HAW HAMBURG

Bachelorarbeit

Michel Kapell

Generierung von Gedichten auf Basis von rekurrenten neuronalen Netzen

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Michel Kapell

Generierung von Gedichten auf Basis von rekurrenten neuronalen Netzen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Michael Neitzke
Zweitgutachter: Prof. Dr.-Ing. Marina Tropmann-Frick

Eingereicht am: 14. November 2019

Michel Kapell

Thema der Arbeit

Generierung von Gedichten auf Basis von rekurrenten neuronalen Netzen

Stichworte

Gedichte, RNN, LSTM, GRU, Sequenzen

Kurzzusammenfassung

Diese Arbeit beschäftigt sich mit der Generierung von Haikus durch rekurrente neuronale Netze. Haikus sind japanische Kurzgedichte. Verwendet wurden RNN, LSTM und GRU Netze. Für das Training wurden die vorgegebenen Gedichte vorverarbeitet. Anschließend wurden die Netze trainiert. Nach dem Training wurden verschiedene Gedichte generiert und diese miteinander verglichen. Es hat sich gezeigt, dass alle Netze fähig waren Gedichte zu generieren. Lediglich die Dauer des Trainings und das Ergebnis wichen voneinander ab.

Michel Kapell

Title of Thesis

Generating poems with recurrent neural networks

Keywords

poems, RNN, LSTM, GRU, sequenzes

Abstract

The thesis deals with generating haikus using recurrent neural networks. Haikus are short japanese poems. The poems were generated with rnn, lstm and gru networks. For the training human written poems were preprocessed. The networks were trained with these preprocessed poems. After the training the networks generated various poems. These generated poems were compared with each other. All networks were able to generate poems. Just the duration time and the quality of the poems differ from each other.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Rekurrente neuronale Netze (RNN) wurden bereits auf eine Menge von Problemstellungen angewandt, wie z.B. das Erzeugen von Geschäftsberichten[1], Generierung von Hashtags[2] oder das Generieren von Musiksequenzen[3]. All diese Problemstellungen haben gemeinsam, dass sie Sequenzen als Eingaben bekommen. RNN sind für solche Problemstellungen angepasst, sodass sie diese besonders gut lösen können. Besondere Varianten des RNN sind die Long-Short-Term-Memory (LSTM) und Gated Recurrent Units (GRU) neuronalen Netze. Diese haben die Fähigkeit Langzeitabhängigkeiten in Sequenzen zu erlernen. In dieser Arbeit geht es darum, mittels neuronaler Netze, Haiku-Gedichte zu generieren. *„Traditionelles japanisches Haiku besteht aus drei Zeilen, die aus siebzehn Silben gebaut sind. Die besondere innere Melodie der Zeilen folgt daraus, dass die erste Zeile fünf, die zweite sieben und die dritte wieder fünf Silben hat.“*[4] Für die Generierung sollen RNN, LSTM und GRU genutzt werden.

1.1 Zielsetzung

Ziel der Arbeit ist es, durch RNN, LSTM und GRU erfolgreich Haiku-Gedichte zu generieren. Des Weiteren soll ein Vergleich zwischen den drei Netztypen geschaffen werden, welcher unter anderem die Genauigkeit, Trainingsdauer und Verwendung von Ressourcen beinhaltet. Hierfür soll ein Algorithmus zur Erzeugung von Wörterbüchern, sowie einer für das Trainieren der Netze entwickelt/umgesetzt werden.

1.2 Aufbau der Arbeit

Die Arbeit beginnt mit der Einleitung, in welcher ein kurzer Überblick über das behandelte Thema geschaffen wird. Weitergehend werden in der Einleitung das Ziel und der Aufbau der Arbeit erläutert. Abgeschlossen wird die Einleitung mit der Aufzählung von

verwandten Arbeiten, welche sich mit ähnlichen Problemstellungen beschäftigt haben. Auf die Einleitung folgt ein Theorie-Teil, welcher mit den Arten von Netzen beginnt. Dort werden die hier verglichenen Netzarten erklärt. Die Erklärungen beinhalten, den Aufbau der Neuronen der einzelnen Netzarten und die Besonderheiten dieser. Anschließend folgt ein Kapitel, in dem die verwendeten Gedichte so wie die Vorverarbeitung der Gedichte beschrieben wird. Nach der Beschreibung der Trainingsdaten kommt ein Teil, welcher das Training und Verifizieren erläutert. Im Anschluss auf den Theorie-Teil folgt die Umsetzung, welche die Implementation der im Theorie-Teil beschrieben Konzepte zeigt. Auf die Umsetzung folgt die Erläuterung welche Experimente durchgeführt wurden. Nach der Vorstellung der Experimente werden die Ergebnisse ausgewertet. Abgeschlossen wird die Arbeit mit einem Fazit, welches alle Ergebnisse und Auffälligkeiten zusammen fast und reflektiert.

1.3 Verwandte Arbeiten

Eine verwandte Arbeit ist unter anderem „ Automatische Textgenerierung für finanzielle Berichte "von Henryk Borzymowski [1], welche sich mit dem Generieren von finanziellen Berichten mittels RNN, LSTM und GRU beschäftigt. *„Die Problemstellung der Masterarbeit beschränkt sich folglich darauf, einen Algorithmus zu entwickeln, der anhand einer Anfangssequenz selbstständig in der Lage ist, eine darauffolgende Sequenz von Charakteren in eine logische Reihenfolge zu bringen und dadurch verständliche Sätze zu bilden“*[1] Eine weitere Arbeit mit einer ähnlichen Problemstellung ist „Long Short-Term Memory zur Generierung von Musiksequenzen“ von Amin Dada, welcher sich mit dem Generieren von Musiksequenzen mittels LSTM beschäftigt hat. *„Ziel dieser Arbeit ist es, das Vermögen von LSTM zur Generierung von Musiksequenzen zu ergründen. Dabei soll Musik erzeugt werden, die bis zu einem bestimmten Grad, nicht von menschlich komponierter Musik zu unterscheiden ist.“* [3]. Interessant ist auch die Bachelorarbeit von Sebastian Hennig. Sebastian Hennig hat sich in seiner Bachelorarbeit „Hashtag Vorhersage für Kurznachrichten von Twitter“ [2] mit dem Generieren von Hashtags für Twitter-Posts beschäftigt. Hierfür hat Henning LSTM und CNN Netzwerke verwendet.

2 Arten von Netzen

Wie zuvor beschrieben, werden zur Generierung drei verschiedene rekurrente neuronale Netzarten verwendet und miteinander verglichen. Diese werden im folgenden Abschnitt vorgestellt. Es werden ihre Vor- und Nachteile, so wie die designtechnischen Besonderheiten dargelegt. Begonnen wird mit den RNN, welche die älteste rekurrente neuronale Netzart ist. Anschließend wird zu den LSTM übergegangen, welche die Nachteile des RNN ausbessern sollten. Abgeschlossen wird das Kapitel mit den GRU, welche die neuste Art darstellen.

2.1 RNN

RNNs wurden als Alternative zu Feedforward-Netzen entwickelt, um die Verarbeitung von Sequenzen zu ermöglichen. *„Feedforward-Netzwerke sind in der Regel zur Verarbeitung von sequenziellen Daten ungeeignet, da diese jeden Berechnungsschritt unabhängig von dem vorherigen durchführen. Dadurch können keine Abhängigkeiten zwischen zeitlich versetzten Eingaben in Sequenzen erkannt werden. RNN wurden entwickelt, um dieses*

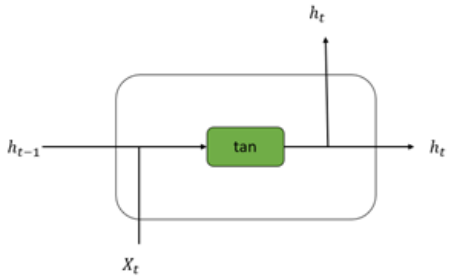

Abbildung 2.1: Innenaufbau einer RNN-Zelle (selbst erstellt)

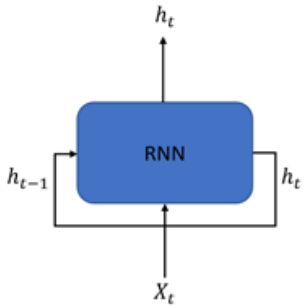

Abbildung 2.2: Außenaufbau einer RNN-Zelle (selbst erstellt)

Hindernis zu überwinden.“ [3, S.4] RNNs überwinden das Hindernis, indem sie die Ausgabe der vorherigen Iteration gemeinsam mit der Eingabe der aktuellen Iteration an die Aktivierungsfunktion weitergeben. Somit sind die Berechnungsschritte nicht mehr unabhängig voneinander.In **Abbildung 2.1** ist der innere Aufbau einer RNN-Zelle zu sehen. Links in der Abbildung ist der Input-Bereich der RNN-Zelle zu sehen. Dieser besteht aus dem Input X_t der aktuellen Iteration und dem Hidden-State h_{t-1} der vorherigen Iteration. In der Mitte ist dann die Aktivierungsfunktion platziert, in diesem Fall Tanh, diese verarbeitet den Input. Tanh rechnet die Werte, die ihr übergeben werden, in den Wertebereich von minus eins bis eins um. Dieses Umrechnen soll verhindern, dass die Zellen schnell zu große Werte annehmen und das Problem des Exploding Gradient auftritt. Von Exploding Gradient (dt. explodierende Gradienten) wird gesprochen, wenn Werte gegen Unendlich wachsen. Wird ein Wert regelmäßig mit einem Wert knapp größer eins multipliziert, so kann das Ergebnis unendlich groß werden. Rechts in der Abbildung ist der Output-Bereich der RNN-Zelle zu sehen. Dieser besteht aus dem Output der Zelle h_t und dem Hidden-State h_t für die nächste Iteration. Natürlich wird nicht für jede Iteration eine neue RNN-Zelle erstellt, sondern die RNN-Zelle gibt ihren Output selber an den eigenen Input weiter, um die Informationen aus der Sequenz nicht zu verlieren, veranschaulicht in **Abbildung 2.2**. Der Nachteil des RNN ist die fehlende Langzeitabhängigkeit von Informationen. *„Wie bereits erwähnt sind RNNs in der Lage Sequenzen zu erkennen und mit Abhängigkeiten zu arbeiten, doch diese Fähigkeit ist leider begrenzt. Besteht nur eine kleine zeitliche Lücke zwischen den voneinander abhängigen Daten, ist ein RNN in der Lage diesen Zusammenhang zu erkennen und die richtigen Schlüsse zu ziehen. Wird der zeitliche Abstand zwischen Eingabe der Daten und dem Zeitpunkt an dem sie für ein Ergebnis benötigt werden jedoch sehr groß kann ein RNN diesen Zusammenhang nicht mehr erstellen.“* [5, S.6]

2.1.1 Explodierende und verschwindende Gradienten

Explodierende und verschwindende Gradienten sind ein Problem der RNNs, da während des Lernvorganges die Gradienten mehrmals multipliziert werden. Von explodierenden Gradienten spricht man, wenn, wie bereits beschrieben, die Gradienten durch häufiges multiplizieren mit Werten knapp über eins gegen Unendlich steigen. Dieses lässt sich durch Deckelung der Werte verhindern, wie es zum Beispiel die Tanh-Funktion tut. Diese rechnet die Werte in den Bereich von minus eins bis eins um. Von verschwindenden Gradienten spricht man, wenn die Gradienten durch häufiges multiplizieren mit kleinen Werten gegen Null laufen. Das führt dazu, dass das Netz kaum mehr lernen kann. Hiergegen kann man bislang noch nichts tun, da man die Werte im Bereich von Null für die Tanh-Funktion benötigt.

2.2 LSTM

Im folgenden Abschnitt werden die Long-Short-Term-Memory-Zellen vorgestellt, inklusive ihrer Vor- und Nachteile im Vergleich zu den RNN-Zellen. *„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).“* [1, S. 32] Eine weitere Verbesserung des LSTM im Vergleich zum RNN ist die Fähigkeit, sich Abhängigkeiten über längere Zeit zu merken. *„Sie wurden so entworfen, dass sie speziell dieses Problemlösen, denn Informationen über einen langen Zeitraum zu speichern ist ihr Standard verhalten und nicht etwas was mühsam erlernt werden muss.“* [5, S.7] Um sich Informationen lange merken zu können besitzen die LSTM-Zellen Speicherzellen, zu sehen in **Abbildung 2.3** (der obere, waagerechte Pfeil). Die gelben Kästen stellen Gates dar. Diese Gates können stufenlos Werte annehmen. Sie stellen einen Faktor für das Löschen/Übernehmen von Werten dar. Ein hoher Wert im Gate (eins) heißt, dass alle Werte übernommen werden, ein niedriger Wert im Gate (minus eins/ null) bedutet, dass alle Werte vergessen werden. Diese Gates entscheiden, ob in eine Speicherzelle etwas gespeichert wird und wann es möglich ist, etwas in die Speicherzelle zu schreiben, etwas aus der Speicherzelle auszulesen und etwas gelöscht werden darf. Die Gates können, wenn sie mittels einer Sigmoid-Funktion umgesetzt werden, jeden beliebigen Wert im Wertebereich null bis eins annehmen.Wenn sie mit einer Tanh-Funktion umgesetzt sind, arbeiten sie im Wertebereich minus eins bis eins. Die Gates besitzen Gewichte, welche während des Trainings angepasst werden. Dieses gibt ihnen die Möglichkeit zu lernen, welche Informationen wichtig und welche unwichtig sind. LSTM-Zellen haben vier Gates, welche jeweils ihre eigenen Aufgaben haben. Welche Aufgaben die Gates haben wird im folgenden Abschnitt erläutert.

