Humboldt Gymnasium Vaterstetten

Oberstufenjahrgang 2021/2023

Seminararbeit

aus dem Fach Physik

Thema: Komponieren von Jazzmusik mit RNN und LSTM Zellen

Verfasser:	Justin Hohenstein
W-Seminar:	Mathematik und Physik in der Musik
Seminarleiter:	Sebastian Bauer
Abgabetermin:	8. November 2022

Erzielte Note:		in Worten:	
Erzielte Punkte:		in Worten:	

(einfache Wertung)

Abgabe beim Oberstufenbetreuer am:

Unterschrift des Seminarleiters

Inhaltsverzeichnis

1 Computer und kreative Tätigkeiten

Während in immer mehr Bereichen Maschinen die Arbeit von Menschen übernehmen, waren es hauptsächlich kreative Tätigkeiten, bei denen Maschinen Menschen deutlich unterlegen waren. Doch in den letzten Jahren gibt es immer mehr Projekte, die dank den Fortschritten in Machine Learning, künstlicher Kreativität immer näher kommen. Auch im Bereich der Musik wird schon seit langem mithilfe von künstlicher Intelligenz Musik komponiert. Diese Stücke sind zwar für die meisten Menschen noch leicht zu identifizieren, allerdings sind diese meist besser als das, was ein durchschnittlicher Mensch komponieren kann. Ziel dieser Arbeit ist es, mithilfe, vergleichsweise einfacher Algorithmen, dieses Problem anzugehen und dadurch das Potenzial aufzuzeigen, das künstliche Intelligenz auch in kreativen Tätigkeiten hat. Dazu soll ein Programm geschrieben werden, das anhand von bereits existierender Jazzmusik neuartige Stücke komponieren kann.

2 Approximieren von Funktionen

2.1 Musik als Funktion

So gut wie jede Problemstellung kann in eine Funktion $f(x) = y$ umgewandelt werden. Alle Informationen, die zur Lösung der Aufgabenstellung benötigt werden, werden dazu für die Variable x eingesetzt. Die Variable y ist das Ergebnis dieser Funktion und enthält die Lösung der Problemstellung. Auch ein Musikstück kann durch eine theoretische Funktion $f(x) = y$ beschrieben werden. Setzt man in diese Funktion für x eine beliebige Note aus dem Stück ein, so soll y die Note sein, die darauf folgt. Soll beispielsweise die Sequenz

a-c-h-a

beschrieben werden, so muss für die Funktion $f(x)$ gelten:

$$f(\text{"a"}) = \text{"c"}$$
$$f(\text{"c"}) = \text{"h"}$$
$$f(\text{"h"}) = \text{"a"}$$

Da Funktionen nicht mit Noten, sondern nur mit Zahlen arbeiten können, müssen diese erst umgewandelt werden. Dazu wird jeder Note genau ein Index von 1 bis n zugeordnet, wobei n die Anzahl an unterschiedlichen Noten ist. Die Reihenfolge ist dabei egal, solange jeder Index genau zu einer Note gehört. In diesem Beispiel könnte diese Zuordnung so aussehen:

Note	Index
a	1
h	2
c	3

Ersetzt man nun die Noten mit ihren zugehörigen Indexen, so muss für die Sequenz gelten:

$$f(1) = 2$$
$$f(2) = 4$$
$$f(4) = 3$$

Um diese theoretische Funktion $f(x)$ zu finden, kann ein sogenanntes künstliches neuronales Netz verwendet werden. Dieses kann jede stetige Funktion $f(x)$ beliebig genau approximieren (nach Nielsen (2019) Kapitel 4). In einer vereinfachter Version wird dies als folgende Funktion definiert:

$$g(x, p) = \hat{y}$$

Neben der Variable x hat diese Funktion als Eingabe einen Parameter p. Jede Problemstellung hat einen, oder mehrere, spezifische Werte für p, für die gilt:

$$f(x) \approx g(x, p)$$
$$y \approx \hat{y}$$

Findet man also einen richtigen Wert für p, so hat man eine Funktion $g(x, p)$ gefunden, die ein Musikstück beschreibt. Mithilfe dieser Funktion kann dann für jede Note vorhergesagt werden, welche Note wahrscheinlich folgen wird. Dies soll in Sektion 7.3 verwendet werden, um neue Musik zu komponieren. Das finden dieses optimalen Wertes von Parametern nennt man Lernen.

2.2 Kostenfunktion

Um den optimalen Wert für p zu finden, ist es nützlich zu wissen, wie stark das approximierte Ergebnis $\hat{y}$ vom richtigen Ergebnis y durchschnittlich für jede Note des Stücks abweicht. Dazu wird jede Noten einzeln in x der Funktion $g(x, p)$ eingesetzt. Das Ergebnis $\hat{y}$ ist die Note, die das neuronale Netz als folgende Note vorhersagt. Der Unterschied zwischen dieser Vorhersage und der tatsächlichen Note y wird mithilfe der Funktion E ausgerechnet:

$$E(\hat{y}, y) = (\hat{y} - y)^2 \tag{1}$$

Je größer ihre Differenz, desto größer ist der Fehler und auch die Funktion E. Diese Gleichung liefert den Fehler von einem Zeitpunkt. Es soll allerdings der gesamte durchschnittliche Fehler für ein Stück gefunden werden. Dazu wird die mittlere quadratische Abweichung verwendet:

$$C = \frac{1}{T} \sum_{t=1}^{T} E_t \tag{2}$$

Die Variable T ist hierbei die Anzahl aller Noten. Für jede Note im Stück wird an Stelle t der Fehler E_t mithilfe der Gleichung 1 gebildet. Alle Fehler werden summiert und durch die Anzahl T dividiert, um den durchschnittlichen Fehler zu bilden.

2.3 Gradientenabstieg

Je kleiner der Fehler C ist, desto kleiner ist der Unterschied zwischen der Vorhersage und dem richtigen Ergebnis und desto genauer wird die ursprüngliche Funktion $f(x)$ approximiert. Es ist also Ziel des Programms, diesen Wert zu minimieren. Eine Möglichkeit dazu heißt Gradientenabstieg. Dafür wird zunächst die Ableitung von C nach p gebildet: $\frac{\partial C}{\partial p}$. Ist diese positiv, so bedeutet dies durch die Eigenschaften der Ableitung, dass eine Verkleinerung von p um einen extrem kleinen Betrag eine Verkleinerung von C um einen ebenfalls extrem kleinen Betrag bedeutet. Ist $\frac{\partial C}{\partial p}$ negativ, so kann p um einen extrem kleinen Betrag vergrößert werden, um C zu verkleinern. Ein kleinerer Wert von C bedeutet einen kleineren Fehler in der Approximation. Das neuronale Netz kann also durch wiederholte Anwendung folgender Formel seine Approximation verbessern und somit lernen:

$$p_{\text{neu}} = \begin{cases} p_{\text{alt}} - \eta & \text{wenn } \frac{\partial C}{\partial p_{\text{alt}}} > 0 \\ p_{\text{alt}} + \eta & \text{wenn } \frac{\partial C}{\partial p_{\text{alt}}} \leq 0 \end{cases}$$

Dabei ist η ein sogenannter Hyperparameter, der angibt, wie groß diese extrem kleine Veränderung sein soll. In dieser Form macht das Netz immer gleich große Veränderungen. Sinnvoll wäre es allerdings, bei einer größeren Ableitungen ebenfalls große Veränderungen zu machen und bei einer kleineren Ableitung nur kleine Veränderungen zu machen. Dazu wird die Veränderung durch folgende Formel von der Größe der Ableitung abhängig:

$$p_{\text{neu}} = p_{\text{alt}} - \frac{\partial C}{\partial p_{\text{alt}}} \cdot \eta \tag{3}$$

Die Bedingung fällt hierbei weg, da durch die Multiplikation mit der Ableitung das Vorzeichen automatisch angepasst wird. Auch in dieser Formel findet sich der Hyperparameter η. Dieser muss für jedes Problem einzeln experimentell bestimmt werden. Ist er zu klein, so findet fast keine Veränderung statt und das Netz lernt gar nicht oder nur sehr langsam. Ist dieser zu groß, so stimmt in der Regel die Approximation $\frac{\partial C}{\partial p} \cdot \Delta p \approx \Delta C$ nicht mehr und der Fehler wird größer.

3 Neuronale Netze

3.1 Funktionsweise von neuronalen Netzen

Dem folgende Kapitel liegt das Buch Nielsen (2019) zugrunde. In Sektion 2.1 wurde ein neuronales Netz als Funktion $g(x, p) = \hat{y}$ definiert. In den meisten Fällen wird das Lernen allerdings erleichtert, wenn anstatt der eindimensionalen Variablen x und y Vektoren verwendet werden. Außerdem ist es einfacher, die Variable p durch mehrere Parameter P zu ersetzen. Dieses Erweitern der Variablen zu Vektoren hat zur Folge, dass der Informationsgehalt der pro Element aus einem solchen Vektor geringer ist und dadurch das Finden einer Approximation erleichtert wird. Die neue Definition eines neuronalen Netzes lautet also:

$$g(X, P) = \hat{Y}$$

Da nun Vektoren, anstatt einzelnen Werten als Ergebnis vorliegen, kann der Fehler des neuronalen Netzes für eine Note nicht mehr mit Gleichung 1 ausgerechnet werden. Diese Funktion wird erweitert, indem für jedes Element des Ergebnisvektors der Fehler nach Gleichung 1 gebildet wird und diese anschließend summiert werden. Für eine Funktion mit einem Ergebnisvektor $\hat{Y}$ der Größe n^L lautet die Funktion E also:

$$\begin{aligned} E &= \sum_{i=1}^{n^L} (\hat{Y}_i - Y_i)^2 \\ &= (\hat{Y} - Y)^2 \end{aligned} \tag{4}$$

Die Funktionsweise von künstliche neuronale Netzen ähnelt der biologischer. Diese bestehen aus vielen einzelnen Neuronen, die Informationen bekommen, verarbeiten und anschließend an andere Neuronen weiter geben. Ein einzelnes Neuron verarbeitet Informationen mit folgender Formel:

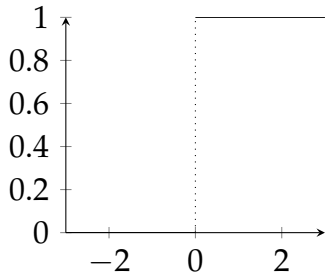
$$z = b + \sum_{k=1}^{n^{L-1}} w_k \cdot X_k \tag{5}$$

$$a = \sigma(z) \tag{6}$$

Dabei ist X ein Vektor der Größe n^{L-1}, der alle Informationen enthält, die das Neuron verarbeiten soll. Jede dieser Informationen ist unterschiedlich wichtig und soll daher einen unterschiedlich großen Einfluss auf das letztendliche Ergebnis haben. Daher wird jede dieser Informationen mit einem sogenanntem Gewicht aus dem Vektor w multipliziert. Ist ein Gewicht groß, so hat die dazugehörige Information einen großen Einfluss auf das letztendliche Ergebnis. Ist es klein, so hat sie einen geringen Einfluss. Aus allen gewichteten Informationen wird dann eine Summe gebildet, zu der eine weitere Variable, die Verschiebung b, addiert wird. Diese beeinflusst, wie hoch das Ergebnis des Neurons ohne Einfluss der gewichteten Informationen sein soll. Ist diese hoch, so ist das Ergebnis des Neurons auch bei kleinen oder unwichtigen Informationen groß. Als Ergebnis der Informationsverarbeitung hat ein biologisches Neuron immer entweder eine Null oder Eins. Um dieses Verhalten zu erreichen wird auf das Zwischenergebnis z eine sogenannte Aktivierungsfunktion σ mit folgender Formel angewandt:

$$\sigma(z) = \begin{cases} 1 & \text{wenn } z > 0 \\ 0 & \text{wenn } z \leq 0 \end{cases}$$

Der Graph dieser Funktion sieht so aus:

