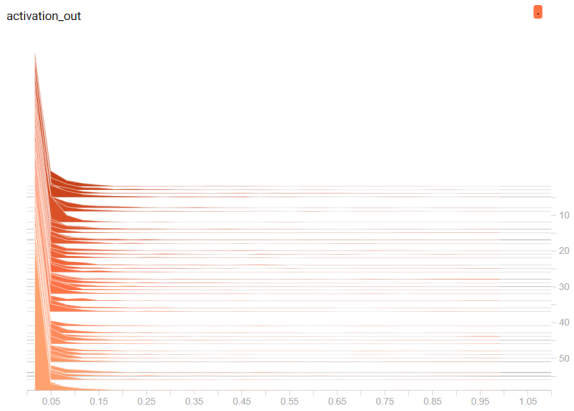
Abbildung 57: Graph model_10



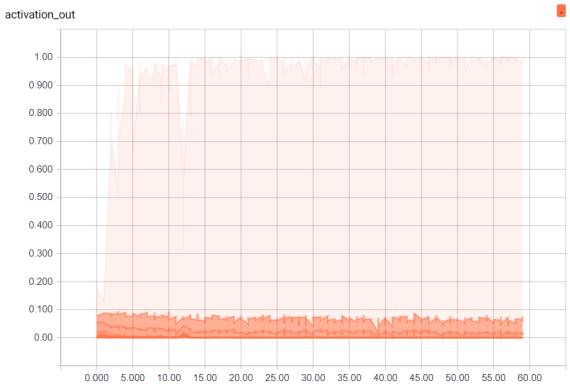
Quelle: Eigene Darstellung

Abbildung 58: Histogramme und Verteilungen der *RNN* Elemente Modell 10

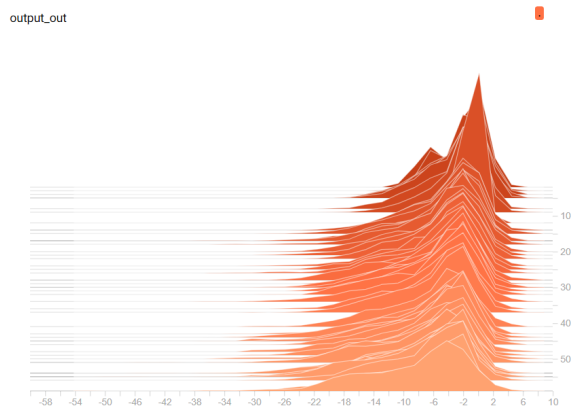
(a) Histogramm *Activation Output*



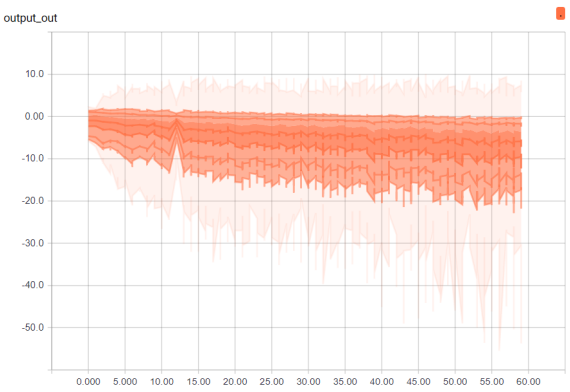
(b) Verteilung *Activation Output*



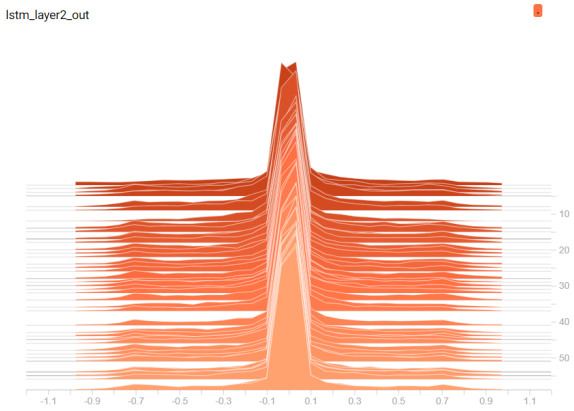
(c) Histogramm *Dense Layer Output*