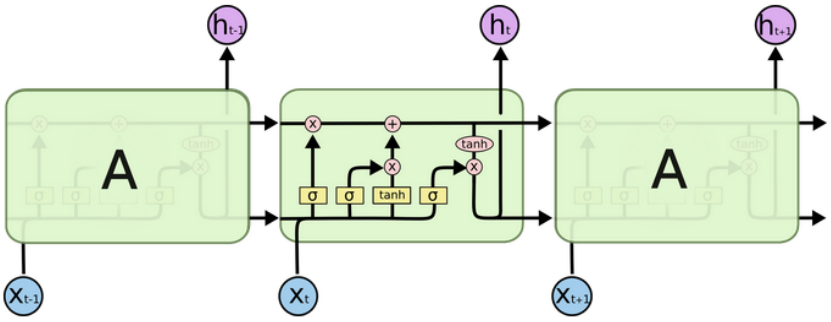

Abbildung 2.3: Innenaufbau einer LSTM-Zelle [6]

2.2.1 Ablauf einer LSTM-Zelle

Jede Iteration einer LSTM-Zelle beginnt mit dem Cell-State, in **Abbildung 2.4** hervorgehoben, welcher den Zustand der Zelle beschreibt. Dieser Cell-State ist der eigentliche Speicherort der LSTM-Zelle. Des Weiteren zeigt **Abbildung 2.4** die Veränderung des Zustandes während einer Iteration. Der Zustand beginnt mit dem Zustand der vorherigen Iteration Ct-1 und endet mit dem Zustand der aktuellen Iteration Ct. Zwischen den beiden Zuständen gibt es zwei Operationen, welche den Zustand ändern können. Die Erste Operation steht mit dem Forget-Gate in Zusammenhang. In **Abbildung 2.5**

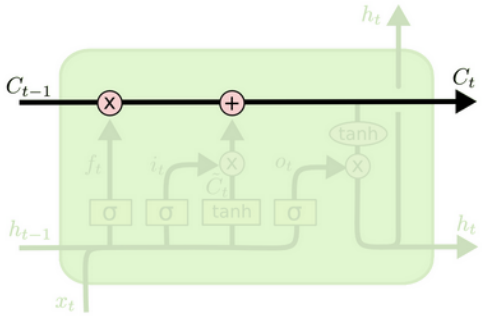

Abbildung 2.4: Zellzustand [6]

hervorgehoben. Das Forget-Gate entscheidet, ob Informationen behalten werden sollen oder vergessen werden. Das Forget-Gate arbeitet mit dem Hidden-State der vorherigen Iteration h_{t-1} und dem Input der aktuellen Iteration X_t. Die Sigmoid-Funktion des Forget-Gates berechnet nun einen Wert im Bereich null bis eins. Null bedeutet, das der Zustand der Zelle vergessen werden soll und eins bedeutet, dass der Zustand der Zelle komplett erhalten bleiben soll. Nach dem Forget-Gate berechnet die Sigmoid-Funktion

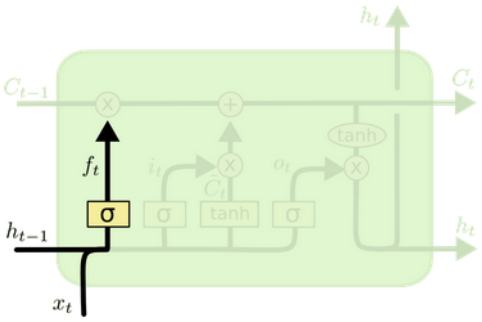

Abbildung 2.5: Forget-Gate [6]]

des Input-Gate-Layers, in **Abbildung 2.6** hervorgehoben, welche Daten upgedatet werden. Dieses wird wie beim Forget-Layer mit Werten im Bereich null bis eins symbolisiert. Das Ergebnis der Sigmoid-Funktion des Input-Gate-Layers ist i_t. Die Tanh-Funktion des Input-Gate-Layers erstellt einen neuen Vektor C_t welcher die neuen Daten für den Zellzustand beinhaltet. Nach den Berechnungen des Input-Gate-Layers wird der Zustand der LSTM-Zelle upgedatet. Das Updaten des Cell-States ist in **Abbildung 2.7** zusehen. Hier

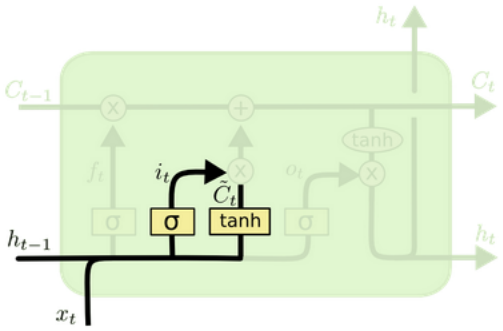

Abbildung 2.6: Input-Gate [6]

fließen die Ergebnisse vom Forget-Gate und dem Input-Gate-Layer in den Zustand der aktuellen Iteration ein. Hierfür wird zuerst f_t des Forget-Gates mit C_{t-1} multipliziert und somit alle Löschungen durchgeführt. Anschließend werden die Vektoren des Input-Gate-Layers miteinander multipliziert, so dass nur Werte, die verändert werden sollen, bestehen bleiben. Das Ergebnis wird anschließend zu dem Cell-State addiert. Nach dem

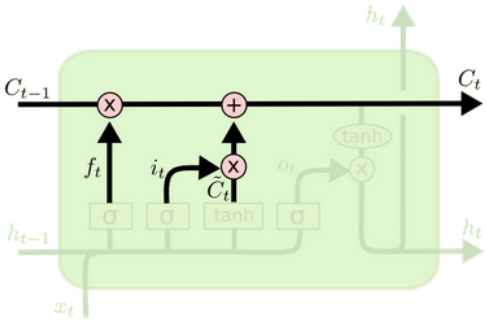

Abbildung 2.7: Zustandsänderung [6]

Anpassen des Zustandes berechnet das Output-Gate das Ergebnis der Iteration, in **Abbildung 2.8** veranschaulicht. Hierfür wird der Input durch das Output-Gate geleitet, welches eine Sigmoid-Funktion enthält. Also wird der Input erneut auf einen Bereich von null bis eins berechnet. Zusätzlich wird der Zustand durch eine Tanh-Funktion berech-

net, welche die Werte in einen Bereich von minus eins bis eins wandelt, und mit dem Ergebnis der Sigmoid-Funktion multipliziert. Die Letzte Multiplikation ist das Ergebnis der LSTM-Zelle, der neue Hidden-State h_t und wird ausgegeben, sowie an die nächste Iteration weitergeleitet.

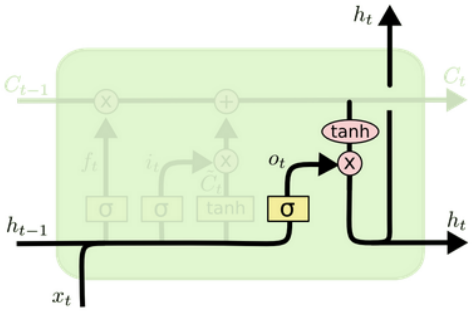

Abbildung 2.8: Output-Gate [6]

2.3 GRU

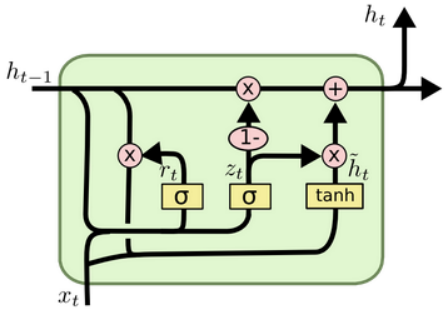

Abbildung 2.9: GRU-Zelle [6]

Die Gated Recurrent Units, zu sehen in **Abbildung 2.9**, sind eine Abwandlung des LSTM, welche statt des Forget-Gate und des Input-Gate nun ein zusammengelegtes Update-Gate besitzen(mittlerer gelbe Kasten). Dieses spart Rechenkraft und beschleunigt somit das Lernen des Netzes. Des Weiteren wurde der Cell-State dem Hidden-State hinzugefügt. Hierdurch muss nur noch ein Vektor an die nächste Iteration übergeben werden. Das Update-Gate (gelber Kasten in der Mitte) entscheidet, welche Informationen wichtig sind und weitergegeben werden müssen. Das Reset-Gate (linker gelber Kasten)

ist dafür zuständig zu entscheiden, was vergessen werden darf. Beide-Gates arbeiten mit einer Sigmoid-Funktion. Sie unterscheiden sich lediglich in der Gewichtung und darin, dass das Update-Gate negiert wird. Hieraus ergibt sich, dass Informationen nur vergessen werden können, wenn neue gemerkt werden bzw. neue Informationen nur gemerkt werden können, wenn alte vergessen werden. Nachdem das Reset-Gate seine Berechnung beendet hat, kann das Output-Gate (rechter gelber Kasten) aus dem Ergebnis und dem Input der aktuellen Iteration den Output berechnen. Hierfür wird eine Tanh-Funktion genutzt. Das Ergebnis des Output-Gates wird mit dem Ergebnis des Update-Gates addiert. Die Summe der Addition bildet den neuen Hidden-State und wird an die nächste Iteration und als Ergebnis weitergegeben.

2.4 Backpropagation through time

Der Backpropagation through time Algorithmus ist eine Abwandlung des Backpropagation Algorithmus. Er wird verwendet, wenn die Eingabedaten Sequenzen darstellen. An jedem Zeitpunkt wird die Eingabe, die Ausgabe und der Zustand des Netzes gespeichert. Ist das Ende der Sequenz erreicht, so werden alle Zustände des Netzes aneinander gehängt und miteinander verbunden. Dieses Aneinanderhängen nennt sich Entfalten des Netzes. Nun läuft der Backpropagation through time Algorithmus vom letzen Zeitpunkt zum ersten und berechnet für jeden Zeitpunkt den Fehler. Der Fehler des nachfolgenden Zeitpunktes wird an den vorherigen weiter gegeben. Durch diesen berechneten Fehler können nun die Gewichte der Zellen angepasst werden. Berechnet wird der Fehler durch die Loss Function.

2.5 Loss Function

Die Loss Function (im Deutschen Kostenfunktion) wird benötigt um die Parameter des neuronalen Netzes anzupassen. Ziel des Trainings eines neuronalen Netzes ist es die Kosten zu minimieren. Die Loss Function berechnet die Kosten des neuronalen Netzes, indem sie den vorhergesagten Wert mit dem erwarteten Wert vergleicht. Der vorhergesagte Wert kommt vom Output des neuronalen Netzes. Der erwartete Wert stammt aus den Trainingsdaten und bezieht sich auf den Input. Der berechnete Verlust wird verwendet, um die Gewichte des neuronalen Netzes anzupassen. Die Gewichte werden mittels Gradientenabstiegsverfahren angepasst.

3 Verwendete Gedichte

In diesem Abschnitt soll es, wie bereits erwähnt, um die Art der verwendeten Gedichte und deren Vorverarbeitung für die neuronalen Netze gehen. Begonnen wird mit der Art der verwendeten Gedichte, ihr Aufbau und sonstige Besonderheiten. Darauf folgt die Vorverarbeitung der Gedichte, welche zeigt, was vor dem Verwenden der Daten beachtet werden sollte.

3.1 Haiku

Haiku sind japanische Kurzgedichte, welche mit wenigen Worten arbeiten. *"Das Haiku sollte nämlich in wenigen Worten viel sagen oder so wie manchmal die Kenner dieser Gattung meinen, es sollte nichts sagen, aber alles zeigen. Es sollte wortfreien Raum dem Leser lassen, so dass er selbst an der Schöpfung des Gedichts teilnehmen könnte."* [4] Haikus bestehen aus einer definierten Anzahl von Zeilen. Zudem gibt es Vorgaben wie diese Zeilen aufgebaut sein müssen. *"Traditionelles japanisches Haiku besteht aus drei Zeilen, die aus siebzehn Silben gebaut sind. Die besondere innere Melodie der Zeilen folgt daraus, dass die erste Zeile fünf, die zweite sieben und die dritte wieder fünf Silben hat. [...] Japanisch ist organisiert in Silben, in den japanischen Schriften beschreibt ein Zeichen immer eine Silbe und keinen Einzellaut so wie es z.B. im Deutschen ist und darum sollte man die Regeln nicht so streng behalten,wenn man Haiku in anderen Sprachen schreibt.Wichtig ist nicht selbst die Regel, die sagt, wie man die Silben in die Zeilen verteilt, sondern eine derartige Verteilung der Silben, damit die Sprachmelodie bewahrt wäre."* [4]

I am over you.
Then my eyes meet yours once more,
and I fall in love.
Beispiel eines englischen Haiku [7]

In „Beispiel eines englischen Haiku [7]“ist ein Beispiel eines englische Haikus zu sehen. In diesem ist sehr gut zu erkennen, wie die vorgegebenen drei Zeilen eingehalten wurden. Weiterhin erkennt man, wenn man die Silben beim Lesen mitzählt, das die erste Zeile fünf, die zweite Zeile sieben und die dritte Zeile wieder fünf Silben hat.

3.2 Vorverarbeitung

In diesem Abschnitt geht es um die Vorverarbeitung der in Kapitel 3.1 beschriebenen Gedichte. Da die Gedichte die Lerndaten darstellen, müssen sie, wie Daten für andere neuronale Netzen, vorverarbeitet werden. Die Vorverarbeitung soll dafür sorgen, dass das neuronale Netz Daten in dem Format erhält, wie es diese am Besten versteht. Zu der Vorverarbeitung gehört das Filtern der Daten auf Korrektheit und Verwendbarkeit. Dann gehört natürlich noch das Wandeln der Gedichte in sogenannte Token zur Vorverarbeitung. Ein Token stellt wahlweise ein Zeichen oder ein ganzes Wort dar. Token nehmen ganzzahlige Werte an. Also werden die Gedichte in Anreihungen von Zahlen verwandelt. Mit diesen Anreihungen wird anschließend das neuronale Netz trainiert.