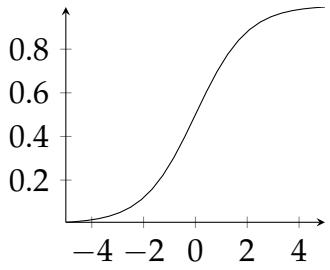


In Sektion 2.3 wurde erläutert, dass der Gradientenabstieg durch das Bilden der Ableitung $\frac{\partial C}{\partial p}$ und die Subtraktion dieser vom Parameter p funktioniert. Leitet diese Funktion ab, so ist diese an jeder Stelle null. Dadurch ist $\frac{\partial C}{\partial p}$ ebenfalls null

und der Gradientenabstieg nach Gleichung 3 bewirkt keine Veränderung. Daher wird stattdessen folgende Funktion verwendet:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{7}$$

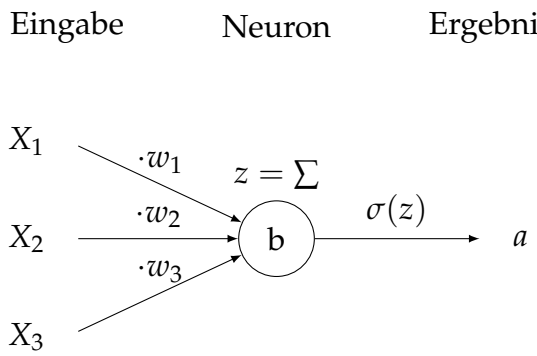
Mit folgenden Verlauf:



Ähnlich wie bei einem biologischen Neuron ist diese bei sehr negativen Werten null und bei sehr positiven Werten eins. Sie hat folgende Ableitung und löst damit das Problem:

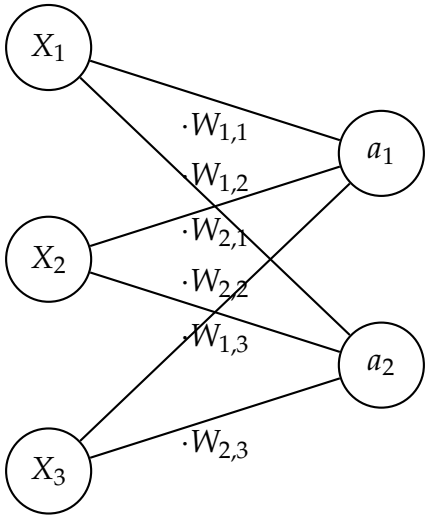
$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z) \cdot (1 - \sigma(z)) \tag{8}$$

Die Funktionsweise eines Neurons mit dem Eingabevektor X mit der Größe 3 kann grafisch so dargestellt werden:



Ein Neuron hat als Ergebnis genau einen Wert. Um als Endergebnis wieder auf einen Vektor zu kommen, wird eine sogenannte Schicht aus mehreren Neuronen gebildet. Jedes Neuron einer Schicht hat denselben Eingabevektor, verarbeitet diesen aber unterschiedlich. Das Ergebnis jedes Neurons ist ein Element des Ergebnisvektors der Schicht a. Um die Gewichte der unterschiedlichen Neuronen zu unterscheiden wird folgende Notation verwendet: $w_{j,k}$. Dabei ist j der

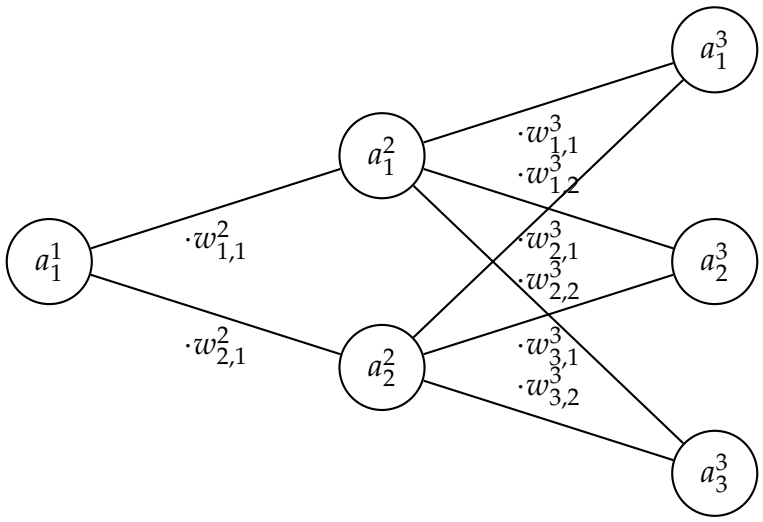
Index des Neurons in der Schicht und k der Index der Information, die es gewichtet. Für die Verschiebungen wird folgende Notation verwendet: b_j. Auch hier ist j der Index des Neurons, zu dem eine Verschiebung gehört. Eine Schicht mit einem Eingabevektor der Größe drei und einem Ergebnisvektor der Größe zwei kann beispielsweise so aussehen:



In dieser Abbildung wurden die Verschiebungen b_j zur Vereinfachung weg gelassen. In diesem Fall ist der Vektor a, der entsteht, das Gesamtergebnis des neuronalen Netzes $\hat{Y}$. In den meisten Fällen müssen mehrere Schichten verwendet werden, um eine gute Approximation zu erreichen. Um die Ergebnisse der unterschiedlichen Schichten zu unterscheiden, wird der Ergebnisvektor einer Schicht a^l genannt. Dabei ist l die Nummer der Schicht. Um mehrere Schichten miteinander zu verbinden, wird das Ergebnis von einer Schicht als Eingabe für die nächste Schicht verwendet. Es gilt also beispielsweise für die zweite Schicht:

$$X = a^1$$

Zur Unterscheidung der Parameter mehrere Schichten wird jeder Parameter zusätzlich mit der Nummer der Schicht l, in der er sich befindet, benannt. Die Verschiebungen heißen nun also b_j^l und die Gewichte heißen $w_{j,k}^l$. Das Gewicht $w_{2,1}^3$ befindet sich beispielsweise in der dritten Schicht und verbindet das zweite Neuron mit dem ersten Ergebnis a_1^2 aus der vorherigen Schicht. Werden mehrere Schichten verbunden, so ist das Gesamtergebnis des neuronalen Netzes das Ergebnis der letzten Schicht a^L, wobei L die Anzahl an Schichten ist. Ein neuronales Netz mit 3 Schichten kann beispielsweise so aussehen:



Passt man Gleichungen 5 und 6 für die Verwendung in einem Netz mit mehreren Schichten an, so ergeben sich für eine Schicht l folgende Gleichungen:

$$z_j^l = b_j^l + \sum_{k=1}^{n^{l-1}} w_{j,k}^l \cdot a_k^{l-1} \tag{9}$$

$$a_j^l = \sigma(z_j^l) \tag{10}$$

Als erste Schicht wird die Eingabe X verwendet. Das Ergebnis der letzten Schicht a^L ist das Gesamtergebnis des neuronalen Netz $\hat{Y}$. Es gilt also:

$$\begin{aligned} a^0 &= X \\ \hat{Y} &= a^L \end{aligned} \tag{11}$$

Eine Schicht besitzt mehrere Neuronen, die jeweils eine Verschiebung haben. Daher werden die Verschiebungen einer Schicht l in einem Vektor der Größe n^l gespeichert. Dieser sieht so aus:

$$b^l = \begin{pmatrix} b_1^l \\ b_2^l \\ \vdots \\ b_{n^l}^l \end{pmatrix} \tag{12}$$

Da eine Schicht mehrere Neuronen hat und jedes dieser mehr als ein Gewicht hat, werden die Gewichte einer Schicht in einer Matrix gespeichert. Jede Zeile

enthält dabei alle Gewichte eines einzelnen Neurons. Die Spalten jeder Zeile enthalten die Gewichte für dieses Neuron. Die Gewichte einer Schicht l mit n^l Neuronen und n^{l-1} Eingaben sieht so aus:

$$w^l = \begin{pmatrix} w^l_{1,1} & w^l_{1,2} & \dots & w^l_{1,n^{l-1}} \\ w^l_{2,1} & w^l_{2,2} & \dots & w^l_{2,n^{l-1}} \\ \vdots & \vdots & \ddots & \vdots \\ w^l_{n^l,1} & w^l_{n^l,2} & \dots & w^l_{n^l,n^{l-1}} \end{pmatrix} \tag{13}$$

3.2 Ableitung von neuronalen Netzen

In Sektion 2.3 wurde bereits erläutert wie der Gradientenabstieg mit einem einzelnen Parameter funktioniert. Neuronale Netze haben allerdings mehr als einen Parameter. Jedes Gewicht $w^l_{j,k}$ und jede Verschiebung b^l_j ist ein Parameter, für den ein optimaler Wert gefunden werden muss. Dies wird in neuronalen Netzen durch das Bilden der partiellen Ableitung erreicht. Die Herleitung der dafür benötigten Gleichungen orientiert sich an Kurbiel (2020). Es wird die Ableitung nach jedem einzelnen Parameter gebildet und anschließend der Gradientenabstieg nach Gleichung 3 durchgeführt. Dazu sollen zunächst die partiellen Ableitungen nach dem Ergebnis der letzten Schicht $\hat{Y} = a^L$ gefunden werden. In einem neuronalen Netz mit L Schichten und n^L Neuronen in der letzten Schicht wird also der Vektor

$$\begin{pmatrix} \frac{\partial C}{\partial \hat{Y}_1} \\ \frac{\partial C}{\partial \hat{Y}_2} \\ \vdots \\ \frac{\partial C}{\partial \hat{Y}_{n^L}} \end{pmatrix} \tag{14}$$

gesucht. Dazu wird zunächst die Ableitung der Funktion 2 gebildet:

$$\frac{\partial C}{\partial \hat{Y}} = \frac{1}{T} \sum_{t=1}^{T} \frac{\partial E_t}{\partial \hat{Y}_t} \tag{15}$$

Weiterhin muss für jede Note die Ableitung $\frac{\partial E_t}{\partial \hat{Y}_t}$ gebildet werden. Zur Vereinfachung wird im Folgenden diese partielle Ableitung als $\frac{\partial E}{\partial \hat{Y}}$ geschrieben. Sie muss allerdings für jeden Zeitpunkt t gebildet und addiert werden. Es wird nun al-

so die Ableitung nach $\hat{Y}$ gesucht. Da dies ein Vektor der Größe n^L ist, müssen die partiellen Ableitungen nach jedem seiner Elemente $\hat{Y}_j$ gebildet werden. Leitet man die Funktionen 4 ab, so erhält man nach der Kettenregel:

$$\frac{\partial E}{\partial \hat{Y}_j} = \sum_{i=1}^{n^L} 2(\hat{Y}_i - Y_i) \cdot \frac{\partial \hat{Y}_i}{\partial \hat{Y}_j} \tag{16}$$

Das Ergebnis ist dabei eine Summe. Jedes Element dieser Summe wird mit einer partiellen Ableitung $\frac{\partial \hat{Y}_i}{\partial \hat{Y}_j}$ multipliziert. Da ein Ergebnis $\hat{Y}_j$ nicht den anderen Ergebnissen Y_i abhängt ist der Inhalt der Summe null, außer wenn gilt: $i = j$ und daher die Ableitung $\frac{\partial \hat{Y}_j}{\partial \hat{Y}_j}$ gleich eins ist. Der Term lässt sich also wie folgt vereinfachen:

$$\frac{\partial E}{\partial a_j^L} = \frac{\partial E}{\partial \hat{Y}_j} = 2(\hat{Y}_j - Y_j) \tag{17}$$

Da nach Gleichung 11 gilt: $\hat{Y} = a^L$, so gilt auch: $\frac{\partial E}{\partial \hat{Y}} = \frac{\partial E}{\partial a^L}$. Mit dieser partiellen Ableitung können nun die partiellen Ableitungen nach den Parametern in der Schicht a^L gefunden werden. Für die Verschiebungen in einer Schicht l wird gesucht:

$$\frac{\partial E}{\partial b^l} = \begin{pmatrix} \frac{\partial E}{\partial b_1^l} \\ \frac{\partial E}{\partial b_2^l} \\ \vdots \\ \frac{\partial E}{\partial b_{n^l}^l} \end{pmatrix} \tag{18}$$

Mit der Kettenregel ergibt sich mit den Gleichungen 9 und 10:

$$\begin{aligned} \frac{\partial E}{\partial b_j^l} &= \frac{\partial E}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l} \\ &= \frac{\partial E}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l} \end{aligned} \tag{19}$$