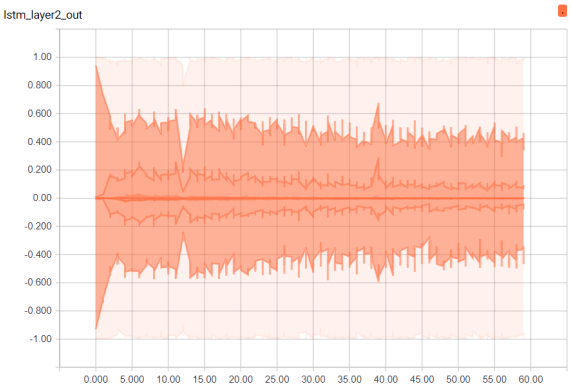
(d) Verteilung *Dense Layer Output*



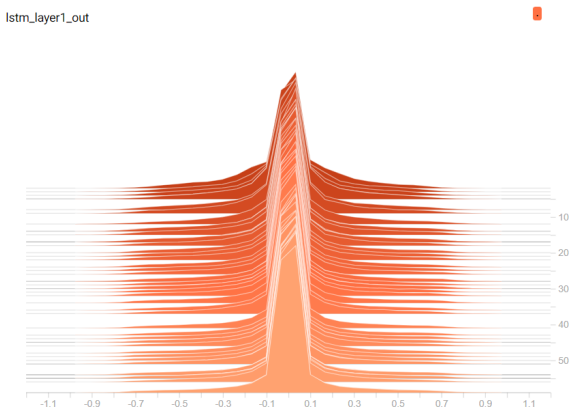
(e) Histogramm *LSTM Layer II Output*



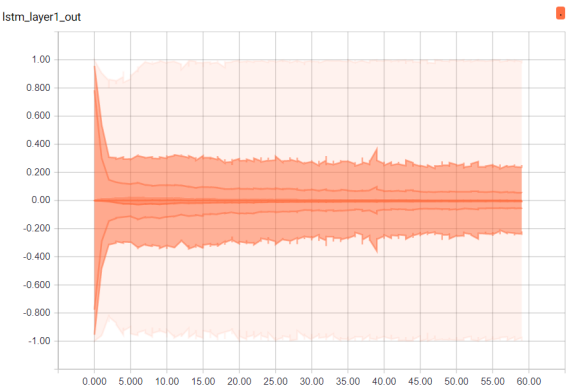
(f) Verteilung *LSTM Layer II Output*



(g) Histogramm *LSTM Layer I Output*



(h) Verteilung *LSTM Layer I Output*

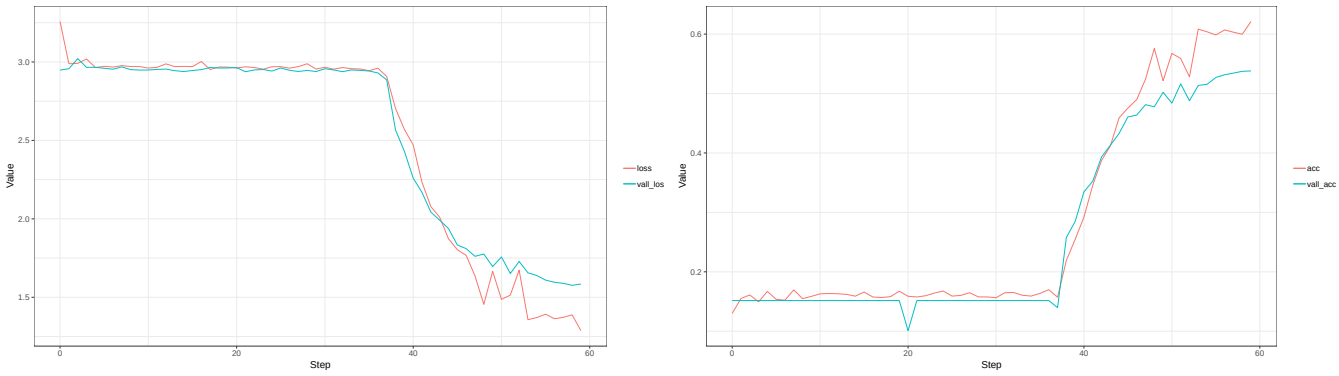


Quelle: Eigene Darstellung

Abbildung 59: Verlust und Accuracy für Trainings und Validierungs Datensatz - model_11