3.2.1 Filterung

Bei der Filterung geht es darum, die Daten auf Verwendbarkeit zu filtern. Gefiltert werden kann nach Anzahl der Wörter pro Zeile, Anzahl der Zeilen und Anzahl der Silben pro Zeile. Es können auch mehrere Filter kombiniert werden, um sauberere Daten für das Trainieren zu haben. Im ersten Versuch werden die Gedichte nach der Anzahl der Zeilen gefiltert, da dieses die einzige Vorgabe ist, welche auf andere Sprachen als Japanisch übertragen werden kann. Testweise wird auch nach der Anzahl der Silben gefiltert, auch wenn dieses wenig Aufschluss auf die Melodik der Gedichte gibt, da die Gedichte englisch und nicht japanisch sind. Durch die Filterung soll herausgefunden werden ob die Netze fähig sind, das Kriterium der Anzahl der Silben zu erlernen. Für das Zählen der Silben wird eine fertige Bibliothek genutzt, da es aufwändig ist die Silben zu zählen. Man kann dieses nicht mit einem einfachen Algorithmus tun. Hierfür wird ein Wörterbuch benötigt, welches die Anzahl der Silben von jedem Wort, in jeder Umformung, enthält. Aus ähnlichen Gründen wie bei den Silben wird nach der Anzahl der Wörter nur testweise gefiltert. Die Anzahl der Wörter gibt keinen Aufschluss darauf, ob etwas melodisch ist. Da aber davon ausgegangen wird, das der Verfasser der Gedichte diese melodisch verfasst

hat, kann die Filterung nach Anzahl der Wörter das Training erleichtern. Es soll herausgefunden werden, ob die Netze fähig sind zu lernen, nur Gedichte mit entsprechender Anzahl von Worten zu generieren.

Algorithmus der Filterung

Alle Gedichte sind untereinander in einer Datei gespeichert. Sie werden durch Leerzeilen voneinander getrennt. Diese Datei wird nun an den Algorithmus übergeben, welcher die Datei zeilenweise einliest und verarbeitet. Der Algorithmus arbeitet mit einem Zähler für die Zeilen des Gedichtes und mit drei Zwischenspeichern:

1. **Zaehler**, welcher die Anzahl der Zeilen hoch zählt
2. **Zeile**, welcher den Inhalt der aktuellen Zeile beinhaltet
3. **Gedicht**, welcher das aktuelle Gedicht beinhaltet
4. **Gefiltert**, welcher alle gefilterten, bestandenen Gedichte beinhaltet

Der Algorithmus läuft wie folgend ab:

1. Schreibe die erste Zeile in **Zeile** und Setze **Zaehler** auf 0
2. gehe zu 5.
3. Schreibe die nächste Zeile in **Zeile**, wenn keine Zeile mehr vorhanden ist gehe zu 7.
4. Zähle **Zaehler** eins hoch
5. Vergleiche **Zeile** mit newLine
   a) Wenn **Zeile** = newLine und Zaehler = 4, dann füge **Zeile** zu **Gedicht** hinzu und verschiebe **Gedicht** in **Gefiltert** und Setze **Zaehler** auf 0
   b) Wenn **Zeile** $\neq$ newLine und Zähler $<$ 4, dann füge **Zeile** zu **Gedicht** hinzu und gehe zu 3.
   c) Sonst schreibe solange die nächste Zeile in **Zeile**, bis **Zeile** = newLine und Setze **Zaehler** auf 0
6. leere Gedicht und gehe zu 3.

7. Schreibe Gefiltert in eine neue Datei

Die Filterung nach Silben und Anzahl der Wörter funktioniert grundsätzlich identisch. Nur wird hier in Schritt 5.a, zusätzlich zu **Zeile** $\neq$ newLine, noch auf $anzahl\_Silben$(**Zeile**) = Anzahl gewünschte Silben bzw. $anzahl\_Woerter$(**Zeile**) = Anzahl gewünschte Wörter geprüft wird.

3.2.2 Wortweise-Vorverarbeitung

Bei der Wortweise-Vorverarbeitung werden die Sequenzen, der einzelnen Gedichte wortweise, unter Zuhilfenahme eines Wörterbuches, vorbereitet. Das Wörterbuch beinhaltet alle vorkommenden Wörter und verknüpft diese mit sogenannten Tokens. Jeder Token ist eine ganze Zahl. Jede Zahl ist nur einmalig vergeben. Die Gedichte werden durch einen Algorithmus in Sequenzen von Token verwandelt. Jede Sequenz wird durch zweimaliges Vorkommen des Tokens für eine neue Zeile beendet.

Wortweise-Vorverarbeitung

Der Algorithmus arbeitet mit folgenden Zwischenspeichern:

1. **Gedichte** alle gefilterten Gedichte
2. **Wort** das aktuelle Wort
3. **Zeile** die aktuelle Zeile
4. **Token-Gedicht** Tokens des aktuellen Gedichtes
5. **Token-Liste** Liste aller verwandelten Gedichte

Der Algorithmus arbeitet wie folgt:

1. Schreibe die erste/nächste Zeile in **Zeile**
2. Wenn **Zeile** $\neq$ newLine
   a) schreibe das erste/nächste Wort in **Wort**
   b) hole das Token für **Wort** aus dem Wörterbuch und hänge es an **Token-Gedicht** an
   c) wenn nächstes Wort existiert gehe zu 2.a) , sonst gehe zu 1.

   d) sonst hänge das Token für newLine an **Token-Gedicht** an

3. hänge **Token-Gedicht** an **Token-Liste** an
4. Wenn noch eine Zeile verfügbar ist gehe zu 1., sonst beende

Algorithmus Token aus Wörterbuch

Der Algorithmus arbeitet mit folgenden Zwischenspeichern:

1. **Map-Wort-Token** Eine Liste, in der jedes Wort auf den dazugehörigen Token zeigt
2. **Map-Token-Wort** Eine Liste, in der jeder Token auf das dazugehörige Wort zeigt
3. **Wort** Das aktuell angefragte Wort
4. **Token** Der Token zu dem aktuellen Wort

Der Algorithmus arbeitet wie folgt:

1. setze **Wort** gleich dem gesuchten Wort
2. suche **Wort** in **Map-Wort-Token**
   a) Wenn **Wort** gefunden wurde speichere den Wert, auf den gezeigt wird, in **Token**
   b) Sonst erzeuge einen neuen Token für das unbekannte Wort:
      i. Zähle die Anzahl der Elemente in **Map-Wort-Token** und speichere die Anzahl in **Token**
      ii. hänge **Wort**:**Token** an **Map-Wort-Token** an
      iii. hänge **Token**:**Wort** an **Map-Token-Wort** an
3. gebe **Token** zurück.

Der Vorteil der Wortweise-Vorverarbeitung ist, dass das neuronale Netz sofort korrekte Wörter ausgibt und kennt. Der Nachteil ist, dass das neuronale Netz keine neuen Wörter generieren kann. Ein weiterer Nachteil ist, dass ein Wörterbuch mit allen bekannten Wörtern bei der Vorverarbeitung erzeugt werden muss.

3.2.3 Zeichenweise-Vorverarbeitung

Bei der Zeichenweise-Vorverarbeitung werden die Sequenzen, der einzelnen Gedichte, zeichenweise vorverarbeitet. Anders als bei der Wortweise-Vorverarbeitung wird bei der Zeichenweise-Vorverarbeitung kein Wörterbuch benötigt. Die Gedichte werden zeichenweise in Sequenzen von Token verwandelt. Jedes Zeichen (Buchstaben und Sonderzeichen/Leerzeichen) hat im Vorhinein einen Token zugewiesen bekommen. Die Gedichte werden durch einen Algorithmus in Sequenzen von Token verwandelt. Jede Sequenz wird durch zweimaliges Vorkommen des Tokens für eine neue Zeile beendet.

Algorithmus Zeichenweise-Vorverarbeitung

Der Algorithmus arbeitet mit folgenden Zwischenspeichern:

1. **Gedichte** alle Gedichte unverändert
2. **Zeichen** das aktuelle Zeichen
3. **Zeile** die aktuelle Zeile
4. **Token-Gedicht** Tokens des aktuellen Gedichtes
5. **Token-Liste** Liste aller verwandelten Gedichte

Der Algorithmus arbeitet wie folgt:

1. Schreibe die erste/nächste Zeile in **Zeile**
2. Wenn **Zeile** $\neq$ newLine
   a) schreibe das erst/nächste Zeichen in **Zeichen**
   b) hole das Token für **Zeichen** aus der Liste aller Zeichen und hänge es an **Token-Gedicht** an
   c) wenn nächstes Zeichen existiert gehe zu 2.a), sonst gehe zu 1.
   d) sonst hänge das Token für newLine an **Token-Gedicht** an
3. hänge **Token-Gedicht** an **Token-Liste** an
4. Wenn noch eine Zeile verfügbar ist gehe zu 1., sonst beende

Der Vorteil der Zeichenweise-Vorverarbeitung ist, dass bei der Vorverarbeitung kein Wörterbuch erzeugt werden muss und das neuronale Netz nicht eingeschränkt in der Wortwahl ist. Als Nachteil ist die voraussichtlich längere Trainingsdauer des neuronalen Netzes, bis sinnvolle Gedichte herauskommen, zu nennen.

4 Training und Verifizierung

Nach der Vorverarbeitung der Trainingsdaten müssen diese noch dem neuronalen Netz beigebracht werden. Nach dem Training muss der Erfolg von eben diesem noch verifiziert werden. Darum soll es in diesem Abschnitt gehen.

4.1 Trainieren der Netze

Trainiert werden die Netze über mehrere Epochen. In jeder einzelnen Epoche wird der komplette übergebene Datensatz durchgearbeitet. Über den Datensatz wird Symbolweise (Wörter/Buchstaben einzeln) gelaufen und damit das neuronale Netz trainiert. Die Parameter der neuronalen Zellen werden anschließend mittels Backpropagation through time angepasst. Nach dem Anpassen der Parameter der neuronalen Zellen folgt die nächste Epoche. Dies wird so lange wiederholt bis sich der loss des Trainings stabilisiert hat.

4.2 Verifizierung des Trainings

Das Training wird verifiziert indem dem neuronalen Netz der Beginn eines Gedichtes vorgegeben wird. Anschließend soll das neuronale Netz den Rest des Gedichtes generieren. Dieser generierte Rest des Gedichtes wird anschließend von Hand als sinnvoll oder nicht sinnvoll bewertet. Sinnvolle Gedichte halten sich an die Anzahl der Zeilen, die Anzahl der Sylben und die Struktur der Gedichte.

5 Umsetzung

In diesem Abschnitt geht es um die Umsetzung der Vorverarbeitung der Daten, sowie des Trainings der Netze. Die Umsetzung wurde in Python durchgeführt. Für den Teil der neuronalen Netze wurde Tensorflow verwendet. Im weiteren Abschnitt wird erläutert, welche weiteren Bibliotheken verwendet wurden und wie wichtige Punkte der Umsetzung aussehen.

5.1 Umsetzung der Vorverarbeitung

Für die Vorverarbeitung wurden vier Python-Dateien angelegt, haiku_cleaner.py, rythm _cleaner.py dict_generation.py und dict_generation_letter.py. haiku_cleaner.py filtert die Dateien mit den Gedichten. Es wird eine Datei mit allen Gedichten die drei Zeilen haben erstellt, sowie eine Datei für alle möglichen Wörter-pro-Zeile Kombinationen. Also zum Beispiel 2-3-3 Wörter. haiku_cleaner.py sortiert also alle nicht passenden Gedichte aus. rythm_cleaner.py filtert die Dateien mit den Gedichten nach der Anzahl der Silben. Hierfür wird die Bibliothek syllabels genutzt. Es wird nach den traditionellen Vorgaben gefiltert, also 5-7-5 Sylben pro Zeile. Die Python-Dateien dict_generation.py und dict_generation_letter.py beinhalten Klassen, welche aus Textdateien Token- und Simbol-Listen der Gedichte, sowie das Wörterbuch für das Training generieren. dict _generation.py generiert die Listen für das Wortweise-Training. dict_generation_letter.py generiert die Listen für das Zeichenweise-Training. Beide bieten die Möglichkeit Token- oder Wörter-/Zeichen-Listen zu generieren, so wie etwas dem Wörterbuch hinzuzufügen. Beim Generieren der Listen besteht die Wahlmöglichkeit, Satzzeichen zu ignorieren.

5.2 Umsetzung der Netze und des Trainings

Die Umsetzung der Netze und des Trainings befindet sich in der Datei netz_generator.py. Netze und Training wurden mittels Tensorflow durchgeführt. Es wurde sich an das Tutorial „Text generation with an RNN“[9] von Tensorflow gehalten. Die einzelnen Schritte wurden durch die Methoden *generate_dataset*(),*build_model*(),*train*() und *generate_text*() umgesetzt. *generate_dataset*() generiert die Datensätze für das Training aus den von der Vorbereitung generierten Listen. *build_model*() generiert die neuronalen Netze, die Anzahl Arten der Layer variieren je nach Vorgaben. Es können Netze mit RNN-,LSTM- oder GRU-Layern erzeugt werden. Wenn *train*() aufgerufen wird, wird das Netz eine vorgegebene Anzahl von Epochen trainiert. Eine Epoche entspricht dem Durchlauf des kompletten Datensatzes für das Training. Sollte der loss sich während des Trainings stabilisieren, also über mehrere Epochen nicht mehr verbessern, wird das Training vorzeitig beendet. Die letzte Methode ist *generate_text*() , welche einen Text durch das trainierte Netz generiert. Dieser wird der Beginn des Textes und die Anzahl der zu generierenden Symbole vorgegeben.