Der Term $\frac{\partial E}{\partial a_j^l}$ ist bereits durch Gleichung 17 bekannt. Der Term $\frac{\partial a_j^l}{\partial z_j^l}$ ist ebenfalls bekannt aus Gleichung 8. Die einzige unbekannt partielle Ableitung ist $\frac{\partial z_j^l}{\partial b_j^l}$ und

wird durch das Ableiten von Gleichung 9 nach b_j^l gebildet:

$$\frac{\partial z_j^l}{\partial b_j^l} = 1 \tag{20}$$

Setzt man Gleichung 19 in den Vektor aus den gesuchten partiellen Ableitungen 18 ein, so ergibt sich für die Schicht l mit n^l Neuronen:

$$\begin{aligned} \frac{\partial E}{\partial b^l} &= \begin{pmatrix} \frac{\partial E}{\partial z_1^l} \\ \frac{\partial E}{\partial z_2^l} \\ \vdots \\ \frac{\partial E}{\partial z_{n^l}^l} \end{pmatrix} \\ &= \frac{\partial E}{\partial z^l} \end{aligned} \tag{21}$$

Weiterhin müssen die partiellen Ableitung nach den Gewichten gebildet werden. Gesucht wird hier also:

$$\frac{\partial E}{\partial w^l} = \begin{pmatrix} \frac{\partial E}{\partial w_{1,1}^l} & \frac{\partial E}{\partial w_{1,2}^l} & \cdots & \frac{\partial E}{\partial w_{1,n^{l-1}}^l} \\ \frac{\partial E}{\partial w_{2,1}^l} & \frac{\partial E}{\partial w_{2,2}^l} & \cdots & \frac{\partial E}{\partial w_{2,n^{l-1}}^l} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial w_{n^l,1}^l} & \frac{\partial E}{\partial w_{n^l,2}^l} & \cdots & \frac{\partial E}{\partial w_{n^l,n^{l-1}}^l} \end{pmatrix} \tag{22}$$

Mit der Kettenregel ergibt sich mit den Gleichungen 9 und 10:

$$\begin{aligned} \frac{\partial E}{\partial w_{j,k}^l} &= \frac{\partial E}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial w_{j,k}^l} \\ &= \frac{\partial E}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial w_{j,k}^l} \end{aligned} \tag{23}$$

Leitet man Gleichung 9 nach $w_{j,k}^l$ ab, so ergibt sich:

$$\frac{\partial z_j^l}{\partial w_{j,k}^l} = a_k^{l-1} \tag{24}$$

Setzt man diese Gleichungen in die gesuchte Matrix 22 ein, so ergibt sich:

$$\begin{pmatrix} \frac{\partial E}{\partial z_1^l} \cdot a_1^{l-1} & \frac{\partial E}{\partial z_1^l} \cdot a_2^{l-1} & \cdots & \frac{\partial E}{\partial z_1^l} \cdot a_{n^{l-1}}^{l-1} \\ \frac{\partial E}{\partial z_2^l} \cdot a_1^{l-1} & \frac{\partial E}{\partial z_2^l} \cdot a_2^{l-1} & \cdots & \frac{\partial E}{\partial z_2^l} \cdot a_{n^{l-1}}^{l-1} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial z_{n^l}^l} \cdot a_1^{l-1} & \frac{\partial E}{\partial z_{n^l}^l} \cdot a_2^{l-1} & \cdots & \frac{\partial E}{\partial z_{n^l}^l} \cdot a_{n^{l-1}}^{l-1} \end{pmatrix} \tag{25}$$

Diese neu entstandene Matrix kann schließlich in das dyadische Produkt der Vektoren $\frac{\partial E}{\partial z^l}$ und a^{l-1} umgeformt werden:

$$\begin{aligned} \frac{\partial E}{\partial w^l} &= \begin{pmatrix} \frac{\partial E}{\partial z_1^l} \\ \frac{\partial E}{\partial z_2^l} \\ \vdots \\ \frac{\partial E}{\partial z_{n^l}^l} \end{pmatrix} \otimes \begin{pmatrix} a_1^{l-1} \\ a_2^{l-1} \\ \vdots \\ a_{n^{l-1}}^{l-1} \end{pmatrix} \\ &= \frac{\partial E}{\partial z^l} \otimes a^{l-1} \end{aligned} \tag{26}$$

Mithilfe dieser Gleichungen kann für jede Schicht, für die $\frac{\partial E}{\partial a^l}$ und somit $\frac{\partial E}{\partial z^l}$ bekannt ist, die partielle Ableitung nach jedem Parameter gebildet werden. Bisher ist $\frac{\partial E}{\partial a^l}$ allerdings nur für die letzte Schicht a^L bekannt. Um die partiellen Ableitungen der Parameter in vorherigen Schichten zu finden, muss die Ableitung zur vorherigen Schicht $\frac{\partial E}{\partial a^{l-1}}$ gefunden werden. Gesucht wird also:

$$\frac{\partial E}{\partial a^{l-1}} = \begin{pmatrix} \frac{\partial E}{\partial a_1^{l-1}} \\ \frac{\partial E}{\partial a_2^{l-1}} \\ \vdots \\ \frac{\partial E}{\partial a_{n^{l-1}}^{l-1}} \end{pmatrix} \tag{27}$$

Leitet man Gleichung 9 nach a_k^{l-1} ab, so ergibt sich:

$$\frac{\partial z_j^l}{\partial a_k^{l-1}} = w_{j,k}^l \tag{28}$$

Da a^{l-1} von mehreren Neuronen aus der Schicht l als Eingabe verwendet wird ist ihre Ableitung zu a_k^{l-1} folgender Summe partieller Ableitungen:

$$\begin{aligned}\frac{\partial E}{\partial a_k^{l-1}} &= \sum_{j=1}^{n^l} \frac{\partial E}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial a_k^{l-1}} \\ &= \sum_{j=1}^{n^l} \frac{\partial E}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial a_k^{l-1}}\end{aligned} \tag{29}$$

Ersetzt man die partiellen Ableitungen in der Matrix 27 erhält man:

$$\begin{pmatrix} \sum_{j=1}^{n^l} \frac{\partial E}{\partial z_j^l} \cdot w_{j,1}^l \\ \sum_{j=1}^{n^l} \frac{\partial E}{\partial z_j^l} \cdot w_{j,2}^l \\ \vdots \\ \sum_{j=1}^{n^l} \frac{\partial E}{\partial z_j^l} \cdot w_{j,n^{l-1}}^l \end{pmatrix} = \begin{pmatrix} \frac{\partial E}{\partial z_1^l} \cdot w_{1,1}^l + \frac{\partial E}{\partial z_2^l} \cdot w_{2,1}^l + \cdots + \frac{\partial E}{\partial z_{n^l}^l} \cdot w_{n^l,1}^l \\ \frac{\partial E}{\partial z_1^l} \cdot w_{1,2}^l + \frac{\partial E}{\partial z_2^l} \cdot w_{2,2}^l + \cdots + \frac{\partial E}{\partial z_{n^l}^l} \cdot w_{n^l,2}^l \\ \vdots \\ \frac{\partial E}{\partial z_1^l} \cdot w_{1,n^{l-1}}^l + \frac{\partial E}{\partial z_2^l} \cdot w_{2,n^{l-1}}^l + \cdots + \frac{\partial E}{\partial z_{n^l}^l} \cdot w_{n^l,n^{l-1}}^l \end{pmatrix} \tag{30}$$

Diese kann in das elementweise Produkt einer Matrix und eines Vektors umgeformt werden:

$$\begin{pmatrix} w_{1,1}^l & w_{2,1}^l & \cdots & w_{n^l,1}^l \\ w_{1,2}^l & w_{2,2}^l & \cdots & w_{n^l,2}^l \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,n^{l-1}}^l & w_{2,n^{l-1}}^l & \cdots & w_{n^l,n^{l-1}}^l \end{pmatrix} \odot \begin{pmatrix} \frac{\partial E}{\partial z_1^l} \\ \frac{\partial E}{\partial z_2^l} \\ \vdots \\ \frac{\partial E}{\partial z_{n^l}^l} \end{pmatrix} \tag{31}$$

Die Matrix ist dabei gleich der Matrix der Gewichte der Schicht l, die transponiert wurden. Daher kann dies so zusammengefasst werden:

$$\frac{\partial E}{\partial a^{l-1}} = (w^l)^T \odot \frac{\partial E}{\partial z^l} \tag{32}$$

Schicht für Schicht von wird mit diesen Gleichungen also die partielle Ableitung $\frac{\partial E}{\partial a^l}$ gebildet und mithilfe dieser die partiellen Ableitungen nach den einzelnen Parametern gebildet. Schließlich werden die Gewichte und Verschiebungen nach dem Gradientenabstieg laut Gleichung 3 verändert:

$$w^l_{\text{neu}} = w^l_{\text{alt}} - \eta \frac{\partial E}{\partial w^l_{\text{alt}}} \tag{33}$$

$$b^l_{\text{neu}} = b^l_{\text{alt}} - \eta \frac{\partial E}{\partial b^l_{\text{alt}}} \tag{34}$$

4 Rekurrente Neuronale Netze

4.1 Funktionsweise von rekurrente Neuronale Netzen

Bei sequentiellen Daten kann es vorkommen, dass neuronale Netze Probleme haben diese zu approximieren. Ein solches Problem tritt beispielsweise auf, wenn eine Funktion aufgestellt werden soll, um folgende Sequenz zu komponieren:

a-c-a-d

Für diese Sequenz müsste gelten:

$$f(\text{"a"}) = \text{"c"}$$
$$f(\text{"c"}) = \text{"a"}$$
$$f(\text{"a"}) = \text{"d"}$$

Es ergibt allerdings keinen Sinn, dass $f(\text{“a“})$ gleichzeitig “c“ und “d“ ergibt. In diesem Fall ist eine Note nicht nur von ihrer vorherigen Note, sondern auch von anderen vorherigen Noten abhängig. Eine solche Abhängigkeit kann ein normales neuronales Netz nicht berücksichtigen. Ein häufig verwendeter Weg dieses Problem zu lösen ist durch ein sogenanntes rekurrentes neuronales Netz (RNN) (nach Karpathy (2018)). Dieses erhält ein Zwischenergebnis, in dem es alle benötigten Informationen aus vorherigen Kalkulationen speichern kann. Dieses Zwischenergebnis ist ein Vektor und heißt Hidden State h. Zu Beginn sind darin keine Informationen enthalten und er ist mit Nullen gefüllt. Das neuronale Netz wird nun in zwei neuronale Netze aufgeteilt. Das erste neuronale Netz g_1 hat als Eingabe das vorherige Zwischenergebnis h_{t-1} und die jetzige Note und errechnet daraus das neue Zwischenergebnis h_t. Die Parameter P der neuronalen Netze g_1

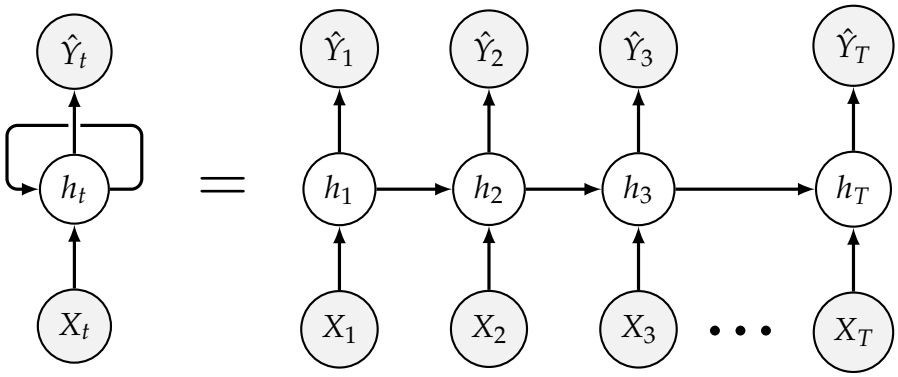
und g_2 werden in den folgenden Gleichungen nummeriert, um sie voneinander zu unterscheiden Das Zwischenergebnis errechnet sich also wie folgt:

$$h_t = g_1(X_t, h_{t-1}, P_1) \tag{35}$$

Die Variable t gibt dabei den Zeitpunkt an, zu dem das Zwischenergebnis ausgerechnet wird. Für die Note X_t an Stelle t wird also das Zwischenergebnis h_t gebildet. Diese Funktion verarbeitet die beiden Vektoren X_t und h_{t-1}. Ein neuronales Netz kann allerdings nur einen Vektor als Eingabe haben. Daher werden diese Vektoren zunächst zu einem gemeinsamen Vektor $X_t \frown h_{t-1}$ konkateniert. Das zweite Netz g_2 generiert aus dem neuen Zwischenergebnis das Gesamtergebnis für die Note:

$$\hat{Y}_t = g_2(h_t, P_2) \tag{36}$$

Dies kann grafisch so dargestellt werden:



4.2 Ableitung von rekurrenten neuronalen Netzen

Auch bei rekurrenten neuronalen Netzen muss für das Anwenden des Gradientenabstiegs die partielle Ableitung nach den einzelnen Parametern gebildet werden. Wie in Sektion 2.3 wird nach Gleichung 15 zu jedem Zeitpunkt die partielle Ableitung $\frac{\partial E_t}{\partial \hat{Y}_t}$ gebildet. Für das neuronale Netz g_2 gilt allgemein:

$$a_t^1 = h_t$$
$$\hat{Y}_t = a_t^L$$

Aus Sektion 2.3 sind die Gleichungen bekannt um den Gradientenabstieg für g_2 durchzuführen und die partielle Ableitung $\frac{\partial E_t}{\partial a_t^1} = \frac{\partial E_t}{\partial h_t}$ für g_2 zu bilden.