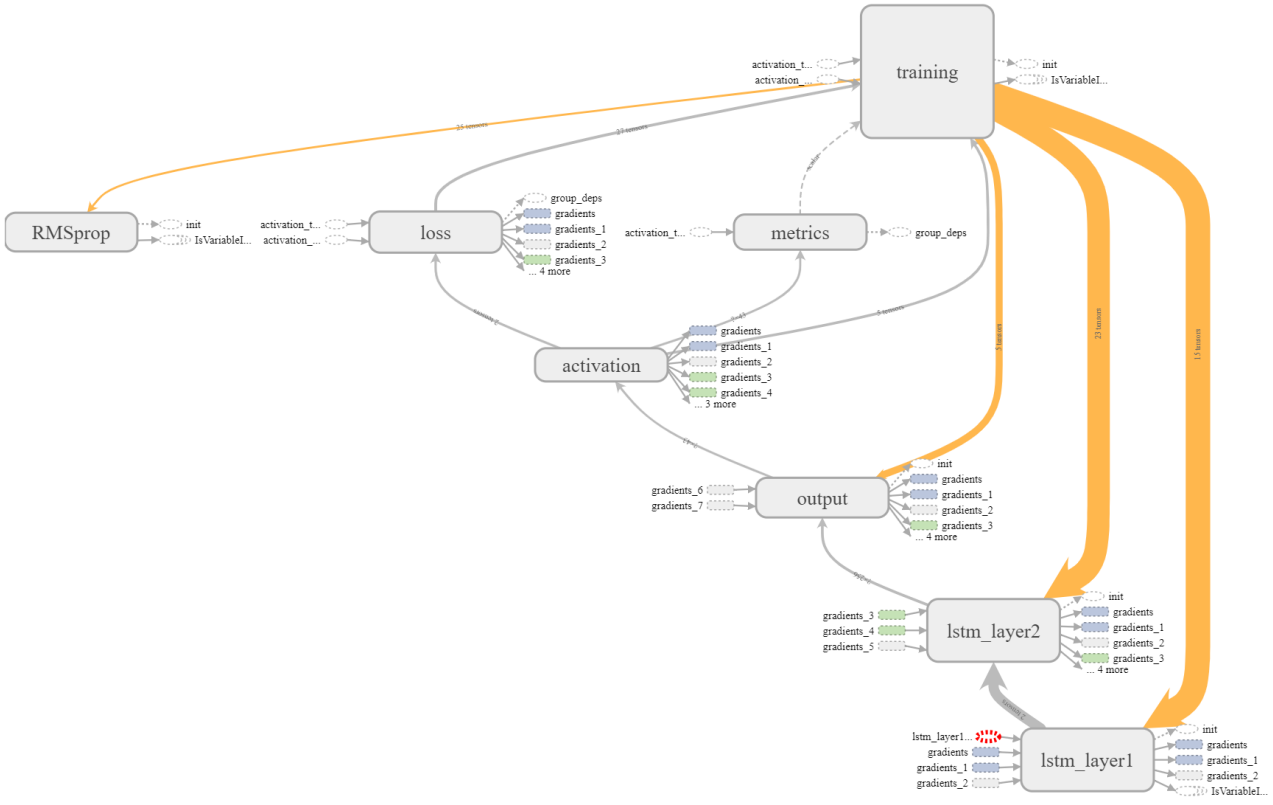
(a) Verlust

(b) Accuracy



Quelle: Eigene Darstellung

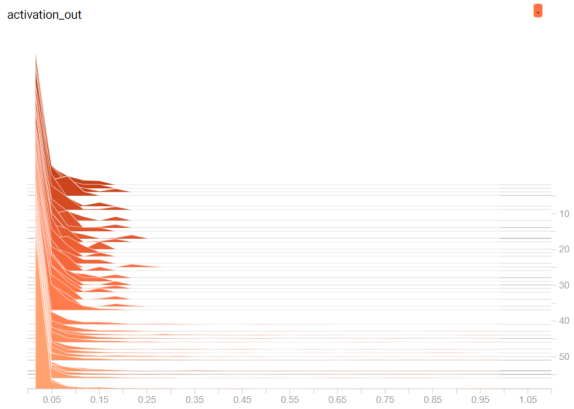
Abbildung 60: Graph model_11



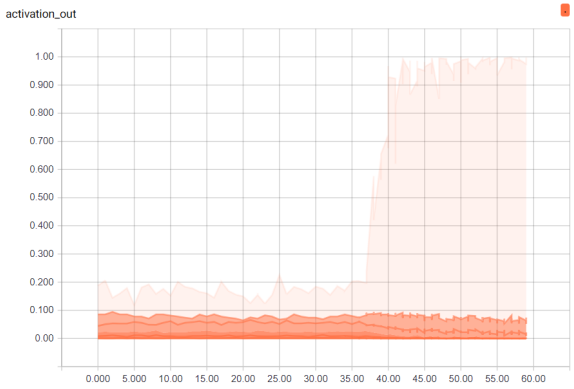
Quelle: Eigene Darstellung

Abbildung 61: Histogramme und Verteilungen der *RNN* Elemente Modell 11

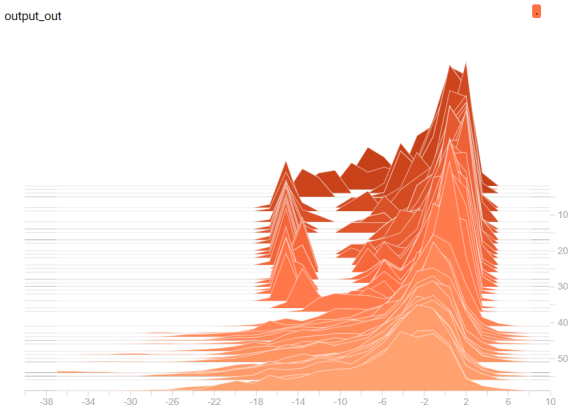
(a) Histogramm *Activation Output*



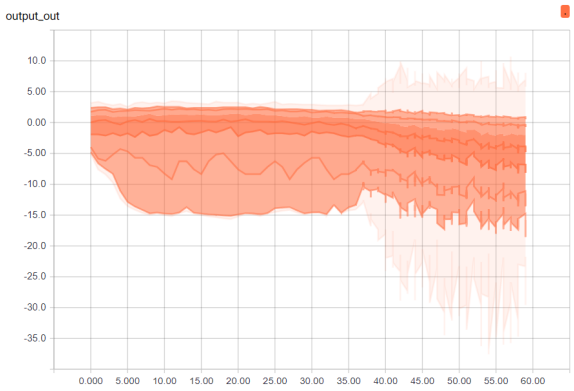
(b) Verteilung *Activation Output*



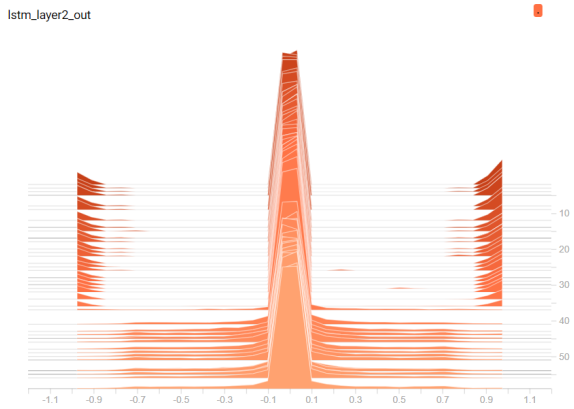
(c) Histogramm *Dense Layer Output*