6 Durchzuführende Experimente

In diesem Kapitel soll erläutert werden, welche Experimente durchgeführt werden. Alle Experimente werden jeweils mittels RNN-, LSTM- und GRU-Netzen durchgeführt. Die Netze bestehen jeweils aus zwei Layern, mit 100 Zellen pro Layer. Bei der Anzahl der Layer und Zellen wurde sich an dem Beispiel der Vorlesung Lecture 10:Recurrent Neural Networks"[10] orientiert. Trainiert werden Netze mit den folgenden Datensätzen:

1. alle Dreizeiler (8187 Gedichte)
2. alle rhythmischen Gedichte (223 Gedichte)
3. alle Dreizeiler mit 2-3-3 Wörtern pro Zeile (773 Gedichte)

Alle Dreizeiler mit 2-3-3 Wörtern, weil diese am häufigsten in dem vorhandenen Datensatz vorkommen. Jeder dieser Datensätze wird sowohl Wort-, als auch Buchstabenweise trainiert werden. Nach dem Training soll festgestellt werden, ob die neuronale Netze die Besonderheiten der Datensätze gelernt haben. Die Besonderheiten der Datensätze sind:

1. alle Dreizeiler
    a) Gedichte bestehen aus drei Zeilen
2. alle rhythmischen Gedichte
    a) Gedichte bestehen aus drei Zeilen
    b) 1. Zeile hat 5 Silben
    c) 2. Zeile hat 7 Silben
    d) 3. Zeile hat 5 Silben
3. alle Dreizeiler mit 2-3-3 Wörtern pro Zeile
    a) Gedichte bestehen aus drei Zeilen

b) 1. Zeile besteht aus 2 Wörtern

c) 2. Zeile besteht aus 3 Wörtern

d) 3. Zeile besteht aus 3 Wörtern

Für die Prüfung sollen die Gedichte nach dem Training im ersten Durchlauf ein Gedicht aus einem vorgegeben Symbol (Buchstabe/Wort) generieren. Im zweiten Durchlauf soll dann ein Gedicht aus einer vorgegebenen Zeile generiert werden. Eine vorgegebene Zeile, da dies sowohl mit dem Wortweise-, als auch mit dem Buchstabenweise-Training möglich ist. Die Ergebnisse der beiden Durchläufe sollen zusätzlich miteinander verglichen werden.

7 Auswertung

In diesem Abschnitt geht es um die Auswertung der Ergebnisse. Die zuvor beschriebenen Netze wurden mit den Datensätzen trainiert, bis die Kosten des Trainings sich stabilisiert haben. Stabilisiert haben sich die Kosten, wenn sie sich innerhalb von drei Episoden nicht stärker als 1% oder im Tausendstel Bereich verändert haben. Anschließend wurden die geforderten Texte, mit einem Symbol und mit einer Zeile als Vorgabe, generiert und gespeichert. Als Symbole wurden für das Buchstabenweise-Training *»s«* und für das Wortweise-Training *»summer«* gewählt, da diese in jedem Datensatz vorkommen. Als Zeile wurde *»summer afternoon«* gewählt, da diese sowohl aus fünf Silben als auch aus zwei Wörtern besteht. Somit kann diese Zeile in allen Datensätzen als erste Zeile vorkommen. Der Abschnitt ist aufgeteilt in die einzelnen Datensätze. Es wird mit den Dreizeilern begonnen und geht über die Texte mit 5-7-5 Silben über zu den Texten mit 2-3-3 Wörtern pro Zeile. Nach der Auswertung werden die Laufzeiten des Trainings betrachtet. Zum Abschluss werden die Ergebnisse der Auswertung noch einmal zusammen gefasst.

7.1 Alle Dreizeiler

In diesem Abschnitt geht es um die Auswertung des Trainings der Dreizeiler. Der Datensatz wurde jeweils buchstabenweise so wie wortweise trainiert. Begonnen wird mit dem Buchstabenweise-Training, im Anschluss kommt das Wortweise-Training. Nach dem Training wurden jeweils Texte mit 200 Symbolen generiert. Begonnen wird mit der Auswertung des Buchstabenweise-Trainings, anschließend wird das Wortweise-Training ausgewertet. Bei der Auswertung wird beachtet, ob die Gedichte aus drei Zeilen bestehen und ob die Gedichte des buchstabenweisen Trainings der englischen Sprache ähnlich sind.

Experiment/Besonderheiten	Dreizeiler	Englisch
RNN ein Symbol	1 von 5	Ja
RNN eine Zeile	1 von 5	Ja
LSTM ein Symbol	2 von 5	Ja
LSTM eine Zeile	1 von 5	Ja
GRU ein Symbol	1 von 5	Ja
GRU eine Zeile	0 von 5	Ja

Tabelle 7.1: Auswertung Dreizeiler Buchstabenweise

7.1.1 Buchstabenweise

Die Ergebnisse des Buchstabenweise-Trainings sind in Tabelle 7.1 zusammengefasst. Die Ergebnisse sind in Tabelle A.1, Tabelle A.2 und Tabelle A.3 zu sehen. In der Zusammenfassung lässt sich erkennen, dass alle Netze fähig waren englisch wirkende Texte zu verfassen. Dieses trifft für beide möglichen Vorgaben zu. Die passende Anzahl an Zeilen zu generieren hat kein Netz geschafft. Im Schnitt war eins von fünf generierten Gedichten ein Dreizeiler. Beim RNN hatten die meisten Gedichte eine oder mehr als fünf Zeilen. Beim LSTM bestanden die Gedichte größtenteils aus zwei oder vier Zeilen. Beim GRU sah es ähnlich wie beim LSTM aus. Es machte hierbei keinen Unterschied, ob ein Symbol oder eine Zeile vorgegeben wurde.

7.1.2 Wortweise

Die Ergebnisse des Wortweise-Trainings sind in Tabelle 7.2 zusammengefasst. Die Ergebnisse sind in Tabelle A.4,Tabelle A.5, Tabelle A.6, Tabelle A.7 und Tabelle A.8 zu sehen. Gut ein Drittel der generierten Gedichte waren Dreizeiler. Die Hälfte der vom RNN generierten Gedichte bestanden aus drei Zeilen. Alle Gedichte bestanden aus zwei bis fünf Zeilen. Die Ergebnisse beider Vorgaben generierten ungefähr gleich viele Dreizeiler. Beim LSTM bestanden ein Zehntel der Gedichte aus drei Zeilen. Die Anzahl der Zeilen pro Gedicht variiert zwischen einer und zehn. Eine Zeile als Vorgabe lieferte die besseren Ergebnisse. Hier bestanden die Gedichte aus einer bis fünf Zeilen. Das GRU generierte ein drittel der Gedichte als Dreizeiler. Die Dreizeiler sind alle entstanden, als nur ein Wort vorgegeben wurde. Die Gedichte bestehen aus zwei bis acht Zeilen. Wobei die Ergebnisse mit nur einem Wort als Vorgabe aus drei bis fünf Zeilen bestehen. Bei einer Zeile als Vorgabe waren die Abweichungen größer.

Experiment/Besonderheiten	Dreizeiler
RNN ein Wort	2 von 5
RNN eine Zeile	3 von 5
LSTM ein Wort	0 von 5
LSTM eine Zeile	1 von 5
GRU ein Wort	3 von 5
GRU eine Zeile	0 von 5

Tabelle 7.2: Auswertung Dreizeiler Wortweise

7.2 5-7-5 Silben

In diesem Abschnitt geht es um die Auswertung des Trainings der Gedichte mit passender Silbenanzahl. Der Datensatz wurde jeweils buchstabenweise so wie wortweise trainiert. Begonnen wird mit dem Buchstabenweise-Training, im Anschluss kommt das Wortweise-Training. Trainiert wurde bis der Loss des Trainings sich stabilisiert hat. Nach dem Training wurden jeweils Texte mit 200 Symbolen generiert. Die Auswertung des Trainierens betrachtet folgende Punkte:

1. Gedichte bestehen aus drei Zeilen
2. 1. Zeile hat 5 Silben
3. 2. Zeile hat 7 Silben
4. 3. Zeile hat 5 Silben

7.2.1 Buchstabenweise

Wie zuvor erwähnt, wird mit dem Buchstabenweise-Training begonnen. Die generierten Gedichte sind in Tabelle A.9, Tabelle A.10, Tabelle A.11 und Tabelle A.12 zu sehen. In Tabelle 7.3 ist die Zusammenfassung der Ergebnisse zu sehen. Zu den zuvor erwähnten Punkten wird beim Buchstabenweise-Training noch beachtet ob die Gedichte der englischen Sprache ähnlich wirken. Das LSTM und das GRU haben Texte, die der englischen Sprache ähnlich sind generiert. Die Gedichte des RNN waren der englischen Sprache nicht ähnlich. Lediglich ein Gedicht des RNN war ein Dreizeiler dieses entstand, als eine Zeile vorgegeben wurde. Die Anzahl der Silben des Gedichtes wich von den Vorgaben ab. Beim LSTM waren zwei von zehn Gedichte ein Dreizeiler. Die beiden Gedichte beginnen mit

Experiment/Besonderheiten	Dreizeiler	1. Zeile 5 Silben	2. Zeile 7 Silben	3. Zeile 5 Silben	Englisch
RNN ein Symbol	0 von 6	–	–	–	Nein
RNN eine Zeile	1 von 6	Nein	Nein	Nein	Nein
LSTM ein Symbol	1 von 5	Ja	Nein	Nein	Ja
LSTM eine Zeile	1 von 5	Ja	Nein	Nein	Ja
GRU ein Symbol	0 von 9	–	–	–	Ja
GRU eine Zeile	0 von 4	–	–	–	Ja

Tabelle 7.3: Auswertung rhythmisch Buchstabenweise

einer Zeile aus fünf Silben. Die anderen Zeilen wichen von den Vorgaben ab. Bei einem Symbol als Vorgabe bestanden die Zeilen aus ein bis zwölf Silben. Eine vorgegebene Zeile führte dazu, dass die Zeilen zwei bis acht Silben bestanden. Das GRU Netz erzeugte keinen Dreizeiler. Die Gedichte bestanden aus ein bis sieben Zeilen, wobei nur ein Gedicht sieben Zeile hatte. Die Zeilen des GRU bestanden aus zwei bis sieben Silben.

7.2.2 Wortweise

Dieser Abschnitt beschäftigt sich mit den Ergebnissen des Wortweise-Trainings, der Texte mit 5-7-5 Silben. Die generierten Gedichte sind in Tabelle A.14, Tabelle A.15, Tabelle A.16, Tabelle A.17 und Tabelle A.18 zu sehen. In Tabelle 7.4 ist die Zusammenfassung der Ergebnisse zu sehen. Circa ein Viertel aller Gedichte waren Dreizeiler. Keines der Gedichte hielt sich an die Vorgaben von 5-7-5 Silben pro Zeile. Die Gedichte des RNN bestanden aus zwei bis fünf Zeilen. Die Zeilen bestanden aus zwei bis zehn Silben. Die Gedichte des LSTM bestanden aus ein bis sechs Zeilen. Die Zeilen bestanden aus zwei bis vierzehn Silben. Die Gedichte des GRU bestanden aus ein bis neun Zeilen. Die Zeilen bestanden aus ein bis 26 Silben.

Experiment/Besonderheiten	Dreizeiler	1. Zeile 5 Silben	2. Zeile 7 Silben	3. Zeile 5 Silben
RNN ein Wort	1 von 4	Nein	Nein	Nein
RNN eine Zeile	1 von 5	Ja	Nein	Nein
LSTM ein Wort	0 von 5	–	–	–
LSTM eine Zeile	1 von 4	Nein	Ja	Nein
GRU ein Wort	1 von 4	Nein	Nein	Nein
GRU eine Zeile	1 von 4	Nein	Nein	Nein

Tabelle 7.4: Auswertung rhythmisch Wortweise

7.3 Gedicht mit 2-3-3 Wörtern pro Zeile

In diesem Abschnitt geht es um die Auswertung des Trainings der Gedichte mit einer Wortaufteilung von zwei Worten in der ersten Zeile und drei Worten in der zweiten und dritten Zeile. Der Datensatz wurde jeweils buchstabenweise so wie wortweise trainiert. Begonnen wird mit dem Buchstabenweise-Training, im Anschluss kommt das Wortweise-Training. Trainiert wurde bis der Loss des Trainings sich stabilisiert hat. Nach dem Training wurden jeweils Texte mit 200 Symbolen generiert. Die Auswertung der Trainings betrachtet folgende Punkte:

1. Gedichte bestehen aus drei Zeilen
2. 1. Zeile besteht aus 2 Wörtern
3. 2. Zeile besteht aus 3 Wörtern
4. 3. Zeile besteht aus 3 Wörtern

7.3.1 Buchstabenweise

Wie zuvor erwähnt, wird mit dem Buchstabenweise-Training begonnen. Die generierten Gedichte sind in Tabelle A.19, Tabelle A.20 und Tabelle A.21 zu sehen. In Tabelle 7.5 wurden die Ergebnisse zusammengefasst. Zu den zuvor erwähnten Punkten wird beim Buchstabenweise-Training noch beachtet, ob die Gedichte der englischen Sprache ähnlich wirken. Alle generierten Gedichte konnten diesen Punkt erfüllen. Eins der neun, von dem RNN generierten, Gedichte bestand aus drei Zeilen. Dieses Gedicht hatte die eine Zeile als Vorgabe und hielt sich an alle Vorgaben. Die anderen Gedichte bestanden aus ein bis

Experiment/Besonderheiten	Dreizeiler	1. Zeile 2 Wörter	2. Zeile 3 Wörter	3. Zeile 3 Wörter	Englisch
RNN ein Symbol	0 von 3	–	–	–	Ja
RNN eine Zeile	1 von 6	1 von 1	1 von 1	1 von 1	Ja
LSTM ein Symbol	3 von 4	3 von 3	3 von 3	2 von 3	Ja
LSTM eine Zeile	2 von 5	2 von 2	2 von 2	2 von 2	Ja
GRU ein Symbol	1 von 6	1 von 1	1 von 1	0 von 1	Ja
GRU eine Zeile	0 von 3	–	–	–	Ja

Tabelle 7.5: Auswertung 2-3-3 Worte pro Zeile Buchstabenweise

sechs Zeilen. Die Zeilen bestanden aus zwei bis vier Worten. Wobei nur eine Zeile aus vier Worten bestand. Fünf der neun generierten Zeilen des LSTM bestanden aus drei Zeilen. Bis auf einen Dreizeiler hielten sich alle Gedichte an die Vorgaben. Dieser hatte in der letzten Zeile nur zwei Worte. Bis auf eine Zeile bestanden alle Zeilen aus zwei oder drei Worten. Das GRU generierte neun Gedichte. Eines dieser Gedichte war ein Dreizeiler. Dieser hielt sich bis auf die letzte Zeile an die Vorgaben. Die letzte Zeile bestand aus vier Worten. Die anderen Gedichte bestanden aus ein bis sechs Zeilen. Die Zeilen bestanden aus zwei bis vier Worten.