In Sektion 2.3 war $\frac{\partial E_t}{\partial a_t^1}$ gleich der Ableitung nach X_t, also ein konstanter Vektor. Bei einem RNN ist diese Schicht allerdings das Ergebnis von Netz g_1. Daher müssen, um die Ableitung zu vervollständigen, weitere partielle Ableitungen nach g_1 gebildet werden. Allgemein gilt für g_1:

$$a_t^1 = X_t \frown h_{t-1} \tag{37}$$

$$a_t^L = h_t \tag{38}$$

Die partielle Ableitung $\frac{\partial E_t}{\partial a_t^1} = \frac{\partial E_t}{\partial h_{t-1}}$ kann mit den in Sektion 2.3 beschriebenen Gleichungen gebildet werden. Mithilfe dieser partiellen Ableitung kann nun auch der Gradientenabstieg nach den Parametern von g_1 stattfinden. Da g_1 als Eingabe neben der Konstante X einen nicht konstanten Vektor h_{t-1} hat, ist die Ableitung noch nicht vollständig. Es muss daher noch die Ableitung nach h_{t-1} gebildet werden:

$$E_t{}' = \frac{\partial E_t}{\partial h_{t-1}} \cdot h_{t-1}{}' \tag{39}$$

Leitet man Gleichung 35 ab, so erhält man:

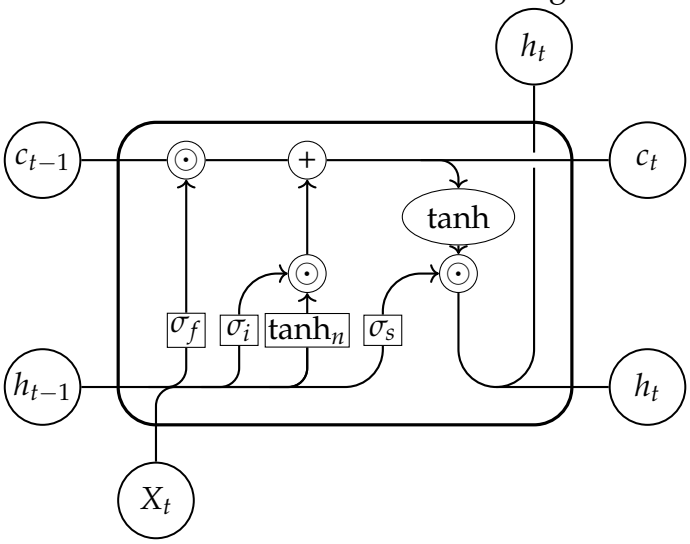
$$h_t{}' = \frac{\partial h_t}{\partial X_t} + \frac{\partial h_t}{\partial h_{t-1}} \cdot h_{t-1}{}' \tag{40}$$

Mithilfe von den Gleichungen aus Sektion 2.3 können die partiellen Ableitungen $\frac{\partial h_t}{\partial X_t}$ und $\frac{\partial h_t}{\partial h_{t-1}}$ gebildet werden. Um $h_{t-1}{}'$ zu errechnen, wird die gleiche Formel verwendet. Da t in dieser Formel eine Variable ist, so kann hier einfach t durch $t-1$ ersetzt werden. Dies wird so lange fortgeführt, bis man bei $t = 1$ angekommen ist. Der Hidden State h_{1-1} ist ein Vektor gefüllt mit Nullen und hat somit eine konstante Ableitung.

5 Long short-term memory Zelle

5.1 Funktionsweise von Long short-term memory Zelle

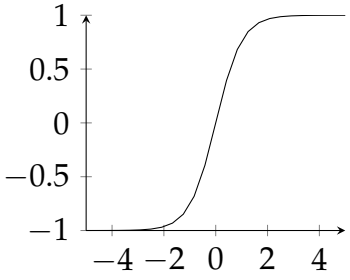
Ein Musikstück ist ein zusammenhängendes Werk. Ein Motiv, welches sich in der Einleitung findet, kann beispielsweise wieder im Schluss auftreten. Ein RNN kann Schwierigkeiten haben, solche Abhängigkeiten über einen längeren Zeitraum zu erfassen und Informationen länger zu behalten. Dies liegt daran, dass die in h_t gespeicherten Informationen, nach Gleichung 35 mit unterschiedlichen Werten wiederholt multipliziert werden. Dabei können wichtige Informationen mit der Zeit verloren gehen. Die Lösung für dieses Problems ist die Verwendung einer sogenannten Long short-term memory (LSTM) Zelle (nach Hochreiter und Schmidhuber (1997)). Diese ersetzt das neuronale Netz g_1 und hat selbst ein zusätzliches Zwischenergebnis, in dem Informationen langfristig gespeichert werden können. Solch eine Zelle kann so dargestellt werden:



Wie die Funktion g_1, hat eine LSTM Zelle die die Eingaben X_t und h_{t-1}. Zudem hat sie als Eingabe das zusätzliche Zwischenergebnis c_{t-1}. Als Ergebnis hat diese Funktion nicht nur den neuen Hidden State h_t, sondern auch den neuen Cell State c_t. Dieser wird nur für die LSTM Zelle verwendet und wird nicht weitergegeben. Wie in einem normalen RNN wird der Hidden State anschließend von Netz g_2 weiter verarbeitet, um zum Gesamtergebnis für eine Note zu gelangen. Die Zelle besteht aus mehreren kleinen neuronalen Netzen, sogenannten Gattern, mit je unterschiedlicher Funktion. Diese werden in den eckigen Kästen dargestellt. Jedes dieser Gatter hat die gleiche Eingabe, die g_1 hatte, also den vorherigen Hidden

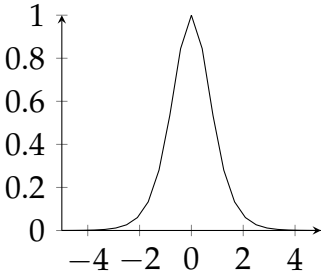
State konkateniert mit der jetzigen Note: $X_t \frown h_{t-1}$. Die Netze σ_f, σ_i und σ_n haben dabei als Aktivierungsfunktion die σ Funktion mit Gleichung 7. Daher liegen die Werte in ihrem Ergebnisvektor immer im Bereich $]0, 1[$. Multipliziert man also einen anderen Vektor mit solch einem Ergebnis elementweise, so wird dieser an jeder Stelle um einen bestimmten Faktor verkleinert. Diese Multiplikation kann als das Durchfließen einer Information durch ein Ventil interpretiert werden. Ein Vektor, der mit einem solchen Gatter σ_f multipliziert wird, ist der vorherige Cell State c_{t-1}. Das Ergebnis dieser Multiplikation ist Teil des neuen Cell State und erlaubt es der LSTM Zelle unwichtige Informationen aus dem alten Cell State zu vergessen. Welche Informationen relevant sind, entscheidet das Gatter σ_f. Sind alle Informationen wichtig, so ist sein Ergebnisvektor an jeder Stelle eins. Sind bestimmte Informationen unwichtig, so ist dieser Vektor an diesen Stellen null. Das LSTM soll außerdem neue Informationen in seinem Cell State speichern können. Dazu verarbeitet das neuronale Netz $\tanh_n$ die neuen Informationen. Im Gegensatz zu den anderen neuronalen Netzen hat das Netz $\tanh_n$ die Aktivierungsfunktion $\tanh(z)$. Diese hat folgenden Verlauf und Gleichung:

$$\tanh(z) = \frac{2}{1 + e^{-2z}} - 1 \tag{41}$$



Ihre Ableitung erfüllt auch alle Bedingungen für den Gradientenabstieg, die die Funktion σ erfüllt und sieht so aus:

$$\tanh'(z) = 1 - \tanh^2(z) \tag{42}$$



Sie wird verwendet, da sie auch negative Werte annehmen kann, wodurch der Cell State ebenfalls negativ sein kann und somit mehr Informationen speichern kann. Welche dieser verarbeiteten Informationen wichtig sind und weiter geleitet werden, entscheidet das Gatter σ_i, mit dem diese multipliziert werden. Dieses Ergebnis wird nun zum alten Cell State addiert und bildet somit den neuen Cell State. Die Gleichung für den Cell State sieht also wie folgt aus:

$$c_t = c_{t-1} \cdot \sigma_f(X_t, h_{t-1}, P_f) + \sigma_i(X_t, h_{t-1}, P_i) \cdot \tanh_n(X_t, h_{t-1}, P_n) \tag{43}$$

Nun soll die Zelle aus dem Cell State das Zwischenergebnis h_t errechnen. Dazu wird der neue Cell State verwendet, auf den zunächst die Funktion tanh angewendet wird. Dies liegt daran, dass der Cell State durch beliebig viele Additionen jeden beliebigen Wert annehmen kann. Ein hoher Wert als Eingabe für ein neuronales Netz kann allerdings Probleme verursachen. Auf diese soll in Sektion 6.1 genauer eingegangen werden. Bevor dieses Ergebnis als neuer Hidden State verwendet werden kann, muss sichergestellt werden, dass nur die wichtigen Informationen aus dem Cell State weitergeleitet werden. Dazu wird das Ergebnis mit dem Gatter σ_s multipliziert. Für den Hidden State ergibt sich dadurch die Gleichung:

$$h_t = \tanh(c_t) \cdot \sigma_s(X_t, h_{t-1}, P_s) \tag{44}$$

Dieser Hidden State kann dann wie in Sektion 4.1 von g_2 weiter verarbeitet werden, um zum Endergebnis zu gelangen.