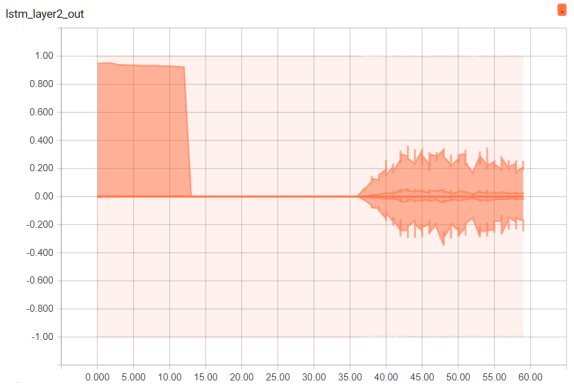
(d) Verteilung *Dense Layer Output*



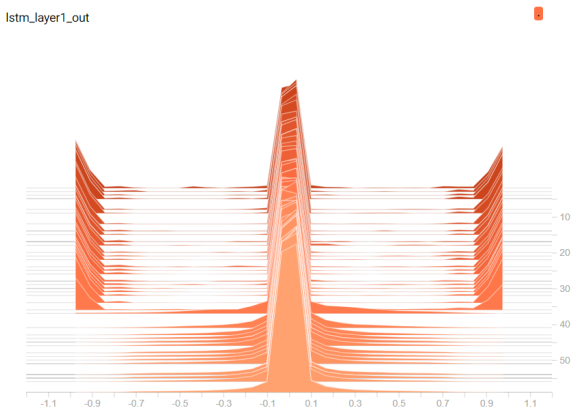
(e) Histogramm *LSTM Layer II Output*



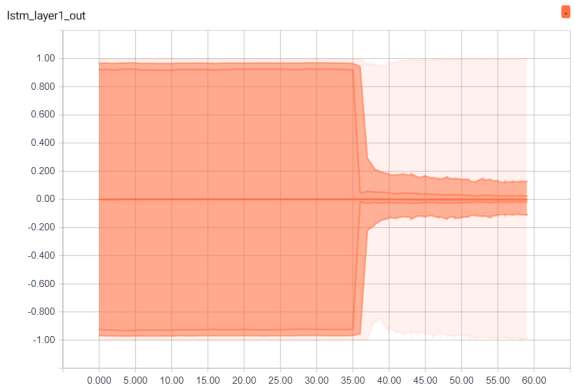
(f) Verteilung *LSTM Layer II Output*



(g) Histogramm *LSTM Layer I Output*



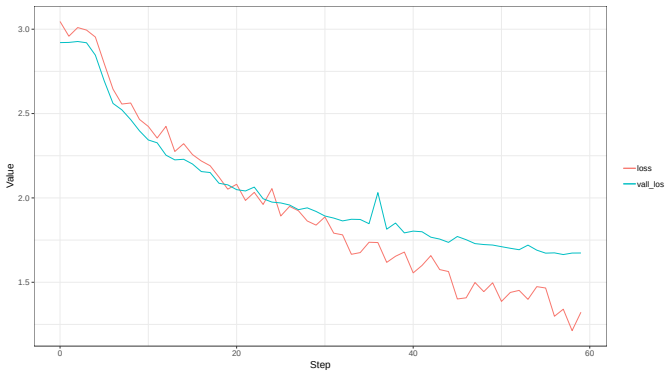
(h) Verteilung *LSTM Layer I Output*



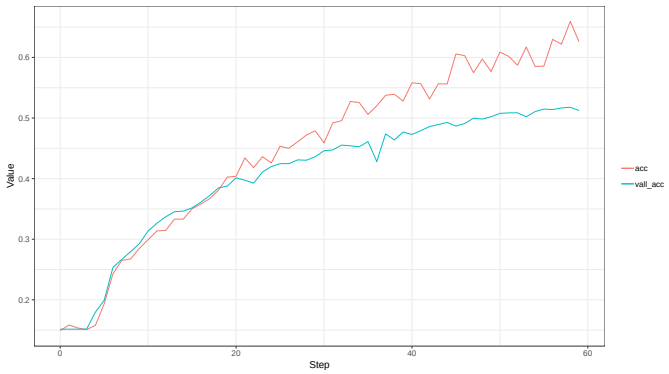
Quelle: Eigene Darstellung

Abbildung 62: Verlust und Accuracy für Trainings und Validierungs Datensatz - model_12