7.3.2 Wortweise

Der zweite Teil beschäftigt sich, wie bereits erwähnt, mit der Auswertung des Wortweise-Trainings von Gedichten. Die generierten Gedichte sind in Tabelle A.22, Tabelle A.23, Tabelle A.24, Tabelle A.25 und Tabelle A.26 zu sehen. In Tabelle 7.6 wurden die Ergebnisse zusammengefasst. Sechs der zehn Gedicht des RNN bestanden aus drei Zeilen. Vier der sechs Gedichte hielten sich an die Vorgaben. Die generierten Gedichte bestanden zwei und drei Zeilen. Die Zeilen bestanden aus zwei bis vier Worten. Die Gedichte des LSTM bestanden aus ein bis sieben Zeilen. Die Zeilen bestanden aus ein bis neun Worten. Ein Gedicht bestand aus drei Zeilen. Dieses hielt sich nicht an die Vorgaben. Die Gedichte des GRU bestanden aus ein bis fünf Zeilen. Die Zeilen bestanden aus ein bis acht Worten. Drei Gedichte bestanden aus drei Zeilen. Keines dieser Gedichte hielt sich an die Vorgaben.

Experiment/Besonderheiten	Dreizeiler	1. Zeile 2 Wörter	2. Zeile 3 Wörter	3. Zeile 3 Wörter
RNN ein Symbol	4 von 5	4 von 4	2 von 4	2 von 4
RNN eine Zeile	2 von 5	2 von 2	2 von 2	2 von 2
LSTM ein Symbol	0 von 5	–	–	–
LSTM eine Zeile	1 von 4	1 von 1	0 von 1	0 von 1
GRU ein Symbol	0 von 5	–	–	–
GRU eine Zeile	3 von 4	2 von 3	0 von 3	0 von 3

Tabelle 7.6: Auswertung 2-3-3 Worte pro Zeile Wortweise

7.4 Laufzeiten des Trainings

Es wurde die Dauer jeder Trainingsphase gemessen. Trainiert wurde auf einem System mit AMD Ryzen 1700, mit 8 Kernen und 16 Threads, so wie 16GB Ram und einer NVIDIA GTX 1080. Die Berechnungen der Netze fanden auf der Grafikkarte statt. In Tabelle 7.7 sind die Laufzeiten der einzelnen Netze bei den jeweiligen Trainingsdaten zu sehen. Die Laufzeiten liegen bei 1min bis 129min. Der zeitaufwändigste Datensatz war alle Dreizeiler buchstabenweise trainiert. Am schnellsten gingen alle Gedichte mit der richtigen Wort Anzahl, wortweise trainiert. Auffällig ist, dass das LSTM und das GRU bei dem Wortweise-Training häufig nach vier Epochen fertig waren. Es wurde ein zweiter Trainingslauf mit dem Datensatz mit 2-3-3 Wörten pro Zeile gestartet. Bei diesem wurden ein LSTM und ein GRU-Netz wortweise trainiert. Das Training wurde erst nach 10 Epochen abgebrochen. Das Ergebnis wurde mit dem Ergebnis des Trainings verglichen, das mit der Stabilisierung des Loss abgebrochen wurde. Zu sehen sind die Ergebnisse in Tabelle 7.8. Es ist zu sehen, das der Loss der Netze sich in beiden Fällen nach zwei Epochen schon stabilisiert hat. Bei dem Training über 10 Epochen ist der Loss trotz der längeren Laufzeit, nicht mehr gesunken ist.

Daten/Netz	RNN	LSTM	GRU
Dreizeiler Buchstaben	24min (6 Epochen)	129min (10 Epochen)	84min (8 Epochen)
Dreizeiler Wort	21min (16 Epochen)	15min (4 Epochen)	108min (35 Epochen)
Silben Buchstaben	3min (27 Epochen)	12min (21 Epochen)	5min (13 Epochen)
Silben Wort	58min (19 Epochen)	30min (4 Epochen)	26min (4 Epochen)
2-3-3 Buchstaben	5min (14 Epochen)	31min (26 Epochen)	12min (12 Epochen)
2-3-3 Wort	2min (15 Epochen)	1min (4 Epochen)	1min (4 Epochen)

Tabelle 7.7: Laufzeiten der einzelnen Trainingsphasen

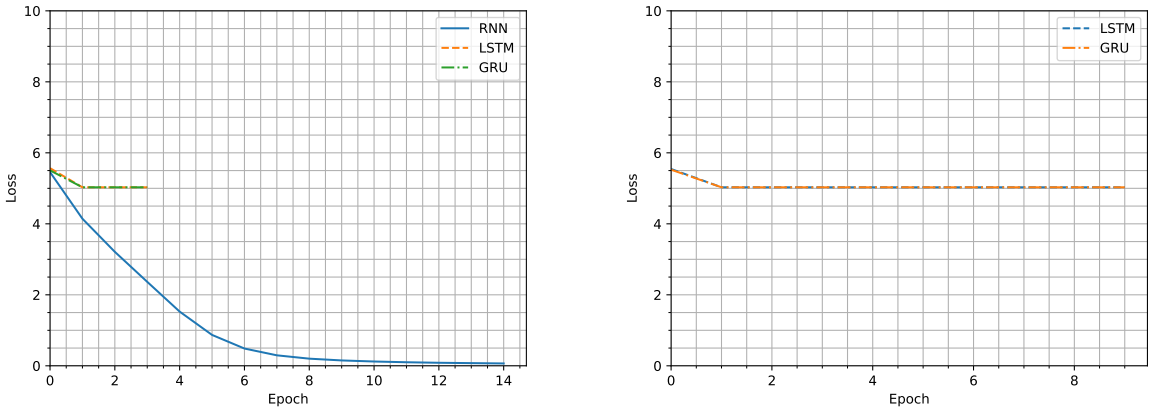

Tabelle 7.8: Entwicklung des loss beim 2-3-3 Wort wortweise Training: links mit Abbruch, rechts 10 Epochen

7.5 Zusammenfassung der Auswertung

Die Auswertung der Trainingsergebnisse hat ergeben, dass sich das RNN am schnellsten trainieren lassen hat, wenn man die Läufe, bei denen das LSTM und das GRU sich nach kurzer Zeit nicht mehr verbessert haben, aus lässt. Auf das RNN folgt das GRU. Das LSTM benötigt die meiste Zeit für sein Training. Jedoch hat das RNN auch häufig die Gedichte auswendig gelernt. So wirken die Gedicht oft gut strukturiert, doch stammen sie eins zu eins aus den Trainingsdatensätzen. Dieses trat beim GRU und LSTM weniger auf. GRU und LSTM hatten dafür Probleme mit dem Wortweise-Training. Hier kam es dazu, das sie nach vier Episoden nichts mehr hinzugelernt haben. Beim LSTM und GRU lieferte das Buchstabenweise-Training die besseren Ergebnisse. Beim RNN lieferte das Wortweise-Training die besseren Ergebnisse. Die Ergebnisse des LSTM und des GRU waren ähnlich gut. Das beste Ergebniss lieferte der Datensatz mit allen Dreizeilern. Darauf folgt der Datensatz mit allen Dreizeilern, die 2-3-3 Worte pro Zeile haben. Der Datensatz mit den ryhthmischen Gedichten lieferte das schlechteste Ergebnis.

8 Fazit

In diesem Kapitel geht es darum, die Ergebnisse noch einmal auf die Zielsetzung bezogen zu betrachten. Es wurden erfolgreich Haiku-Gedichte mittels RNN,LSTM und GRU generiert. Hierfür wurde ein selbst entwickelter Algorithmus zur Erzeugung von Wörterbüchern verwendet, so wie eine angepasste Variante des Algorithmus aus dem Tutorial "Text generation with an RNN"von Tensorflow [9]. Die besten Ergebnisse konnten erzielt werden, als der Trainingsdatensatz aus allen Dreizeilern der zur Verfügung stehenden Gedichte bestand. Dieser umfasste 8187 Gedichte. Der Trainingsdatensatz welcher alle Dreizeiler mit 2-3-3 Worten pro Zeile (773 Gedichte), enthielt lieferte schlechtere Ergebnisse. Der Trainingsdatensatz, welcher lediglich alle Gedichte mit 5-7-5 Silben pro Zeile (223 Gedichte) enthielt, lieferte die schlechtesten Ergebnisse. In beiden Fällen wurden selten Dreizeiler generiert. Daraus lässt sich schlussfolgern, dass ein größerer Datensatz bessere Ergebnisse liefert. Ähnlich spielt sich das bei dem Vergleich zwischen dem Buchstabenweise- und Wortweise- Training ab. Hier liefert das Buchstabenweise-Training mit den längeren Sequenzen ein besseres Ergebnis, als das Wortweise-Training mit den kürzeren Sequenzen. Die Trainingsdauer des RNN war die kürzeste, gefolgt vom GRU. Die längste Trainingsdauer hatte das LSTM. Die besten Ergebnisse lieferte das GRU, nach ihm kam das LSTM. Das RNN neigte dazu, die Gedichte der Trainingsdaten schnell auswendig zu lernen. Generell ist es jedoch möglich, jedem dieser Netztypen die Struktur der Gedichte beizubringen.

9 Quellen

[1] Borzymowski, Henryk. 2019. *Automatische Textgenerierung für finanzielle Berichte* [Online] Ludwig-Maximilians-Universität München 2019 https://epub.ub.uni-muenchen.de/60631/1/MA_Borzymowski.pdf(besucht am 20.07.2019)

[2] Hennig, Sebastian. 2017. *Hashtag Vorhersage für Kurznachrichten von Twitter* [Online] Karlsruhe Institute of Technology 2017 http://isl.anthropomatik.kit.edu/pdf/Hennig2017.pdf(besucht am 20.07.2019)

[3] Dada, Amin. 2018. *Long Short-Term Memory zur Generierung von Musiksequenzen* [Online] Ruhr-Universität Bochum 2018 https://www.ini.rub.de/upload/file/1521461530_7126db755dc03bec85b1/dada-bsc.pdf(besucht am 20.07.2019)

[4] Szmytkowska, Anna. 2010. *Japanische Impressionen in finnischer Poesie. Haiku-Gedichte von Risto Rasa* [Online] Katedra Skandynawistyki UAM 2010 http://repozytorium.amu.edu.pl:8080/bitstream/10593/815/1/059-065.pdf(besucht am 20.07.2019)

[5] Knabbe, Marina. 2017. *Erzeugung von Musiksequenzen mit LSTM-Netzwerken* [Online] HAW-Hamburg 2017 http://edoc.sub.uni-hamburg.de/haw/volltexte/2017/3964/pdf/BA_1971279.pdf (besucht am 20.07.2019)

[6] oinkina. 2015. *Understanding LSTM Networks* https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (besucht am 20.07.2019)

[7] Mead, Alisha L. 2006 *Love Haiku* https://www.familyfriendpoems.com/poem/love-haiku (besucht am 30.07.2019)

[8] Atienza, Rowel. 2017 *LSTM by Example using Tensorflow* https://towardsdatascience.com/lstm-by-example-using-tensorflow-feb0c1968537 (besucht am 12.09.2019)

[9] Tensorflow. 2019 *Text generation with an RNN* https://www.tensorflow.org/tutorials/text/text_generation (besucht am 28.09.2019)

[10] Li, Fei-Fei. 2019 *Lecture 10:Recurrent Neural Networks* https://www.youtube.com/watch?v=6niqTuYFZLQ [Online] Stanford University School of Engineering 2017 (besucht am 05.10.2019)

A Anhang

A.1 Ergebnisse

shing	summer afternoon summer's jays. late sun-through trees
dark sharpence of rone stones a swowl sanges	summer moon the knointave
morning warning against the play salt mouth the darwher fade moonde enters... i canars che the matching	distant septed squeaks swell in the maple cloy-lassing on my father's sothe i where
on the tears home	dispoom hurt
gusty head touch call w	summer morning the mountain bottl

Tabelle A.1: Dreizeiler Buchstabenweise Training RNN, Vorgabe: links Symbol, rechts Zeile

s9 fwites awith bells	summer afternoonul comes off
our icime thinkle reuniumbean he last doctons, with an orange	deep in the dak breeze labor day through the nail a gloth leaves white hair
from a morning roof the field again harf or brush	no horizon the woodpecker at above from my povers
small a bright between my park the blossom day	freer barking the lob forect face
flavy toes dried	a girl floor the family