5.2 Ableitung von long short-term memory Zellen

Jedes der Gatter hat seine eigenen Parameter, die eingestellt werden müssen, damit es weiß, wie es Informationen verarbeiten soll. Daher müssen für die LSTM Zellen die partiellen Ableitungen nach den Parametern der einzelnen neuronalen Netze gebildet werden. Da eine LSTM Zelle g_2 lediglich ersetzt, werden für den Gradientenabstieg zunächst die Ableitungen wie in Sektion 4.2 gebildet. Ist dies abgeschlossen, so müssen nun ausgehend von der partiellen Ableitung $\frac{\partial E}{\partial h_t}$ die partiellen Ableitungen zu den einzelnen Parametern gefunden werden. Da eine LSTM Zelle zwei, nicht konstante, Eingaben und zwei, nicht konstante, Ergebnis-

se hat, müssen dazu folgende vier partiellen Ableitungen gebildet werden: $\frac{\partial h_t}{\partial h_{t-1}}$, $\frac{\partial h_t}{\partial c_{t-1}}$, $\frac{\partial c_t}{\partial h_{t-1}}$ und $\frac{\partial c_t}{\partial c_{t-1}}$ Die partielle Ableitung von Gleichung 44 nach h_{t-1} ist nach der Produktregel:

$$\frac{\partial h_t}{\partial h_{t-1}} = \tanh'(c_t) \cdot \frac{\partial c_t}{\partial h_{t-1}} \cdot \sigma_s(X_t, h_{t-1}, P_s) + \tanh(c_t) \cdot \frac{\partial \sigma_s(X_t, h_{t-1}, P_s)}{\partial h_{t-1}} \tag{45}$$

Leitet man Gleichung 43 nach h_{t-1} ab, so ergibt sich:

$$\begin{aligned} \frac{\partial c_t}{\partial h_{t-1}} &= c_{t-1} \cdot \frac{\partial \sigma_f(X_t, h_{t-1}, P_f)}{\partial h_{t-1}} \\ &+ \frac{\partial \sigma_i(X_t, h_{t-1}, P_i)}{\partial h_{t-1}} \cdot \tanh_n(X_t, h_{t-1}, P_n) \\ &+ \sigma_i(X_t, h_{t-1}, P_i) \cdot \frac{\partial \tanh_n(X_t, h_{t-1}, P_n)}{\partial h_{t-1}} \end{aligned} \tag{46}$$

Mit Gleichung 44 wird die folgende partielle Ableitung gebildet:

$$\frac{\partial h_t}{\partial c_{t-1}} = \tanh'(c_t) \cdot \frac{\partial c_t}{\partial c_{t-1}} \cdot \sigma_s(X_t, h_{t-1}, P_s) \tag{47}$$

Ableitung von Gleichung 43 nach c_{t-1} liefert außerdem:

$$\frac{\partial c_t}{\partial c_{t-1}} = \sigma_f(X_t, h_{t-1}, P_f) \tag{48}$$

Die partiellen Ableitungen zu den vorherigen Hidden und Cell State sind dann nach der Kettenregel:

$$\begin{aligned} \frac{\partial E}{\partial c_{t-1}} &= \frac{\partial E}{\partial c_t} \cdot \frac{\partial c_t}{\partial c_{t-1}} + \frac{\partial E}{\partial h_t} \cdot \frac{\partial h_t}{\partial c_{t-1}} \\ \frac{\partial E}{\partial h_{t-1}} &= \frac{\partial E}{\partial c_t} \cdot \frac{\partial c_t}{\partial h_{t-1}} + \frac{\partial E}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_{t-1}} \end{aligned} \tag{49}$$

Dabei ist $\frac{\partial E}{\partial h_t}$ durch die Gleichungen in Sektion 4.2 bekannt und $\frac{\partial E}{\partial c_t}$ lässt sich nach Gleichung 44 wie folgt bilden:

$$\frac{\partial E}{\partial c_t} = \frac{\partial E}{\partial h_t} \cdot \tanh'(c_t) \cdot \sigma_s(X_t, h_{t-1}, P_s) \tag{50}$$

Mithilfe dieser Gleichungen kann die partielle Ableitung zu jedem Hidden State h_t und Cell State c_t gebildet werden. Es werden nun noch die partiellen Ableitungen der Netze: $\frac{\partial \tanh_n(X_t,h_{t-1},P_n)}{\partial h_{t-1}}$ $\frac{\partial \sigma_f(X_t,h_{t-1},P_f)}{\partial h_{t-1}}$, $\frac{\partial \sigma_s(X_t,h_{t-1},P_s)}{\partial h_{t-1}}$ und $\frac{\partial \sigma_i(X_t,h_{t-1},P_i)}{\partial h_{t-1}}$ benötigt. Diese lassen sich nach den Gleichungen aus Sektion 2.3 bilden. Um mithilfe der partiellen Ableitungen $\frac{\partial E}{\partial c_t}$ und $\frac{\partial E}{\partial h_t}$ die partiellen Ableitungen zu den Parametern der Gatter zu finden, muss zunächst die partielle Ableitung zu dem Ergebnis jedes Gatters a^L gefunden werden. Leitet man die Funktionen 44 und 43 ab und fasst diese zusammen, so erhält man so erhält man:

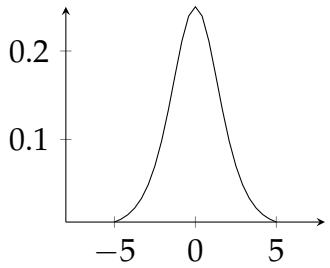
$$\begin{aligned}
\frac{\partial E}{\partial \tanh_n} &= (\frac{\partial E}{\partial c_t} + \frac{\partial E}{\partial h_t} \cdot \sigma_s(X_t, h_{t-1}, P_s) \cdot \tanh'(c_t)) \cdot \sigma_i(X_t, h_{t-1}, P_i) \\
\frac{\partial E}{\partial \sigma_i} &= (\frac{\partial E}{\partial c_t} + \frac{\partial E}{\partial h_t} \cdot \sigma_s(X_t, h_{t-1}, P_s) \cdot \tanh'(c_t)) \cdot \tanh_n(X_t, h_{t-1}, P_n) \\
\frac{\partial E}{\partial \sigma_f} &= (\frac{\partial E}{\partial c_t} + \frac{\partial E}{\partial h_t} \cdot \sigma_s(X_t, h_{t-1}, P_s) \cdot \tanh'(c_t)) \cdot c_{t-1} \\
\frac{\partial E}{\partial \sigma_s} &= \frac{\partial E}{\partial h_t} \cdot \tanh(c_t)
\end{aligned} \tag{51}$$

Der Gradientenabstieg wird nun mit den Gleichungen 51 wie in Sektion 2.3 für jeden Zeitpunkt durchgeführt. Dafür wie in Sektion 4.2 rekursiv h_{t-1} und c_{t-1} gebildet und damit Gleichung 51 angewandt, bis $t = 1$ erreicht wurde. Der Hidden State h_{1-1} und der Cell State c_{1-1} sind dabei Vektoren gefüllt mit Nullen und haben somit eine konstante Ableitung.

6 Probleme beim Trainieren von neuronalen Netzen

6.1 Verschwindende Gradienten

Neuronale Netze lernen, indem sie die partielle Ableitungen nach ihrem Parametern bilden und diese dann davon abziehen. Dabei wird nach den Gleichungen aus 3.2 die partiellen Ableitungen unter anderem mit σ' multipliziert. Diese hat die Gleichung $\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$ und den folgenden Graphen:



Dieser Graph ist bei sehr positiven oder sehr negativen Werten von z fast null. Daraus folgt, dass die partiellen Ableitungen ebenfalls fast null sind und so das neuronale Netz mit Gleichung 3 gar nicht oder nur noch extrem langsam lernen kann. Dies nennt man auch das Problem der verschwindenden Gradienten. Ein Weg dieses Problem zu lösen ist die Werte von z kleinzuhalten. Dafür kann zwischen den Schichten von neuronalen Netzen eine weiter sogenannte Batch Normalization Schicht (nach Ioffe und Szegedy (2015)) hinzugefügt. Für einen Eingabevektor der Größe m gelten dafür folgende Gleichungen:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} X_i \tag{52}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m} (X_i - \mu_B)^2 \tag{53}$$

$$\hat{x}_i = \frac{X_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{54}$$

$$Y_i = \gamma \hat{x}_i + \beta \tag{55}$$

In diesen Gleichungen ist X ein Vektor, der die Ergebnisse der letzten Schicht enthält und Y der Vektor, der als Eingabe für die nächste Schicht dient. Die Variablen μ_B, σ_B^2 und $\hat{x}_i$ sind Zwischenergebnisse, die die Gleichung vereinfachen. Zunächst wird dazu μ_B gebildet. Dies ist der Mittelwert des Vektors X. Anschließend wird die mittlere quadratische Abweichung σ_B^2 von diesem Mittelwert gebildet. Um ein normalisierter Vektor zu bilden, wird der Mittelwert vom Startvektor abgezogen und durch die Wurzel aus der mittleren quadratischen Abweichung dividiert. Der Term ϵ ist dabei ein sehr kleiner Wert, der eine Division durch null verhindert. Im letzten Schritt wird der neu entstandene Vektor mit einem Faktor γ multipliziert und um β verschoben. Die beiden Parameter γ und β müssen durch den Gradientenabstieg gelernt werden. Dafür werden die folgenden parti-

ellen Ableitungen verwendet:

$$\frac{\partial E}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial E}{\partial Y_i} \tag{56}$$

$$\frac{\partial E}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial E}{\partial Y_i} \cdot \hat{x}_i \tag{57}$$

Die Ableitung $\frac{\partial E}{\partial Y_i}$ ist dabei durch die Gleichungen aus Sektion 3.2 bekannt. Um dieses Verfahren weiterzuführen, muss die Ableitung durch dies Batch Normalization Schicht gebildet werden:

$$\frac{\partial E}{\partial \hat{x}_i} = \frac{\partial E}{\partial Y_i} \cdot \gamma \tag{58}$$

$$\frac{\partial E}{\partial \sigma_B^2} = \sum_{i=1}^{m} \frac{\partial E}{\partial \hat{x}_i} \cdot (X_i - \mu_B) \cdot -\frac{1}{2} \cdot (\sigma_B^2 + \epsilon)^{-\frac{3}{2}} \tag{59}$$

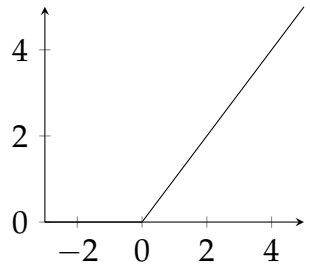
$$\frac{\partial E}{\partial \mu_B} = \sum_{i=1}^{m} \frac{\partial E}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \beta}} + \frac{\partial E}{\partial \sigma_B^2} \cdot \frac{1}{m} \sum_{i=1}^{m} 2(X_i - \mu_B) \tag{60}$$

$$\frac{\partial E}{\partial X_i} = \frac{\partial E}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \beta}} + \frac{\partial E}{\partial \mu_B} \cdot \frac{1}{m} + \frac{\partial E}{\partial \sigma_B^2} \cdot \frac{2}{m}(X_i - \mu_B) \tag{61}$$

Die entstandene partielle Ableitung $\frac{\partial E}{\partial X_i}$ ist dabei Gleich der Ableitung $\frac{\partial E}{\partial a_i^l}$ für die Schicht, deren Ergebnisse normalisiert werden. Diese partielle Ableitung kann nun verwendet werden um die partiellen Ableitungen der vorherigen Schichten zu bilden. Ein weiterer Weg das Problem der verschwindenden Gradienten zu lösen ist ein durch das Verwenden einer alternativen Funktion für σ. Eine Möglichkeit dazu ist die Rectified Linear Unit (ReLU) Funktion:

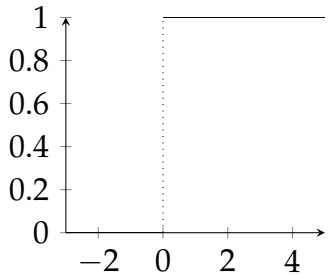
$$\text{ReLU}(z) = \begin{cases} z & \text{wenn } z > 0 \\ 0 & \text{wenn } z \leq 0 \end{cases} \tag{62}$$

Der dazugehörige Graph sieht so aus:



Wie die σ Funktion ist diese bei sehr negativen Werten gleich null. Im Intervall $]0, \infty[$ ist die ReLU Funktion linear, sie ist daher bei sehr hohen Werten ebenfalls hoch. Der Vorteil gegenüber der σ Funktion liegt in ihrer Ableitung:

$$\text{ReLU}'(z) = \begin{cases} 1 & \text{wenn } z > 0 \\ 0 & \text{wenn } z \leq 0 \end{cases} \tag{63}$$



Diese ist auch bei sehr hohen Werten nicht null und der Gradientenabstieg kann weiterhin funktionieren. Ein weiterer Ersatz für σ, der häufig in der letzten Schicht verwendet wird, ist die Softmax Funktion (nach Kurbiel (2021)). Für n^L Neuronen in der letzten Schicht hat sie folgende Gleichung:

$$s(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n^L} e^{z_j}} \tag{64}$$

Sie berücksichtigt nicht nur einen einzelnen Wert von z, sondern alle z Werte einer Schicht gleichzeitig. Sie bildet für alle z Werte e^z, summierte diese und dividiert jeden einzelnen Wert durch diese Summe. Ihre Ableitung sieht so aus:

$$\frac{\partial s_i}{\partial z_j} = s_i \cdot (1\{i = j\} - s_j) \tag{65}$$

Verwendet man die Softmax Funktion zusammen mit der sogenannten Kreuzentropie Kostenfunktion:

$$E = -\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i) \tag{66}$$

mit der Ableitung:

$$\frac{\partial L}{\partial \hat{y}} = -\sum_{i=1}^{n} y_i \cdot \frac{1}{\hat{y}_i} \tag{67}$$

so gilt für die letzte Schicht:

$$\frac{\partial E}{\partial z^L} = s - Y \tag{68}$$

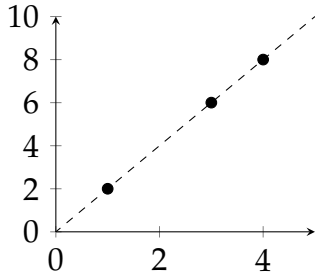
Unabhängig davon, wie hoch der Wert von z ist, tritt somit in der letzten Schicht kein Problem der verschwindenden Gradienten auf.