(a) Verlust

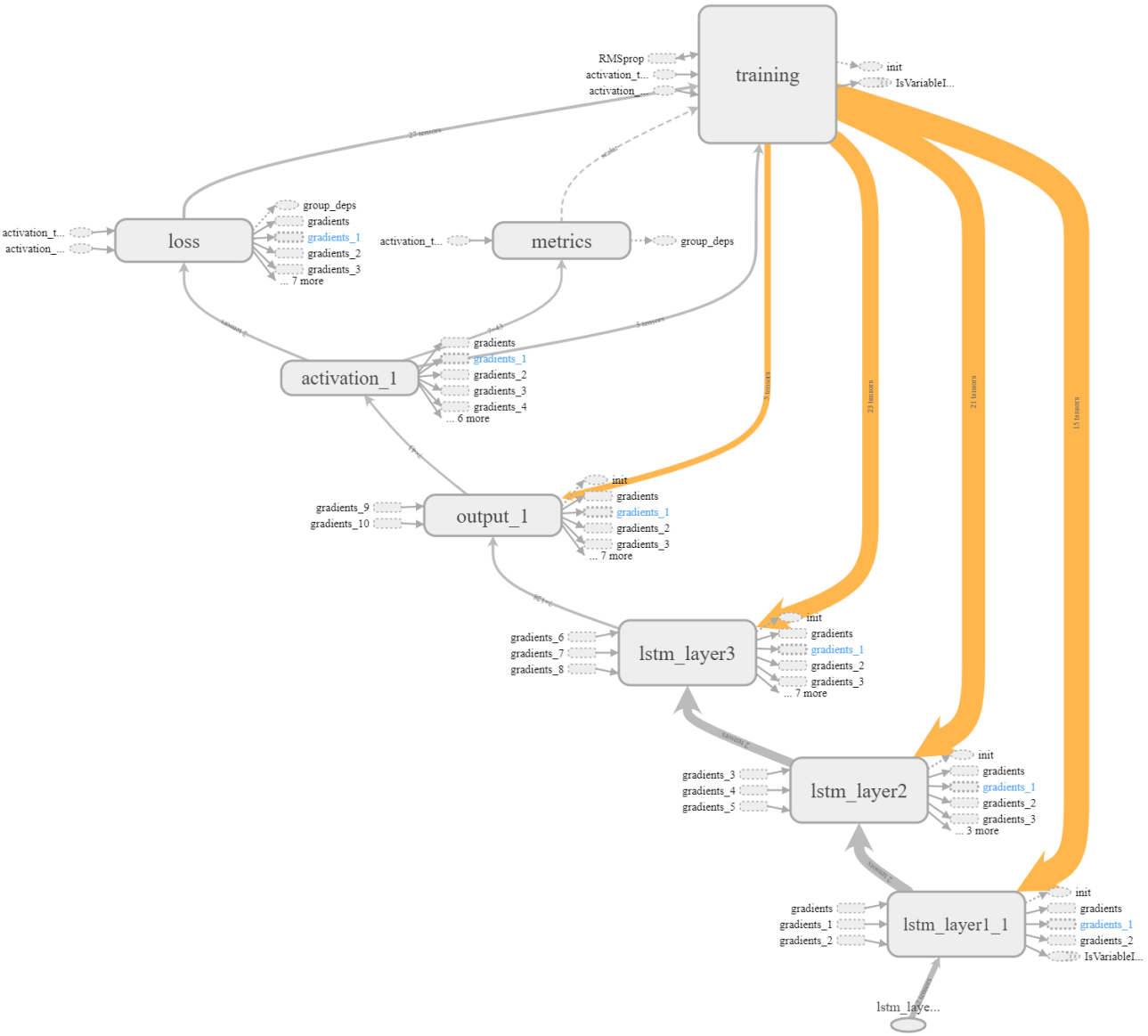


(b) Accuracy



Quelle: Eigene Darstellung

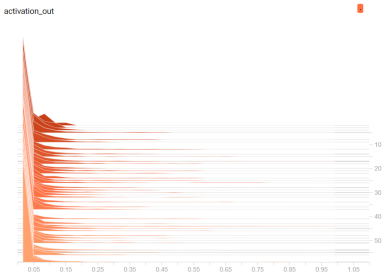
Abbildung 63: Graph model_12



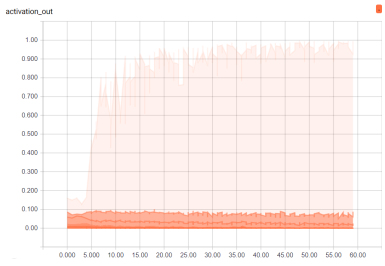
Quelle: Eigene Darstellung

Abbildung 64: Histogramme und Verteilungen der *RNN* Elemente Modell 12

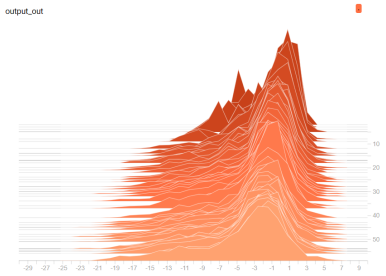
(a) Histogramm *Activation Output*



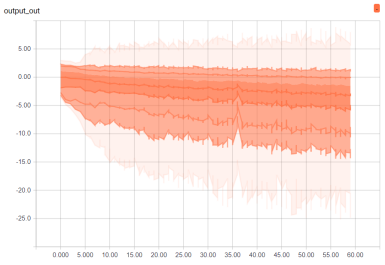
(b) Verteilung *Activation Output*



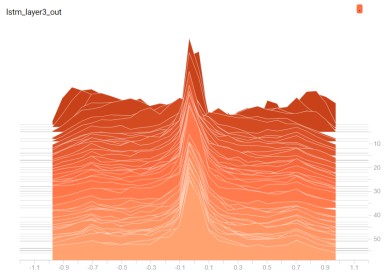
(c) Histogramm *Dense Layer Output*



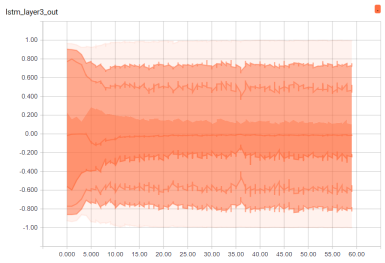
(d) Verteilung *Dense Layer Output*