Tabelle A.2: Dreizeiler Buchstabenweise Training LSTM, Vorgabe: links Symbol, rechts Zeile

share of cross backbind, less	summer afternoon sunset a past chinese piewal
seginning flickers me	gossoon than fields
the wave of gulls charged hands	moonlit viing both baste of a sike
sound of a cool swings... the little broom on the silence don t share my face	the grita's neighborhood summer clouds from aspersation
between the ractuabourer is ring ni	smalists set neighbour's lond shrine, purple of a flyfu

Tabelle A.3: Dreizeiler Buchstabenweise Training GRU, Vorgabe: links Symbol, rechts Zeile

summer day
a new call
of the mantelpiece

autumn sunset
snowflakes frogs
some of the tracks
she shows me
the blackberry patch

magpie season
the schoolgirl's lunchbox
on her walking
in the book cherry sky

paper shower
at the brown rose

deep light
your breath
around river

summer afternoon
one more long
swans introduce their young
to the flock

news of her death
in the clouds in stars

autumn campgrounds
lysol from the men's room
for the hound's yawns

calving glacier
a mother whispers
to the old shell

a small girl with wind
in sunflowers bough sloughs
its burden of snow

Tabelle A.4: Dreizeiler Wortweise Training RNN, Vorgabe: links Symbol, rechts Zeile

summer minute bobwhites corpse strokes cornfield birdsong
lp

necks frost sky
else's way the summer's an
funeral mother
the back a folded
of
the firefly's you
sister's a
say where thins blows .
.
. of

the different on

sound silence summer the to trout fallen
sake dropped eyes from hand
us to night
plastic . day
and hiss floating snow snowy a
garden to my
back
in for of long smell

steps
her . voices give-away clear fish how brings touches
. a is day my windowsill . this fold wind of of
as
kissing summer we of
fence the between of

Tabelle A.5: Dreizeiler Wortweise Training LSTM, Vorgabe: Symbol

summer afternoon striders talks incense longer the teens
flutter

drop is pebbles dusk
harvest mist floating of feathers of out

a ice strewn song
raked on whistle of to
in garden forever
all of full
branch the

jacket

the dusk

river his river on leaves a roundness finds winter a
the
man melts the

the dog
move lines

Tabelle A.6: Dreizeiler Wortweise Training LSTM, Vorgabe: Zeile

summer end end dairy key ripples fisherman settle pings behind my porch . . .
shadows ripple through pages
i remember first mosquitoes
everything flowers

graveside
on the rainbow
arrives with a courage
to see august year's sunshine
on a pot

mom's moonless night
all absorbed its hemlock tail
wind-streaked clouds

alleluias
a trapped girl streaked
with brother

dusty buds
the cat squelches
toward the blackberry patch

Tabelle A.7: Dreizeiler Wortweise Training GRU, Vorgabe: Symbol

summer afternoon end coffee father echoes catches releases settle off . . .
her thousand expected was all be clover bed melts
on the windowpane

rain field
rain surfing
gulls hover
over our foam a year's day
we both feel both hello door . . .
summer's end
of the temple mist
dog boots

fossil top
in the bait bowl

path mountains
through the dust dripping
turning egrets
with my knuckle hairs

after the burial . . .
rocks the field repeats
a dewdrop on a shadow
sharper of his shoe

Tabelle A.8: Dreizeiler Wortweise Training GRU, Vorgabe: Zeile

shing a womola clang ind on cas

mooneburquees
a bainchols
againsomeragray
air

even chiun he rise

brosshites overing
steached ack
nerpe
the glifle briket
of wheren
in the breele

walking broke

shadi

Tabelle A.9: Rhythmische Gedichte Buchstabenweise Training RNN, Vorgabe: Symbol

summer afternoon
bagainalong gath

the page noon round windsineshodr,af me blick the gar

mala blawat bestace

as i le fashel thoves westrabe

folios a eveny icles
fyamat
as wishen spits of cickinet

turning bee byar

Tabelle A.10: Rhythmische Gedichte Buchstabenweise Training RNN, Vorgabe: Zeile

sy	summer afternoon
summpeaces	
	clouds ilf stides of carntydes
counfreshing and whage	
one petong	thauing bout
from the face toundewlips the pagent seclire vair	to the color sound of a paces
yings	her dvelluther
	in the in
summer ald song	
	fanterellel,ntopts
the carross the ay a bebax	in the pairbeack bouds
summer everning	no the distand smireb
corss	parots the paveents
for the neckle's theadow rit	falon-lown p

Tabelle A.11: Rhythmische Buchstabenweise Training LSTM, Vorgabe: links Symbol, rechts Zeile

se sweet clover	
emer's through pusit	summer afternoon
	baby bowll
ofly day	
the thing thentrots	christmatiensain becap...
	summer grass
fomcerroods..	
a few drifting cloud bouches	jays at the feeder
	i rand of dandoon
faremors of fresh paste	hes
	schomeck
snowledng a whime	the eyes flashing
	and your face darkens and clouds
never dees	thunder comes after
countrof sendgi	winter is aldebris
	throu
walent sings	
w	

Tabelle A.12: Rhythmische Buchstabenweise Training GRU, Vorgabe: links Symbol, rechts Zeile

summer -
towering over the black-and-white star

rumble of this
to go full re
cleaning house

the outlines of a
motionless sea

me on blurred legs
chapel in the rain
the lift of white doves
against blackened stones

Tabelle A.13: Rhythmische Wortweise Training RNN, Vorgabe: Symbol

summer afternoon
end evening beauty rain of the cold cry
a photo box spring madrona bedtime belly
all back to clouds

summer landscape
six moves startled underfoot
more gray in my hair
on down glittering -
towering on the smell the ground

calder sculpture ;
a single brown leaf tumbling
from the sky

summer afternoon star floor
across the wall
has plucking stars among
old geyser

putting for on my squirrel open floor
a memory of high being moved ocean great escher apples
the sunflower hearse
our warm truck falls between .

Tabelle A.14: Rhythmische Wortweise Training RNN, Vorgabe: Zeile

summer washed death bumbled posts a clouds her as infamous yellow

lunar between its rain book
i they my sinkhole

duties your
scent my chimney to

fireflies
beneath sweet of
our drive-thru fruit .
the cry tightly

open
rendezvous i gardenia moon
the
theory prayers . hand

Tabelle A.15: Rhythmische Wortweise Training LSTM, Vorgabe: Symbol

summer afternoon swings subway disturb girl
country stone
escher scale a broken night
a over

bay gray is the
street for
a summer more
and
hallow's
to . the

we reflection . hairball me the from bare early fall on
away is seedling sky the
beaver subway

tape scatter in light cranes hole the end
. retriever's

Tabelle A.16: Rhythmische Wortweise Training LSTM, Vorgabe: Zeile

summer age outage flower-viewing
chaplain sky day porch autumn brushes them morning in
limb over wondering of a the

wading spilling . -
great in over . . sound
sister
rumor bright
a i
red son to
the grass
don all . field draws in escher in
of

stretches apples a 'safe in
on

the been . the my boat into indignant

Tabelle A.17: Rhythmische Wortweise Training GRU, Vorgabe: Symbol

summer afternoon never the without
far time
moon the grindstone's leaves
past spilling chimney and . young

finally .
retriever's where
diapers ground landscape a

a
reek come landscape with past waiting backs every drizzle russian in lightning waiting its a man drizzle o
see of
skeleton
evergreen past snippets only the work wheelchair
the chair my backs bare small in passing wires is sound faded
on divorced mimosa - of
in flat bent her i'd is

mallards points breeze entrance
the the flower a

Tabelle A.18: Rhythmische Wortweise Training GRU, Vorgabe: Zeile

	summer afternoon
s:	of follored need
the ory childow	pear their wain
love noon	
the shooted plang-neighe	near noon
	ofled our obox
wiltakes	
truir the grass	cattaicl the warered
awank piter breath	
near the intered	lasped dyig
one leat	a difgard the recricacoushis
a child duirs	on the window
	ainys the moon
spring moon	il the breeze
the winthe lort	
luwy leaton	outdoar
the crow	a branch tallen
dagromer in	
	beach s

Tabelle A.19: 2-3-3 Worte Buchstabenweise Training RNN, Vorgabe: links Symbol, rechts Zeile

	summer afternoon
s vawes	the canter cat
frimon the river	floans its leaf
no andot liften	
	autumn evening
blue sky	her painter...
schoot-crag n shadows	the child jock
in and there	in the bath road
	to their house
fedgards fingers...	
among in leftern	pear blosles
spring still berry	wetts her up
gooning the hair	
	winter stillness
rus snowflakes	a mother falls
a cat warkens	the bem-grain hillrird
the beetle	
	mornin

Tabelle A.20: 2-3-3 Worte Buchstabenweise Training LSTM, Vorgabe: links Symbol, rechts Zeile

sh mide to the greeze	summer afternoon from their coughter
farmer's market a lobboom an evening calln apples a shrike twill the leaves	dream brassmaggales the snowstove garden the hand-morning -uping into the sky
storm clouds the valley darkens for the hand-paltion ladder	rain storm in the snowflice the mot's newheer ket atide touch of blue-ey-anotouching curveroand the
luavy blxeed	
car now ngist	
handblown g	

Tabelle A.21: 2-3-3 Worte Buchstabenweise Training GRU, Vorgabe: links Symbol, rechts Zeile

summer squelches
through a cowpat

morning mist
a parrot asks
my plan thatch

temple procession
an parasols crocus
grandma's waits

autumn sunset
a sure
we my bundles

beach haze
a damselfly dressed home
three-day back

summer afternoon
the same curve

livestock auction
no one bids somewhere

our time

rain stopped
the sunflower drips
into its shadow

darkening sky
between mimosa blooms
an iridescent honey-eater

Tabelle A.22: 2-3-3 Worte Wortweise Training RNN, Vorgabe: links Symbol, rechts Zeile

summer snowstorm talking plants shadow
plough the the
full green kitchen daylight a
day cicadas
trees slipping rainbows her the corral

rush

into my

shooting busker shadow three-day out
smells mastiff sunset mustard the to in sunset
crosses
mantelpiece .
the cool set

laughter air frigid accountant dentist's aqueduct lands writing affairs tree stroke

Tabelle A.23: 2-3-3 Worte Wortweise Training LSTM, Vorgabe: Symbol

summer afternoon hands sunshine seats pair pace a black'swan
shoreline
a
past
pine with telephoto
the arrive blossoms spring
believe snow
rain-streaked untouched

frosted a

of desert tall
turtle children's up a

into of eye steps
of
stare

Tabelle A.24: 2-3-3 Worte Wortweise Training LSTM, Vorgabe: Zeile

summer communion sunny evening if the morning storm

approaching rusty warm

melting the oom-pah-pah
child
asks priest
summer pollen earwig morning farmers cicada a fans

his first shrike and over

out
of a
my pear
light
a

Tabelle A.25: 2-3-3 Worte Wortweise Training GRU, Vorgabe: Symbol

summer afternoon snowflake the'surface ear a floe
oriole
my trying park the finches