6.2 Overfitting

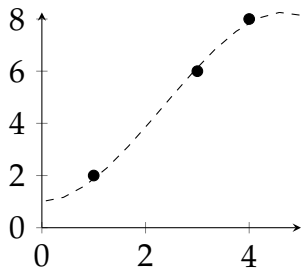
Ein neuronales Netz erhält eine Menge von Daten und versucht eine Funktion zu approximieren, auf der diese Daten liegen. Dabei kann es vorkommen, dass das neuronale Netz nicht das Muster hinter den Daten findet, sondern sich diese lediglich merkt. Soll ein neuronales Netz beispielsweise die Funktion $f(x) = 2x$ lernen, so gibt man ihm unterschiedliche Paare von x und y. Diese könnten beispielsweise so aussehen:

x	f(x) = y
1	2
3	6
4	8

Der Graph eines perfekten neuronalen Netzes, das die Beziehung zwischen x und y versteht, würde wie folgt aussehen:



Es kann allerdings vorkommen, dass der Graph des neuronalen Netzes so aussieht:



Das neuronale Netz hat sich die einzelnen Punkte gemerkt, hat aber unnötige Komplexität und das System hinter den Daten nicht verstanden. Das nennt sich Overfitting. Ein Weg dies zu verhindern ist mehr Daten zum Trainieren zu verwenden. So wird es schwerere für das neuronale Netz eine solche falsche Approximation zu erzeugen. Eine andere Möglichkeit ist es, die Anzahl der Parameter und somit die Komplexität des neuronalen Netzes zu verkleinern. Dadurch ist es gezwungen Muster zu finden und kann sich nicht alle Trainingsdaten merken und muss die Struktur der Daten erkennen. Ein zu kleines Netzwerk bedeutet allerdings auch eine schlechtere Approximation. Es muss also die richtige Größe und Komplexität des Netzwerkes für jedes Problem und die Menge an Daten gefunden werden. Um die Komplexität des Netzes zu reduzieren, kann außerdem eine Technik mit dem Namen Dropout verwendet werden (nach Brownlee (2018)). Diese deaktiviert während des Lernens zufällig wenige Neuronen einer Schicht. Das bedeutet, dass das Netzwerk mit weniger Neuronen eine Approximation finden muss und es schwerer wird sich die einzelnen Daten zu merken. Ist das Lernen abgeschlossen, so werden wieder alle Neuronen aktiviert, um die beste Approximation zu finden.

6.3 Beschleunigen von Gradientenabstieg

Beim Gradientenabstieg wird für das gesamte Netz eine globale Lernrate η festgelegt, nach der für jeden Parameter der Gradientenabstieg durchgeführt wird. Es kann dabei vorkommen, dass einige Parameter eine große Änderung benötigen, andere aber nur eine kleine. Dies kann zu dem langsamen Lernen von manchen Parametern führen. Ein Weg dieses Problem zu lösen ist durch ein Verfahren mit dem Namen Adam (nach Kingma und Ba (2015)). Dabei wird jedem Parameter eine eigene Lernrate, abhängig von seinen vergangenen partiellen Ableitungen $\frac{\partial E_t}{\partial p}$, gegeben. Für diese Verfahren werden zunächst folgende Zwischenergeb-

nisse gebildet:

$$v_{neu_{\delta p}} = \beta_1 \cdot v_{alt_{\delta p}} + (1 - \beta_1) \cdot \frac{\partial E}{\partial p} \tag{69}$$

$$s_{neu_{\delta p}} = \beta_2 \cdot s_{alt_{\delta p}} + (1 - \beta_2) \cdot (\frac{\partial E}{\partial p})^2 \tag{70}$$

$$\tag{71}$$

Anschließend wird die Formel 3, die den Gradientenabstieg beschreibt, durch folgende Formel ersetzt:

$$p_{\text{neu}_i} = p_{\text{alt}_i} - \frac{v_{neu_{\delta p}}}{\sqrt{s_{neu_{\delta P} + \epsilon}}} \cdot \eta \tag{72}$$

Dabei sind β_1 und β_2 Hyperparameter, die experimentell ermittelt werden müssen. Sie haben einen Wert zwischen null und eins und bestimmen, wie stark diese neue Formeln angewandt werden. Sind beide Parameter null, so wird der normale Gradientenabstieg angewandt. Der Hyperparameter ϵ ist dabei ein extrem kleiner Wert, der die Division durch null verhindert. Diese Formel funktioniert nach zwei Prinzipien. Der erste Teil der Formel $v_{neu_{\delta P}}$ kann als Geschwindigkeit interpretiert werden. Lernt ein Parameter sehr langsam, so kann dieser mit der Zeit Geschwindigkeit aufbauen und somit schneller lernen. Der Hyperparameter β_1 ist dabei Reibung, die die Geschwindigkeit mit der Zeit reduziert. Die Geschwindigkeit wird in jedem Schritt um die partielle $\frac{\partial E}{\partial p}$ Ableitung erhöht. Der zweite Teil $s_{neu_{\delta p}}$ hat dabei den Namen Root Mean Squared Propagation. Dieser summiert die früheren quadratischen Gradienten und dividiert dann durch diese. Dadurch werden Gradienten, die über einen längeren Zeitraum zu groß sind, kleiner und langfristig kleine Gradienten vergrößert. Auch hier wird diese Summe in jedem Schritt um einen Faktor β_2 verringert. Diese Modifikation des Gradientenabstiegs kann das Lernen von neuronalen Netzen signifikant beschleunigen.

7 Implementierung

7.1 Implementierung von RNN mit LSTM Zellen

Die gesamte Implementierung orientiert sich an der Implementierung von Skúli (2017). Dabei wurden einige Modifikationen und Verbesserungen an diesem Programm gemacht. Zur Implementierung der, in dieser Arbeit erläuterten Techniken und Gleichungen, wird die Programmiersprache Python mit der Bibliothek Keras verwendet. Diese erleichtert die Implementierung von neuronalen Netzen. Zum Trainieren soll das neuronale Netz 100 Noten der Reihe nach als Eingabe bekommen und anschließend die Note, die darauf folgt, vorhersagen. Dabei enthält die Variable network_input diese 100 Noten und die Variable network_output die folgende Note. Hierfür wird die Klasse Sequential verwendet. Dieser werden bei der Initialisierung alle benötigten Informationen zum Aufbau des neuronalen Netzes gegeben. Dies sieht dann so aus:

```
model = Sequential()

model.add(LSTM(
    512,
    input_shape=(network_input.shape[1], network_input.shape[2]),
    recurrent_dropout=0.3,
    return_sequences=True
))
model.add(LSTM(1024, return_sequences=True,
 ↪  recurrent_dropout=0.3,))
model.add(LSTM(1024))
model.add(BatchNorm())
model.add(Dropout(0.3))
model.add(Dense(512))
model.add(Activation('relu'))
model.add(BatchNorm())
model.add(Dropout(0.3))
model.add(Dense(n_vocab))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam')
```

In der ersten Zeile wird also eine Instanz der Klasse Sequential erstellt, die al-

les verwaltet. Anschließend werden mit der Funktion model.add() unterschiedliche Operationen hinzugefügt. Dabei wird Sequential als großes neuronales Netz mit unterschiedlichen Schichten interpretiert. Als erste Schichten hat es mehrere LSTM Schichten, die aneinandergereiht werden. Das Ergebnis des einen LSTM ist dabei die Eingabe für das nächste LSTM. Um dem Programm diese Aneinanderreihung zu signalisieren, wird der Parameter return_sequences auf True gesetzt. Das letzte LSTM besitzt diesen Parameter nicht, es gibt daher als Ergebnis nur einen Vektor zurück. In den Klammern von LSTM steht an erster Stelle zunächst, wie viele Neuronen diese Schicht haben soll. Die Anzahl der Schichten und Neuronen je Schicht wurde für dieses Problem experimentell ermittelt. Bei der ersten LSTM Schicht muss außerdem input_shape, die Größe des Eingabevektors, angegeben werden. Dieser ist gleich der Größe der Eingabevariable network_input. In den darauf folgenden Schichten ist die Größe des Eingabevektors die Größe des Ausgabevektors der vorherigen Schicht und muss deshalb nicht angegeben werden. Anschließend wird noch eine Batch Norm Schicht aus Sektion 6.1 eingefügt, um verschwindende Gradienten zu verhindern. Danach folgen Schichten von neuronalen Netzen aus Sektion 3.1, die hier Dense heißen. Diese haben die Aktivierungsfunktionen ReLU und Softmax. Zwischen den Schichten befinden sich Dropout Schichten, die in Sektion 6.2 erläutert wurden, um Overfitting zu verhindern. Die Zahl in den Klammern gibt dabei an, wie viel Prozent der Neuronen in der Schicht temporär deaktiviert werden sollen. Auch in den LSTM Schichten wird durch den Parameter recurrent_dropout angegeben, dass Dropout verwendet werden soll. Die Größe der letzten Schicht ist n_vocab. Diese Variable ist gleich der Größe, den der Ergebnisvektor haben soll. Abschließend wird das neuronale Netz kompiliert und ein Verfahren zum Beschleunigen der Gradienten, Adam aus Sektion 6.3, ausgewählt. Der Parameter loss bestimmt, dass die Kostenfunktion Kreuzentropie aus Sektion 6.1 verwendet werden soll. Nun kann das Netzwerk mit den Trainingsdaten network_input und network_output trainiert werden.

```
model.fit(network_input, network_output, epochs=1000)
```

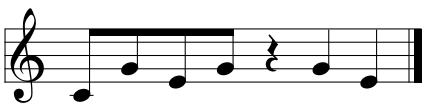
Der Parameter epochs gibt hier vor, dass das Netzwerk 1000-mal an allen Daten trainiert werden soll.

7.2 Umwandeln von Musikstücken in Vektoren

Bevor das neuronale Netz Musikstücke verarbeiten kann, müssen diese erst in eine kompatible Form umgewandelt werden. Dazu wird folgende Python Funktion verwendet:

```
notes = []

for file in glob.glob("musik\*.mid"):
    midi = music21.converter.parse(file)
    notes_to_parse = midi.flat.notes

    for element in notes_to_parse:
        length = str(element.quarterLength)

        if isinstance(element, note.Note):
            notes.append(str(element.pitch) + " " + length)
        elif isinstance(element, chord.Chord):
            notes.append('.'.join(str(n) for n in
            ↪  element.normalOrder) + " " + length)
        elif isinstance(element, note.Rest):
            notes.append(str(element.name) + " " + length)
```

Diese geht durch einen Ordner, in dem alle Musikstücke im MIDI Format gespeichert sind. Jedes Stück in diesem Ordner wird mithilfe der music21 Bibliothek in einzelne Elemente unterteilt. Jedes dieser Elemente ist entweder eine Note, ein Akkord oder eine Pause. Eine Schleife geht durch diese und wandelt sie in eine Zeichenkette um. Diese Zeichenkette besteht aus der Tonhöhe oder dem Namen des Akkordes und der Länge. Die Noten:



sehen beispielsweise nach Anwenden dieser Funktion so aus:

```
['C4 0.5', 'G4 0.5', 'E4 0.5', 'C4 0.5', 'rest 1.0', 'G4 1.0', 'E4
↪  1.0']
```

Alle Stücke werden nun in dieses Format gebracht und zu einem großen Stück zusammen gefügt. Dieses wird wie folgt weiter verarbeitet:

```
pitchnames = sorted(set(item for item in notes))

note_to_int = dict((note, number) for number, note in
↪  enumerate(pitchnames))

network_input = []
network_output = []

for i in range(0, len(notes) - sequence_length, 1):
    network_input.append([note_to_int[char] for char in notes[i:i +
    ↪  sequence_length]])
    network_output.append(note_to_int[notes[i + sequence_length]])


network_input = network_input / float(n_vocab)
```