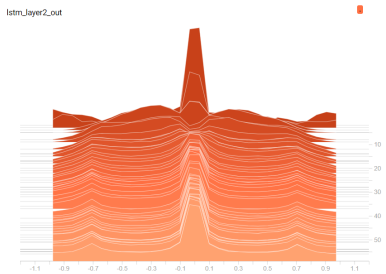
(e) Histogramm *LSTM Layer III Output*



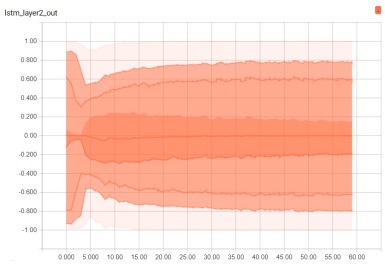
(f) Verteilung *LSTM Layer III Output*



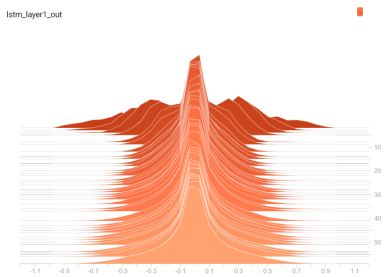
(g) Histogramm *LSTM Layer II Output*



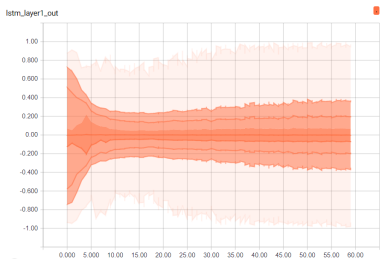
(h) Verteilung *LSTM Layer II Output*



(i) Histogramm *LSTM Layer I Output*



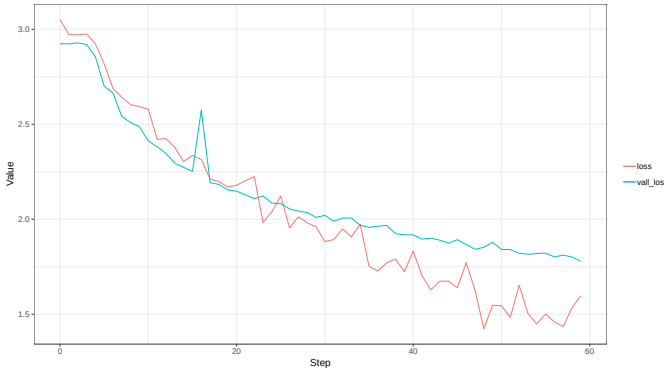
(j) Verteilung *LSTM Layer I Output*



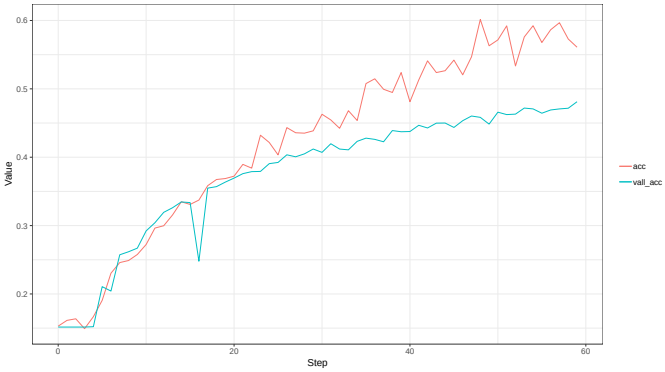
Quelle: Eigene Darstellung

Abbildung 65: Verlust und Accuracy für Trainings und Validierungs Datensatz - model_13