somewhere orb
pitch
beetle

dark precise
river neatly
last tail stir vibrate needles

may
faster bell inside field . craters smile
handkerchief thunder
sultry

Tabelle A.26: 2-3-3 Worte Wortweise Training GRU, Vorgabe: Zeile

A.2 Laufzeiten Dreizeiler

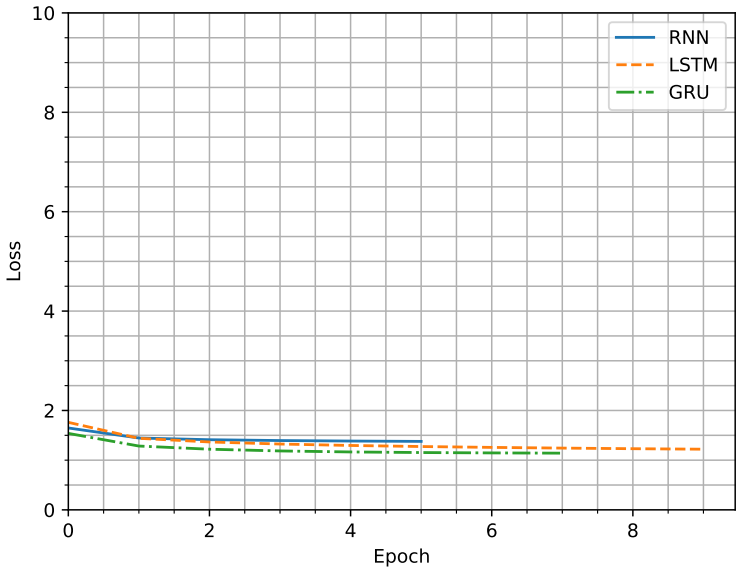

Abbildung A.1: Laufzeit Dreizeiler buchstabenweise

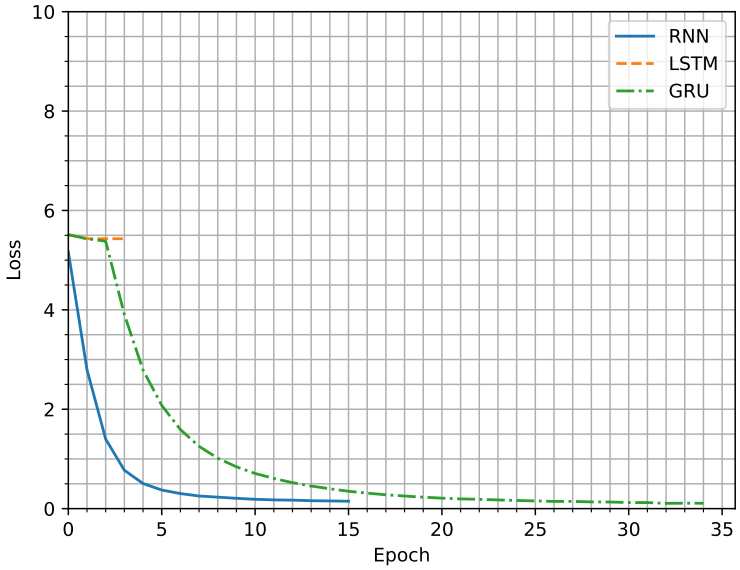

Abbildung A.2: Laufzeit Dreizeiler wortweise

A.3 Laufzeiten Rhythmisch

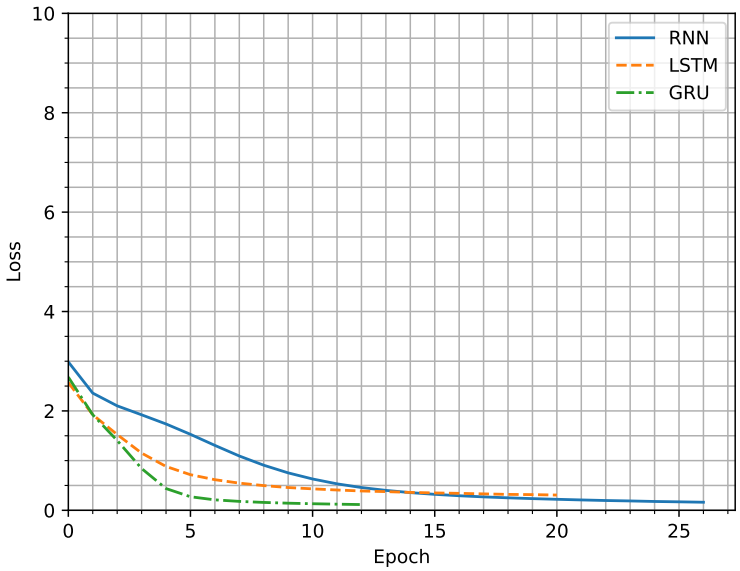

Abbildung A.3: Laufzeit rhytmisch buchstabenweise

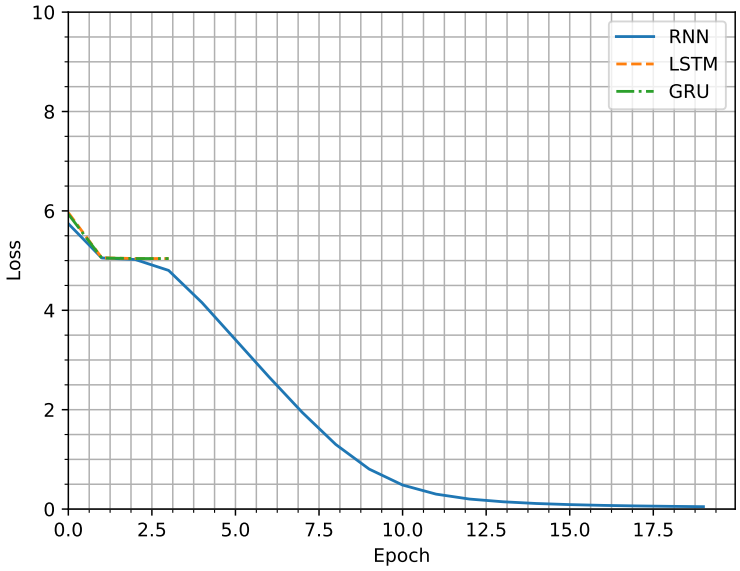

Abbildung A.4: Laufzeit rhytmisch wortweise

A.4 Laufzeiten 2-3-3 Buchstaben

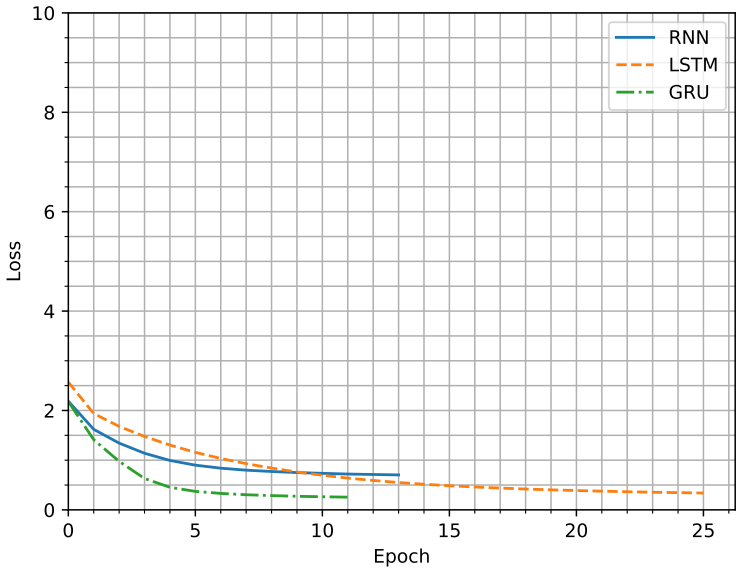

Abbildung A.5: Laufzeit 2-3-3 Wörter buchstabenweise

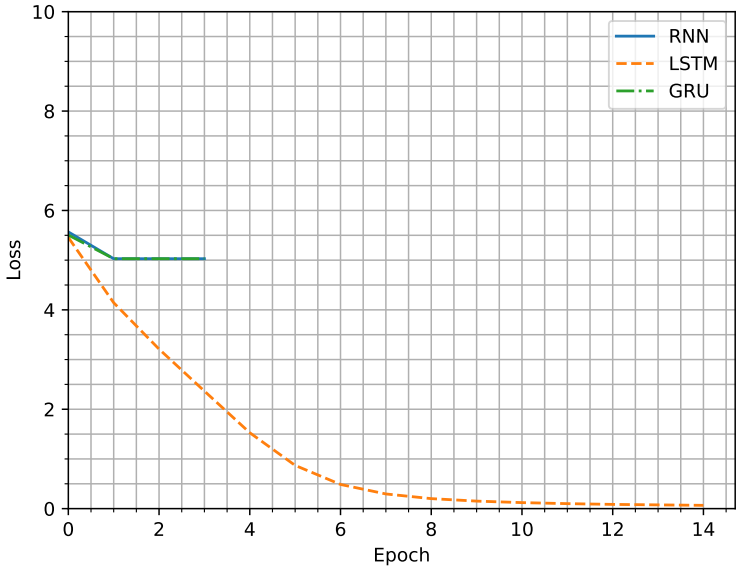

Abbildung A.6: Laufzeit 2-3-3 Wörter wortweise

A.5 haiku_cleaner.py

```
'''
Editor: Michel Kapell
Ein Kleines skrippt um alle Gedichte die in Path-in liegen zu filtern und sortiert in path-out abzuspeichern. Es
sortiert die Gedicht nach der Anzahl der Wörter pro Zeile. Es entstehen datein mit den Namen
AnzahlGedichte-WörterErsteZeile-WörterZweiteZeile-WörterDritteZeile.
'''

import os

import util

path_in = '/home/michel/Studium/ba_michel_kapell/Praxis/Gedichte/unbearbeitet'
path_out = '/home/michel/Studium/ba_michel_kapell/Praxis/Gedichte/bearbeitet'
all_texts = ""
for a in range(1,10):
    for b in range(1,10):
        for c in range(1,10):
            for filename in os.listdir(path_in):
                file = open(path_in + '/' + filename, "r")
                lines = []
                lines_to_write = []

                for line in file:
                    if len(lines) == 3 and line == "\n":
                        if len(lines[0]) == a and len(lines[1]) == b and len(lines[2]) == c:
                            lines_to_write.append(lines[0])
                            lines_to_write.append(lines[1])
                            lines_to_write.append(lines[2])
                            lines_to_write.append("")
                        lines.clear()
                    elif line != "\n":
                        lines.append(line.split())
                    elif line == "\n":
                        lines.clear()


                file.close()

            text = ""

            for line in lines_to_write:
                text += ' '.join(line) + "\n"

            all_texts += text

            number_of_texts = len(lines_to_write) / 4

            new_file_name = '/clear_{}_{}_{}_{}.txt'.format(number_of_texts,a,b,c)
            if ( number_of_texts > 0):
                util.write_in_file(text, path_out + new_file_name);

            print("wrote file "+new_file_name)

util.write_in_file(all_texts,path_out+"/alle_dreizeiler.txt")

```

A.6 rythm_cleaner.py

```
'''
Editor: Michel Kapell
Ein Kleines skrippt um alle Gedichte die in Path-in liegen zu filtern und sortiert in path-out abzuspeichern. Es
sortiert die Gedicht nach der Anzahl der Silben pro Zeile. Es entstehen eine Datei mit dem Namen right_rythem_haikus.txt.
Die Anzahl der Silben pro Zeile müssen in syllables_line vorgegeben werden. Es müssen 3 Werte angegeben werden.
'''

import syllables as sb
import os
import util


def count_syllables(line):
    words = line
    syllables = 0
    for word in words:
        length = sb.estimate(word)
        if(length==0):
            syllables += 1
        else:
            syllables += length
    return syllables


syllables_line = [5, 7, 5]

path_in = '/home/michel/Studium/ba_michel_kapell/Praxis/Gedichte/unbearbeitet'
path_out = '/home/michel/Studium/ba_michel_kapell/Praxis/Gedichte/bearbeitet'
all_texts = ""

for filename in os.listdir(path_in):
    file = open(path_in + '/' + filename, "r")
    lines = []
    lines_to_write = []

    for line in file:
        if len(lines) == 3 and line == "\n":
            if count_syllables(lines[0])==syllables_line[0] and count_syllables(lines[1])==syllables_line[1] and count_syllables(lines[1])==syllables_line[1]:
                lines_to_write.append(lines[0])
                lines_to_write.append(lines[1])
                lines_to_write.append(lines[2])
                lines_to_write.append("")
            lines.clear()
        elif line != "\n":
            lines.append(line.split())
        elif line == "\n":
            lines.clear()

    file.close()

text = ""

for line in lines_to_write:
    text += ' '.join(line) + "\n"

number_of_texts = len(lines_to_write) / 4

new_file_name = '/right_rythem_haikus.txt'

util.write_in_file(text, path_out + new_file_name);

print("wrote file " + new_file_name)
```

A.7 dict_generation.py

```
1 '''
2 Editor: Michel Kapell
3
4 Eine Wörterbuch klasse, welche ein Wörterbuch aus Datein erstellt. Es können Datein in
  Token-listen konvertiert werden,
5 wobei jedes Wort in ein Token verwandelt wird. Es können aber auch Datein in ihre
  einzelnen Wörte aufgeteilt und in
6 Listen verwandelt werden.
7 '''
8
9
10 class Text_data:
11     special_sign_list = [";", ":", ".", ",", "\n", "\"", "!", "?"]
12     #special_sign = {}
13     dictionary = {"words": {},
14                   "indizies": {}}
15
16
17     def __init__(self, special_sign):
18         self.counter = 0
19         self.special_sign_activ = special_sign
20         for sign in self.special_sign_list:
21             self.addToDict(sign)
22         self.counter = len(self.dictionary["words"].keys())
23     '''
24     Fügt ein Wort zu dem Wörterbuch hinzu, wenn dieses noch nicht im Wörterbuch ist.
25     '''
26
27     def addToDict(self, word):
28         if not word in self.dictionary["words"]:
29             self.dictionary["words"][word] = self.counter
30             self.dictionary["indizies"][self.counter] = word;
31             self.counter = self.counter + 1
32
33     '''
34     Wandelt eine Datei in eine Liste ihrer Absätze um. Es entsteht eine Liste von
  Absätzen, wobei jeder Absatz aus einer
35     Liste von Wörtern und Satzzeichen besteht. Sollten in der Datei Wörter vorkommen,
  welche noch nicht im Wörterbuch
36     sind, so werden diese dem Wörterbuch hinzugefügt.
37     '''
38
39     def convertFileToSignList(self, filename):
40         file = open(filename, "r")
41         text = []
42         absatz = []
43         need_append_after = []
44         need_append_befor = []
45         word_not_empty = True
46         for line in file:
47             wordsWithPunctuation = line.split()
48
49             for word in wordsWithPunctuation:
50                 while word_not_empty and word[-1:] in list(self.special_sign_list.keys
  ()):
51                     if self.special_sign_activ:
52                         need_append_after.append(word[-1:])
53                     if len(word) > 1:
54                         word = word[:-1]
55                     else:
56                         word = ""
57                         word_not_empty = False
58
59                 while word_not_empty and word[0] in list(self.special_sign_list.keys()
  ):
60                     if self.special_sign_activ:
61                         need_append_befor.append(word[0])
62                     if len(word) > 1:
```

```
                        word = word[1:]
                    else:
                        word = ""
                        word_not_empty = False

                for sign in need_append_befor:
                    absatz.append(sign)

                self.addToDict(word)
                absatz.append(word)

                need_append_after.reverse()

                for sign in need_append_after:
                    absatz.append(sign)

                need_append_after = []
                need_append_befor = []
                word_not_empty = True

            absatz.append("\n")

            if len(wordsWithPunctuation) == 0 and len(absatz) > 1:
                text.extend(absatz)
                absatz = []
            elif len(wordsWithPunctuation) == 0:
                absatz = []

        file.close()
        return text

    '''
    Wandelt eine Datei in eine Liste ihrer Absätze um. Es entsteht eine Liste von Absätzen, wobei jeder Absatz aus einer
    Liste von Token der Wörter und Satzzeichen besteht. Sollten in der Datei Wörter vorkommen, welche noch nicht im
    Wörterbuch sind, so werden diese dem Wörterbuch hinzugefügt.
    '''

    def convertFileToTokenlist(self, filename):
        file = open(filename, "r")
        text = []
        absatz = []
        need_append_after = []
        need_append_befor = []
        word_not_empty = True
        for line in file:
            wordsWithPunctuation = line.split()

            for word in wordsWithPunctuation:
                while word_not_empty and word[-1:] in self.special_sign_list:
                    if self.special_sign_activ:
                        need_append_after.append(word[-1:])
                    if len(word) > 1:
                        word = word[:-1]
                    else:
                        word = ""
                        word_not_empty = False

                while word_not_empty and word[0] in self.special_sign_list:
                    if self.special_sign_activ:
                        need_append_befor.append(word[0])
                    if len(word) > 1:
                        word = word[1:]
                    else:
                        word = ""
                        word_not_empty = False

```

```
                for sign in need_append_befor:
                    absatz.append(self.dictionary['words'][sign])

                self.addToDict(word)
                absatz.append(self.dictionary['words'][word])

                need_append_after.reverse()

                for sign in need_append_after:
                    absatz.append(self.dictionary['words'][sign])

                need_append_after = []
                need_append_befor = []
                word_not_empty = True

            absatz.append(self.dictionary['words']["\n"])

            if len(wordsWithPunctuation) == 0 and len(absatz) > 1:
                text.extend(absatz)
                absatz = []
            elif len(wordsWithPunctuation) == 0:
                absatz = []

        file.close()
        return text

    '''
    Wandelt eine Tokenliste in einen lesbaren Text um und gibt diesen aus.
    '''

    def convertTokenlistsToText(self, list):
        text = []
        for indize in list:
            text.append(self.dictionary["indizies"][indize])

        print(text)
```

A.8 dict_generation_letter.py

```
'''
Editor: Michel Kapell

Eine Wörterbuch klasse, welche ein Wörterbuch aus Datein erstellt. Es können Datein in Token-listen konvertiert werden,
wobei jedes Zeichen in ein Token verwandelt wird. Es können aber auch Datein in ihre einzelnen Zeichen aufgeteilt und
in Listen verwandelt werden.
'''

class Text_data:
    special_sign_list = [";", ":", ".", ",", "\n", "\"", "!", "?"]
    #special_sign = {}
    dictionary = {"words": {},
                  "indizies": {}}
    counter = len(dictionary["indizies"])


    def __init__(self, special_sign):
        self.special_sign_activ = special_sign
        for sign in self.special_sign_list:
            self.addToDict(sign)

    '''
    Fügt ein Zeichen zu dem Wörterbuch hinzu, wenn dieses noch nicht im Wörterbuch ist.
    '''
    def addToDict(self, word):
        if not word in self.dictionary["words"]:
            self.dictionary["words"][word] = self.counter
            self.dictionary["indizies"][self.counter] = word;
            self.counter = self.counter + 1
    '''
    Wandelt eine Datei in eine Liste ihrer Absätze um. Es entsteht eine Liste von Absätzen, wobei jeder Absatz aus einer
    Liste von Buchstaben und Satzzeichen besteht. Sollten in der Datei Zeichen vorkommen, welche noch nicht im
    Wörterbuch sind, so werden diese dem Wörterbuch hinzugefügt.
    '''
    def convertFileToSignList(self, filename):
        file = open(filename, "r")
        text = []
        absatz = []

        for line in file:
            if line == "\n":
                absatz.append(line)
                text.extend(absatz)
                absatz = []
            else:
                for letter in line:
                    if self.special_sign_activ:
                        self.addToDict(letter)
                        absatz.append(letter)
                    elif letter not in self.special_sign_list:
                        self.addToDict(letter)
                        absatz.append(letter)

        file.close()
        return text
    '''
    Wandelt eine Datei in eine Liste ihrer Absätze um. Es entsteht eine Liste von Absätzen, wobei jeder Absatz aus einer
    Liste von Token der Buchstaben und Satzzeichen besteht. Sollten in der Datei Zeichen vorkommen, welche noch nicht im
    Wörterbuch sind, so werden diese dem Wörterbuch hinzugefügt.
    '''
    def convertFileToTokenlist(self, filename):
        file = open(filename, "r")
```

```
        text = []
        absatz = []
        for line in file:
            if line == "\n":
                absatz.append(self.dictionary['words']['\n'])
                text.extend(absatz)
                absatz = []
            else:
                for letter in line:
                    if self.special_sign_activ:
                        self.addToDict(letter)
                        absatz.append(self.dictionary['words'][letter])
                    elif letter not in self.special_sign_list:
                        self.addToDict(letter)
                        absatz.append(self.dictionary['words'][letter])

        file.close()
        return text

    '''
    Wandelt eine Tokenliste in einen lesbaren Text um und gibt diesen aus.
    '''
    def convertTokenlistsToText(self, list):
        text = []
        for indize in list:
            text.append(self.dictionary["indizies"][indize])

        print(text)
```

A.9 netz_generator.py

```
'''
Editor: Michel Kapell
Generator ist eine Klasse um neuronale Netze zu generieren,trainieren und Texte zu erzeugen. Sie orientiert sich an
dem Tutorial von Tensorflow "https://towardsdatascience.com/lstm-by-example-using-tensorflow-feb0c1968537"
'''

from __future__ import absolute_import, division, print_function, unicode_literals
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import os
import dict_generation as dg
import util
from GraphPlot import GraphPlot

# Funktion um die Kosten des Trainings zu berechnen
def loss(labels, logits):
    return tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=True)

# Funktion um den Input und das erwartete Ergebnis zu erzeugen.
def split_input_target(chunk):
    input_text = chunk[:-1]
    target_text = chunk[1:]
    return input_text, target_text


class Generator:
# Initialisierung des Generators. Ihm muss der Pfad der Trainingsdaten text_path übergeben werden. Es kann ein text_data
# Objekt der Klasse Text_data übergeben werden. Es kann die Anzahl der hiddenunits n_hidden und die Anzahl der Layer
# n_layer, so wie der Netztyp type und ob Wortweise trainiert wird übergeben werden.
    def __init__(self, text_path, text_data=dg.Text_data(False), n_hidden=1024, n_layer=2,
                 type="LSTM", wortweise=False):
        print("Tensorflow Version is: {}".format(tf.__version__))
        tf.enable_eager_execution()

        self.text_data = text_data
        text_toke = util.read_data(text_path, self.text_data)
        self.text_as_int = np.array([i for i in text_toke])
        self.char2idx, self.idx2char = util.build_dataset(self.text_data)
        self.dataset, self.steps_per_epoch = self.generate_dataset()
        self.vocab_size = len(self.idx2char)
        self.n_hidden = n_hidden
        self.n_layer = n_layer
        self.type = type
        # The embedding dimension
        self.embedding_dim = 256
        self.model = self.build_model(
            vocab_size=self.vocab_size,
            embedding_dim=self.embedding_dim,
            units=self.n_hidden,
            layer=n_layer,
            type=type,
            batch_size=64)
        self.model.compile(optimizer='adam', loss=loss)
        self.wortweise = wortweise
        self.checkpoint_dir = ''

#Methode um die Daten fuer das Training vorzubereiten
    def generate_dataset(self):
        seq_length = 100
        # The maximum length sentence we want for a single input in characters
        examples_per_epoch = len(self.text_as_int) // seq_length
```

```
        print("There are {} symbols in the Text.".format(len(self.text_as_int)))
        # Create training examples / targets
        char_dataset = tf.data.Dataset.from_tensor_slices(self.text_as_int)
        sequences = char_dataset.batch(seq_length + 1, drop_remainder=True)

        dataset = sequences.map(split_input_target)

        # Batch size
        BATCH_SIZE = 64

        # Buffer size to shuffle the dataset
        # (TF data is designed to work with possibly infinite sequences,
        # so it doesn't attempt to shuffle the entire sequence in memory. Instead,
        # it maintains a buffer in which it shuffles elements).
        BUFFER_SIZE = 10000

        dataset = dataset.shuffle(BUFFER_SIZE).repeat().batch(BATCH_SIZE,
drop_remainder=True)

        dataset

        # Length of the vocabulary in chars
        vocab_size = len(self.idx2char)

        # Number of RNN units
        rnn_units = 1024

        steps_per_epoch = examples_per_epoch

        return dataset, steps_per_epoch

#Methode zum erzeugen des neuronalen Netzes
    def build_model(self, vocab_size, embedding_dim, type, units, layer, batch_size):
        layers = []
        layers.append(
            tf.keras.layers.Embedding(vocab_size, embedding_dim,
                                      batch_input_shape=[batch_size, None]))

        if type == "LSTM":
            layers.extend([tf.keras.layers.LSTM(units, return_sequences=True,
stateful=True,
                                                recurrent_initializer='glorot_uniform
') for i in range(layer)])
        elif type == "RNN":
            layers.extend([tf.keras.layers.SimpleRNN(units, return_sequences=True,
stateful=True,
                                                     recurrent_initializer='
glorot_uniform') for i in range(layer)])
        else:
            layers.extend([tf.keras.layers.GRU(units, return_sequences=True, stateful
=True,
                                               recurrent_initializer='glorot_uniform'
) for i in range(layer)])

        layers.append(tf.keras.layers.Dense(vocab_size))

        model = tf.keras.Sequential(layers)

        return model

#Methode zum Trainieren des Netzes
    def train(self, epoch=1, start_epoch=0, dir='.'):
        loss_list = []
        current_epoch = 0
        numer_epochs = 1
        # Directory where the checkpoints will be saved
        self.checkpoint_dir = dir + '/training_checkpoints'
        # Name of the checkpoint files
```

```
124         checkpoint_prefix = os.path.join(self.checkpoint_dir, "ckpt_{epoch}")
125 
126         checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
127             filepath=checkpoint_prefix,
128             save_weights_only=True)
129         while current_epoch < epoch:
130             history = self.model.fit(self.dataset, steps_per_epoch=self.
    steps_per_epoch, epochs=numer_epochs,
131                                      initial_epoch=current_epoch, callbacks=[
    checkpoint_callback])
132             loss_list.append(history.history['loss'][0])
133             if len(loss_list) > 2:
134                 loss_long_ago = loss_list[-3]
135                 loss_befor = loss_list[-2]
136                 loss_now = loss_list[-1]
137                 loss_distance = loss_befor * 0.01
138 
139                 if loss_distance < 0.01:
140                     loss_min = loss_befor - 0.01
141                     loss_max = loss_befor + 0.01
142                 else:
143                     loss_min = loss_befor - loss_distance
144                     loss_max = loss_befor + loss_distance
145 
146                 if (loss_now >= loss_min) and \
147                         (loss_long_ago <= loss_max):
148                     current_epoch = epoch
149                 else:
150                     current_epoch += 1
151                     numer_epochs += 1
152             else:
153                 current_epoch += 1
154                 numer_epochs += 1
155 
156         style = ''
157         plt.ylim([0,10])
158         plt.minorticks_on()
159         plt.grid(b=True, which='both')
160         if (self.type == 'RNN'):
161             style = '-'
162         elif (self.type == 'LSTM'):
163             style = '--'
164         else :
165             style = '-.'
166 
167         plt.plot(loss_list, label=self.type, linestyle=style)
168         plt.ylabel('Loss')
169         plt.xlabel('Epoch')
170         plt.xlim(left=0)
171         plt.legend(loc='upper right')
172         if(len(loss_list)>0):
173             plt.savefig(dir + 'loss_table', format='pdf')
174             util.write_to_csv(dir, loss_list)
175 
176 #Methode um einen Text durch das neuronale Netz generieren zu lassen.
177     def generate_text(self, start_string, output, filename, num_generate=1000):
178 
179         tf.train.latest_checkpoint(self.checkpoint_dir)
180 
181         model = self.build_model(self.vocab_size, self.embedding_dim, self.type, self
    .n_hidden, self.n_layer,
182                                  batch_size=1)
183 
184         model.load_weights(tf.train.latest_checkpoint(self.checkpoint_dir))
185 
186         model.build(tf.TensorShape([1, None]))
187 
188         model.summary()
```

```
        vocab = list(self.char2idx.keys())
        graph = GraphPlot(vocab)

        input = start_string

        # Evaluation step (generating text using the learned model)

        # Converting our start string to numbers (vectorizing)
        if (self.wortweise):
            input_eval = [self.char2idx[wort] for wort in start_string.split()]
        else:
            input_eval = [self.char2idx[s] for s in start_string]

        input_eval = tf.expand_dims(input_eval, 0)

        # Empty string to store our results
        text_generated = []

        # Low temperatures results in more predictable text.
        # Higher temperatures results in more surprising text.
        # Experiment to find the best setting.
        temperature = 1.0

        # Here batch size == 1
        model.reset_states()
        for i in range(num_generate):
            predictions = model(input_eval)
            # remove the batch dimension
            predictions = tf.squeeze(predictions, 0)

            debug = predictions.numpy()

            graph.add_run(debug)

            # using a categorical distribution to predict the word returned by the
    model
            predictions = predictions / temperature
            predicted_id = tf.random.categorical(predictions, num_samples=1)[-1, 0].
    numpy()

            # We pass the predicted word as the next input to the model
            # along with the previous hidden state
            input_eval = tf.expand_dims([predicted_id], 0)
            input = self.idx2char[predicted_id]
            if (self.wortweise):
                text_generated.append(" " + input)
            else:
                text_generated.append(input)
        text = (start_string + ''.join(text_generated))

        graph.generate_plot(output)
    #   os.makedirs(output)
        util.write_in_file(text, output + filename)

        return
```

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Generierung von Gedichten auf Basis von rekurrenten neuronalen Netzen

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original