(a) Verlust

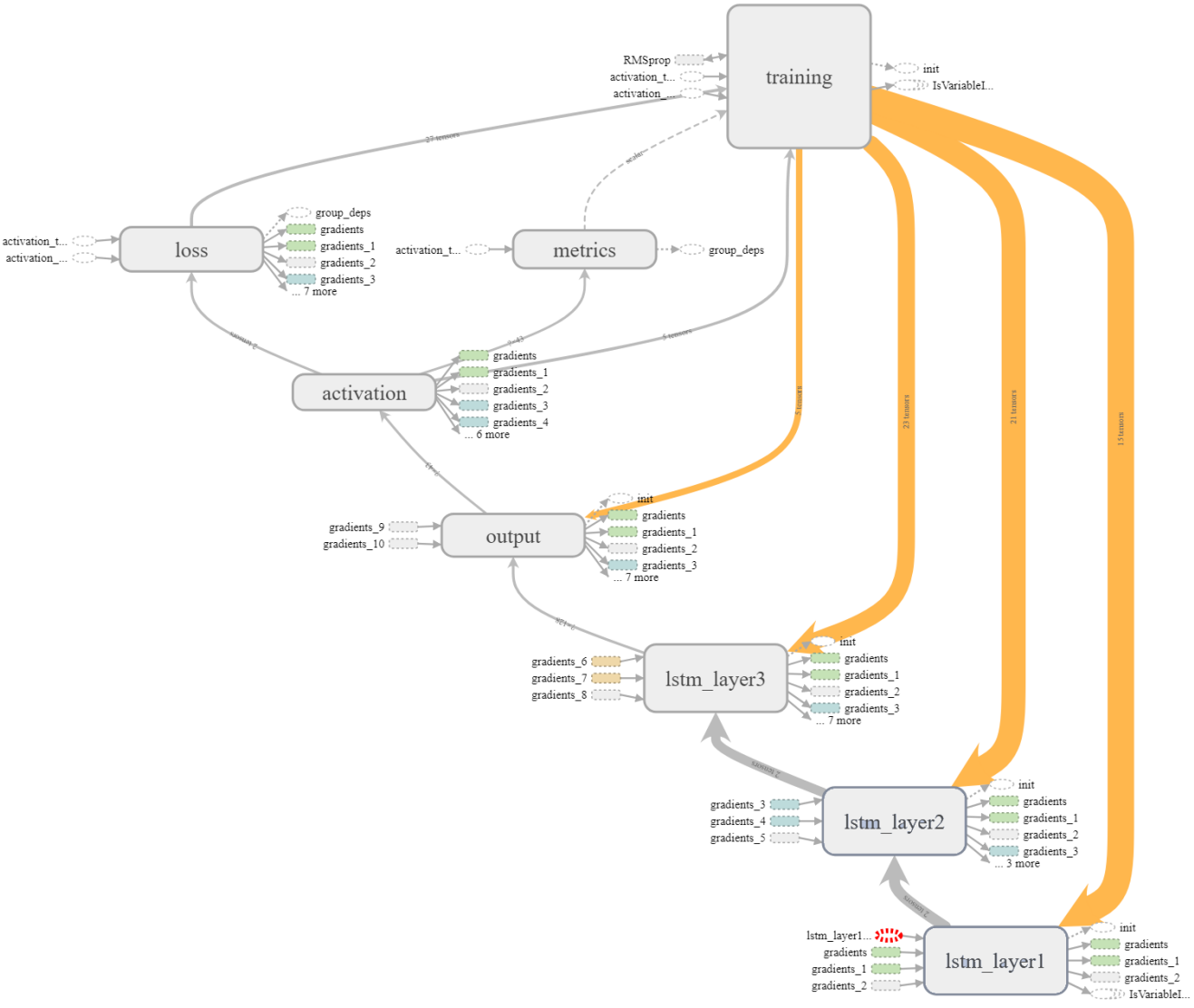


(b) Accuracy



Quelle: Eigene Darstellung

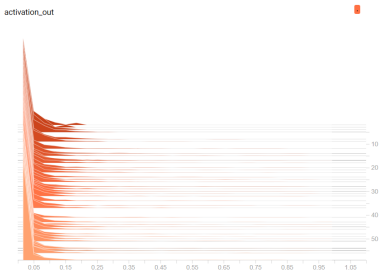
Abbildung 66: Graph model_13



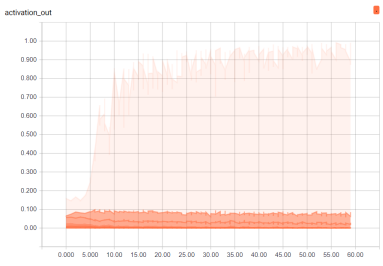
Quelle: Eigene Darstellung

Abbildung 67: Histogramme und Verteilungen der *RNN* Elemente Modell 13

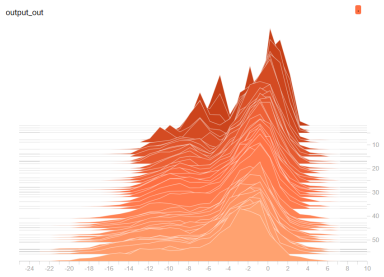
(a) Histogramm *Activation Output*



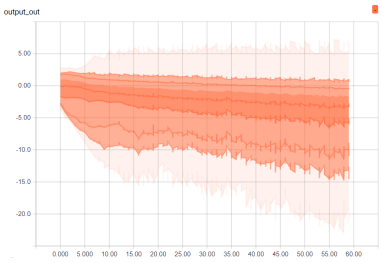
(b) Verteilung *Activation Output*



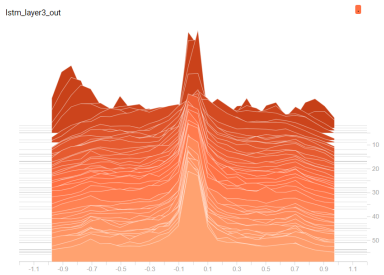
(c) Histogramm *Dense Layer Output*



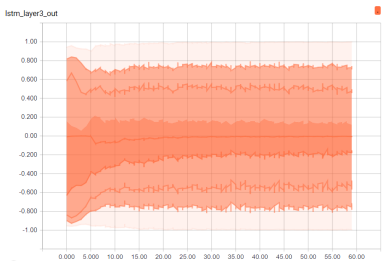
(d) Verteilung *Dense Layer Output*



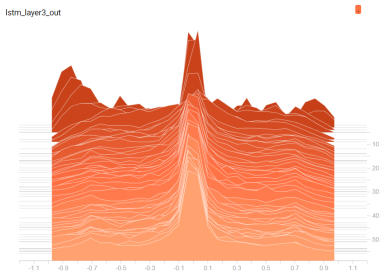
(e) Histogramm *LSTM Layer III Output*



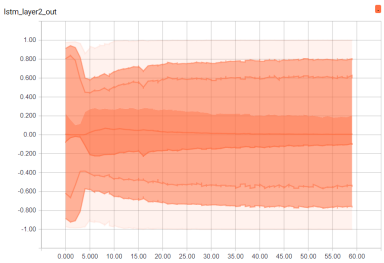
(f) Verteilung *LSTM Layer III Output*



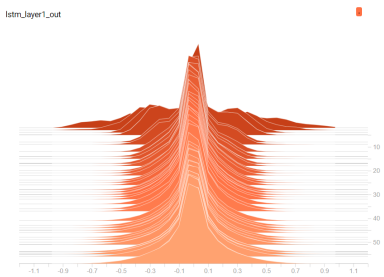
(g) Histogramm *LSTM Layer II Output*



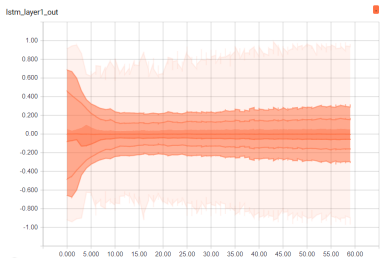
(h) Verteilung *LSTM Layer II Output*



(i) Histogramm *LSTM Layer I Output*



(j) Verteilung *LSTM Layer I Output*



Quelle: Eigene Darstellung

Elektronischer Anhang

1. report_scraper.py
2. pdf2txt_and_textcleaner.R
3. descriptive_statistics4reports.R
4. plot_activation_function.R
5. generator_functions_for_batch_gen.R
6. generate_text_on_epoch_end.R
7. lstm_text_generation_model.R
8. lstm_embedding_example.R