Dafür werden zunächst alle einzigartigen Elemente, die vorkommen, in der Variable pitchnames gespeichert. Anschließend wird das Dictionary, note_to_int erzeugt, das jedem dieser Elemente einen Index zuordnet. Das neuronale Netz soll als Eingabe 100 Noten nehmen und anhand dieser versuchen, die nächste Note hervorzusagen. Daher geht eine Schleife durch alle Noten und fügt der Liste network_input eine Liste mit 100 Noten hinzu und der Liste network_output die Note, die darauf folgen soll. Bevor diese Noten hinzugefügt werden, müssen diese mithilfe des zuvor erstellten Dictionary in Zahlen umgewandelt. Die Elemente von network_input können sehr hohe Werte annehmen. Wie in Sektion 6.1 erläutert können hohe Werte allerdings Probleme beim Lernen verursachen. Daher werden alle Elemente von network_input durch die Anzahl der individuellen Elemente n_vocab dividiert. Somit liegt die Eingabe immer zwischen null und eins. Zum Schluss werden die Eingaben und Ausgaben in Vektoren umgewandelt:

```
network_input = numpy.reshape(network_input, (len(network_input),
↪  sequence_length, 1))

network_output =
↪  keras.utils.np_utils.to_categorical(network_output)
```

Jede Note aus network_input in einen Vektor mit nur einem Element umgewandelt. Um die Ergebnisvektoren leichter zu erlernen, wird network_output

mit der Funktion keras.utils.np_utils.to_categorical() in ein sogenanntes One-Hot-Encoding umgewandelt. Jedem Element wird dabei ein Vektor der Größe n zugewiesen, der mit Nullen gefüllt ist. Am Index des Elementes wird die null durch eine eins ersetzt. Hat ein Element beispielsweise den Index drei, so sieht sein Vektor so aus:

$$[0, 0, 1, \ldots, 0]$$

7.3 Komponieren von neuartigen Musikstücken mit neuronalen Netzen

In den vorherigen Kapiteln wurde die hypothetische Funktion $f(X) = Y$ aufgestellt, die ein bereits existierendes Stück Note für Note erneut komponieren kann. Weiterhin wurde die Funktion $g(X, P) = \hat{Y}$ aufgestellt, die diese Funktion beliebig genau approximieren kann. Dabei ist allerdings noch keine neue Musik entstanden. Neue Musik kann entstehen, wenn das trainierte neuronale Netz, das mehrere bereits existierende Stücke approximieren kann, den Anfang eines unbekannten Stückes weiter komponieren soll. Die Hoffnung besteht hierbei darin, dass durch das Approximieren bereits existierender Musik so viel über Musik gelernt wurde, dass dieses theoretische Wissen auch auf ein unbekanntes Stück angewandt werden kann.

7.4 Programm zum Komponieren von Musik

Um neue Musik zu komponieren wird zunächst folgende Funktion verwendet:

```
int_to_note = dict((number, note) for number, note in
↪  enumerate(pitchnames))

pattern = network_input[numpy.random.randint(0,
↪  len(network_input)-1)]

prediction_output = []

for note_index in range(500):
    prediction_input = numpy.reshape(pattern, (1, len(pattern), 1))
    prediction_input = prediction_input / float(n_vocab)
```

```
        prediction = model.predict(prediction_input, verbose=0)

        index = numpy.argmax(prediction)
        result = int_to_note[index]
        prediction_output.append(result)

        pattern.append(index)
        pattern = pattern[1:len(pattern)]
```

Dafür wird das Dictionary int_to_note erstellt. Dieses nimmt einen Index eines Elements und wandelt diesen wieder in eine Note, in Form einer Zeichenkette, um. Anschließend wird zufällig aus unbekannten Noten eine Sequenz von 100 Noten ausgewählt und in der Variable pattern gespeichert. Diese werden in eine Form gebracht, die das neuronale Netz verarbeiten kann. Ist dies abgeschlossen, so werden dem Netz diese 100 Noten gezeigt und die Note, die darauf folgt vorhergesagt. Der Index, den das neuronale Netz vorhergesagt hat, wird in der Variable index gespeichert und nach Umwandlung in den Namen des Elements der Liste prediction_output hinzugefügt. Zuletzt wird die neu generierte Note der Startsequenz hinzugefügt und die erste Note entfernt, damit die Länge der Startsequenz bei 100 bleibt. Diese Schritte werden 500-mal wiederholt, und es entsteht ein neues Stück aus insgesamt 600 Elementen. Damit dieses Stück angehört werden kann, muss die Liste aus den Namen der Elemente noch in eine .mid Datei mithilfe der music21 Bibliothek und folgender Funktion umgewandelt werden:

```
offset = 0
output_notes = []

for pattern in prediction_output:
    pattern = pattern.split()
    duration = pattern[1]
    pattern = pattern[0]
    # pattern is a chord
    if ('.' in pattern) or pattern.isdigit():
        notes_in_chord = pattern.split('.')
        notes = []
        for current_note in notes_in_chord:
            new_note = note.Note(int(current_note))
```

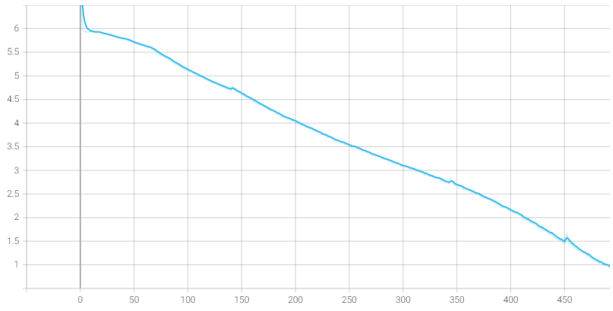
```
                new_note.storedInstrument = instrument.Piano()
                notes.append(new_note)
            new_chord = chord.Chord(notes)
            new_chord.offset = offset
            output_notes.append(new_chord)
        # pattern is a rest
        elif 'rest' in pattern:
            new_rest = note.Rest(pattern)
            new_rest.offset = offset
            new_rest.storedInstrument = instrument.Piano()
            output_notes.append(new_rest)
        # pattern is a note
        else:
            new_note = note.Note(pattern)
            new_note.offset = offset
            new_note.storedInstrument = instrument.Piano()
            output_notes.append(new_note)

        offset += convert_to_float(duration)

    midi_stream = stream.Stream(output_notes)

    midi_stream.write('midi', fp='test_output.mid')
```

Zunächst geht eine Schleife durch alle Elemente, die das neuronale Netz vorhergesagt hat. Ein einzelnes Element wird nun in zwei Teile aufgeteilt. Zum einen den Namen des Elementes, der in der Variable pattern gespeichert wird und die Dauer des Elementes, die in der Variable duration gespeichert wird. Anschließend wird eine Unterscheidung gemacht. Handelt es sich bei dem Element um einen Akkord, so wird dieser in seine einzelnen Noten unterteilt, zu einem Akkordobjekt zusammengefügt und zur Liste output_notes hinzugefügt. Handelt es sich beim Element um eine Pause, so wird ein Pausenobjekt der Liste hinzugefügt. Ist das Element eine Note, so wird ein Notenobjekt der Liste hinzugefügt. Der Zeitpunkt, zu der diese Note gespielt werden soll, wird dabei auf die Variable offset gesetzt. Diese ist zu Beginn des Stückes auf null und wird nach jedem neu hinzugefügtem Element um die Dauer dieses Elementes erhöht. Zum Schluss kann aus dieser Liste aus Objekten eine .mid Datei erzeugt und gespeichert wer-

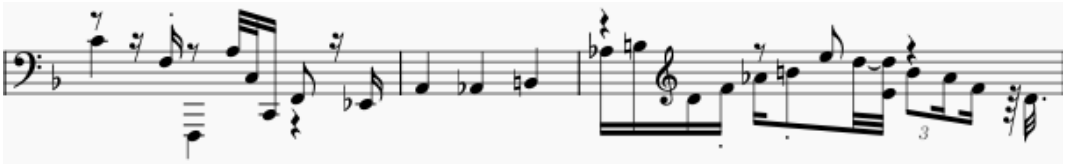
den.

8 Ergebnisse

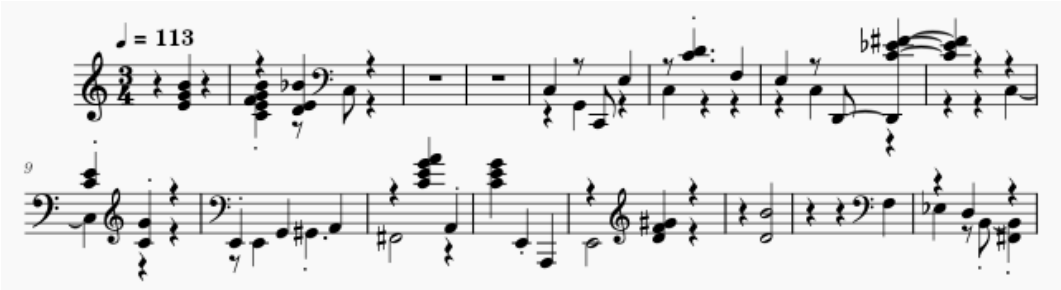
Um Musik zu komponieren wird das Programm für 1000 Epochen mit zehn Jazzsongs trainiert. Dabei wird jede Epoche der Gradientenabstieg angewandt, um den Fehler in der Approximation des Netzes zu reduzieren. Der Verlauf dieses Fehlers sieht für die ersten 500 Epochen wie folgt aus:



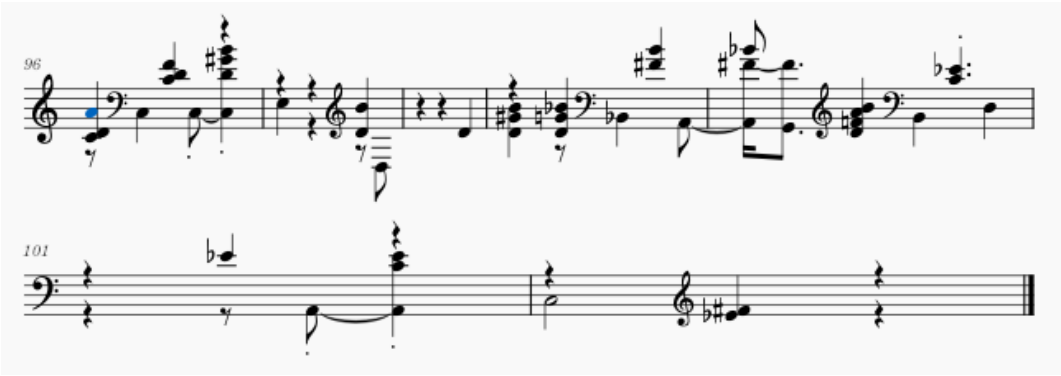
Hier lässt sich deutlich erkennen, dass dieser bei jeder Epoche nach unten geht und sich der Fehler des gesamten Netzes so insgesamt jeden Schritt verkleinert. Um zu testen, ob das neuronale Netz tatsächlich die Stücke, anhand dessen es trainiert wurde, approximieren kann, wird dem trainierten Netz der Anfang eines bekannten Stückes gegeben. Mit diesem wird folgendes Ergebnis komponiert:



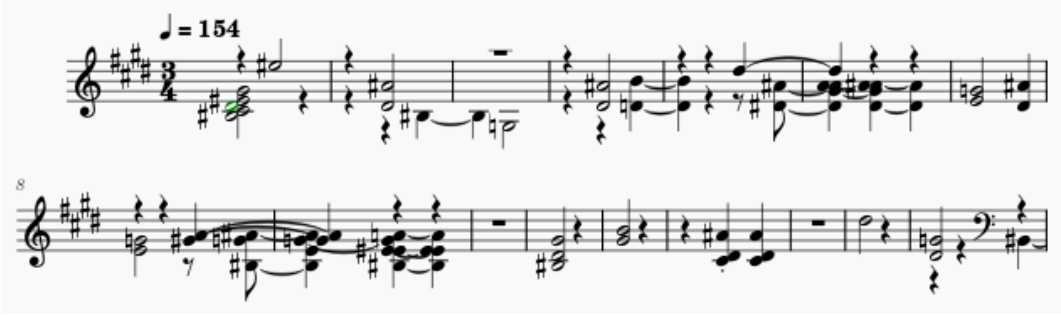
Dieses ist identisch mit dem weiteren Verlauf des originalen Stückes. Dieses Experiment zeigt also, dass das in dieser Arbeit erläuterte neuronale Netz eine Funktion, die das Stück beschreibt, approximieren kann. Gibt man dem neuronalen Netz ein Stück, was den Stücken, mit denen trainiert wurde, sehr ähnlich ist, kann eine solche Komposition beispielsweise so aussehen:



Dieser Ausschnitt erinnert stark an eins der originalen Stücke und ist diesem fast identisch. Daher sind auch die musikalischen Elemente, die man von einem Jazzstück erwarten würde, gegeben. Auffällig ist hier, dass diese Gemeinsamkeiten gegen Ende des Stückes abnehmen. Ein Ausschnitt aus dem hinteren Teil sieht beispielsweise so aus:



Dies lässt sich vermutlich darauf zurückzuführen, dass das neuronale Netz in seiner Approximation und seinem Verständnis von Musik nicht perfekt ist. Wird eine Note falsch vorhergesagt, so wird diese Note verwendet, um die weiteren Noten des Stückes zu ermitteln. Selbst ein kleiner Fehler zu Beginn kann sich daher auf das gesamte Stück negativ auswirken. Zeigt man dem neuronalen Netz mehrere Anfänge von stilistisch unterschiedlichen Stücken, so kann beispielsweise folgendes Ergebnis erzeugt werden:



Auch wenn dieses generierte Stück sich vergleichsweise weniger musikalisch

anhört, sind doch Ansätze musikalischer Elemente zu erkennen und die komponierte Sequenz ist mehr als eine zufällige Anordnung von Notenwerten. So ist beispielsweise in der ersten Zeile der Komposition mehrmals der Ton h zu finden, der in verschiedenen Akkorden wiederholt gespielt wird. Hier fehlen allerdings rhythmische Muster fast komplett und die Töne harmonieren eher weniger miteinander.

9 Schlussbetrachtung

Das in dieser Seminararbeit programmierte Programm konnte erfolgreich Musikstücke komponieren. Es war in der Lage, mehrere Musikstücke auf eine mathematische Funktion zu reduzieren und somit zu approximieren. Die Frage, die sich hier stellt ist, ob das Programm in der Lage war zu lernen, was Musik ist. Das Anwenden des neuronalen Netzes auf ein unbekanntes Stück konnte zeigen, dass es nicht fundamental unmöglich ist, kreative Aufgaben durch Maschinen und Mathematik zu ersetzen. Hier stellt sich außerdem die Frage, ob die komponierten Stücke wirklich kreativ und neu sind. Eine Anwendung auf Stücke, die den originalen Stücken ähnlich waren, zeigt, dass sich das Netz in diesem Fall eher bestimmte Notenfolgen gemerkt hat, als wirklich neue Stücke oder Motive zu erfinden. Wendet man das neuronale Netz auf ein, komplett unbekanntes, Stück an, so hören sich diese vergleichsweise schlechter an. Trotzdem konnten in diesen Stücken Ansätze von musikalischem Verständnis aufgezeigt werden. Der Grund für die schlechteren Ergebnisse, die das neuronale Netz mit neuen Kompositionen erzeugt, liegt vermutlich in der begrenzten Menge an Stücken, mit der das neuronale Netz trainiert wurde. Um die Trainingszeiten gering zu halten und so mehrere Experimente durchführen zu können, musste diese stark reduziert werden. Dadurch muss das neuronale Netz anhand weniger Jazz Stücke allgemeine Erkenntnisse über Musik erlangen. Es ist anzunehmen, dass das Trainieren dieses Netzes mit mehr Daten in deutlich mehr Kreativität resultieren kann.

Quellenverzeichnis

Programmcode: https://github.com/Skuldur/Classical-Piano-Composer (Stand: 28.10.2022)

Trainingsdaten: https://bushgrafts.com/midi/ (Stand: 28.10.2022)

Latex Code Neuron: https://tex.stackexchange.com/questions/365519/draw- single-neural-unit-using-tizk (Stand: 01.11.2022)

Latex Code LSTM: https://tex.stackexchange.com/questions/432312/how-do-i-draw-an-lstm-cell-in-tikz (Stand: 01.11.2022)

Latex Code neuronales Netz: https://tex.stackexchange.com/questions/153957/drawing-neural-network-with-tikz (Stand: 01.11.2022)

Literatur

Brownlee, J. (2018). machinelearningmastery, A Gentle Introduction to Dropout for Regularizing Deep Neural Networks. https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/. (Stand: 06.11.2022).

Bushaev, V. (2018). towardsdatascience, Adam — latest trends in deep learning optimization. https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c. (Stand: 06.11.2022).

Hochreiter, S. und Schmidhuber, J. (1997). LONG SHORT-TERM MEMORY. *Neural Computation*.

Ioffe, S. und Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.

Karpathy, A. (2018). github, The Unreasonable Effectiveness of Recurrent Neural Networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness/. (Stand: 03.08.2022).

Kingma, D. P. und Ba, J. L. (2015). ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. ICLR Konferenz 2015, San Diego.

Krishna, N. (2022). towardsdatascience, Backpropagation in RNN Explained. https://towardsdatascience.com/backpropagation-in-rnn-explained-bdf853b4e1c2. (Stand: 06.11.2022).

Kurbiel, T. (2020). Medium, Deriving the Backpropagation Equations from Scratch. https://towardsdatascience.com/deriving-the-backpropagation-equations-from-scratch-part-1-343b300c585a. (Stand: 06.11.2022).

Kurbiel, T. (2021). Medium, Derivative of the Softmax Function and the Categorical Cross-Entropy Loss. https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1. (Stand: 06.11.2022).

Nielsen, M. (2019). Neural Networks and Deep Learning. http://neuralnetworksanddeeplearning.com. (Stand: 03.08.2022).

Skúli, S. (2017). towardsdatascience, How to Generate Music using a LSTM Neural Network in Keras. https://towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4c5. (Stand: 28.10.2022).

Anhang

.1 Programmcode

```
import glob
import pickle
import numpy
import datetime
from music21 import instrument, note, stream, chord, converter
import tensorflow as tf
from keras.utils import np_utils
from keras.callbacks import ModelCheckpoint
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.layers import BatchNormalization as BatchNorm
from keras.layers import Activation

log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)


# lernen

def train_network():
    """ Train a Neural Network to generate music """
    notes = get_notes()

    # get amount of pitch names
    n_vocab = len(set(notes))

    network_input, network_output = prepare_sequences_train(notes, n_vocab)

    model = create_network(network_input, n_vocab)

    train(model, network_input, network_output)


def get_notes():
    """ Get all the notes and chords from the midi files in the midi_songs directory """
    notes = []

    for file in glob.glob(r"midi_songs/*.mid"):
        midi = converter.parse(file)

        print("Parsing %s" % file)

        try:  # file has instrument parts
            s2 = instrument.partitionByInstrument(midi)
            notes_to_parse = s2.parts[0].recurse()
        except:  # file has notes in a flat structure
            notes_to_parse = midi.flat.notes

        for element in notes_to_parse:
            if isinstance(element, note.Note):
                notes.append(str(element.pitch) + " " + str(element.quarterLength))
            elif isinstance(element, chord.Chord):
                notes.append('.'.join(str(n) for n in element.normalOrder) + " " + str(element.quarterLength))
            elif isinstance(element, note.Rest):
                notes.append(str(element.name) + " " + str(element.quarterLength))

    with open('data/notes', 'wb') as filepath:
        pickle.dump(notes, filepath)

    return notes


def prepare_sequences_train(notes, n_vocab):
    """ Prepare the sequences used by the Neural Network """
    sequence_length = 100

    # get all pitch names
    pitchnames = sorted(set(item for item in notes))

    # create a dictionary to map pitches to integers
    note_to_int = dict((note, number) for number, note in enumerate(pitchnames))

    network_input = []
```

```
    network_output = []

    # create input sequences and the corresponding outputs
    for i in range(0, len(notes) - sequence_length, 1):
        sequence_in = notes[i:i + sequence_length]
        sequence_out = notes[i + sequence_length]
        network_input.append([note_to_int[char] for char in sequence_in])
        network_output.append(note_to_int[sequence_out])

    n_patterns = len(network_input)

    # reshape the input into a format compatible with LSTM layers
    network_input = numpy.reshape(network_input, (n_patterns, sequence_length, 1))
    # normalize input
    network_input = network_input / float(n_vocab)

    network_output = np_utils.to_categorical(network_output)

    return (network_input, network_output)


def create_network(network_input, n_vocab):
    """ create the structure of the neural network """
    model = Sequential()
    model.add(LSTM(
        512,
        input_shape=(network_input.shape[1], network_input.shape[2]),
        recurrent_dropout=0.3,
        return_sequences=True
    ))
    model.add(LSTM(1024, return_sequences=True, recurrent_dropout=0.3, ))
    model.add(LSTM(1024))
    model.add(BatchNorm())
    model.add(Dropout(0.3))
    model.add(Dense(512))
    model.add(Activation('relu'))
    model.add(BatchNorm())
    model.add(Dropout(0.3))
    model.add(Dense(n_vocab))
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam')

    return model


def train(model, network_input, network_output):
    """ train the neural network """
    filepath = "train_weights/weights-improvement-{epoch:02d}-{loss:.4f}-bigger.hdf5"
    checkpoint = ModelCheckpoint(
        filepath,
        monitor='loss',
        verbose=0,
        save_best_only=True,
        mode='min'
    )
    callbacks_list = [checkpoint, tensorboard_callback]

    model.fit(network_input, network_output,  epochs=1000, callbacks=callbacks_list)


# komponieren

def prepare_sequences_generate(notes, pitchnames, n_vocab):
    """ Prepare the sequences used by the Neural Network """
    # map between notes and integers and back
    note_to_int = dict((note, number) for number, note in enumerate(pitchnames))

    sequence_length = 100
    network_input = []
    output = []
    for i in range(0, len(notes) - sequence_length, 1):
        sequence_in = notes[i:i + sequence_length]
        sequence_out = notes[i + sequence_length]
        network_input.append([note_to_int[char] for char in sequence_in])
        output.append(note_to_int[sequence_out])
```

```
    n_patterns = len(network_input)

    # reshape the input into a format compatible with LSTM layers
    normalized_input = numpy.reshape(network_input, (n_patterns, sequence_length, 1))
    # normalize input
    normalized_input = normalized_input / float(n_vocab)

    return (network_input, normalized_input)

def generate():
    """ Generate a piano midi file """

    # load the notes used to train the model
    with open('data/notes', 'rb') as filepath:
        notes = pickle.load(filepath)

    # Get all pitch names
    pitchnames = sorted(set(item for item in notes))
    # Get all pitch names
    n_vocab = len(set(notes))

    network_input, normalized_input = prepare_sequences_generate(notes, pitchnames, n_vocab)
    model = create_network(normalized_input, n_vocab)
    prediction_output = generate_notes(model, network_input, pitchnames, n_vocab)
    create_midi(prediction_output)


def generate_notes(model, network_input, pitchnames, n_vocab):
    """ Generate notes from the neural network based on a sequence of notes """
    # pick a random sequence from the input as a starting point for the prediction
    start = numpy.random.randint(0, len(network_input)-1)

    int_to_note = dict((number, note) for number, note in enumerate(pitchnames))

    pattern = network_input[start]
    prediction_output = []

    # generate 500 notes
    for note_index in range(500):
        prediction_input = numpy.reshape(pattern, (1, len(pattern), 1))
        prediction_input = prediction_input / float(n_vocab)

        prediction = model.predict(prediction_input, verbose=0)

        index = numpy.argmax(prediction)
        result = int_to_note[index]
        prediction_output.append(result)

        pattern.append(index)
        pattern = pattern[1:len(pattern)]

    return prediction_output


def create_midi(prediction_output):
    """ convert the output from the prediction to notes and create a midi file
        from the notes """
    offset = 0
    output_notes = []

    # create note and chord objects based on the values generated by the model
    for pattern in prediction_output:
        pattern = pattern.split()
        temp = pattern[0]
        duration = pattern[1]
        pattern = temp
        # pattern is a chord
        if ('.' in pattern) or pattern.isdigit():
            notes_in_chord = pattern.split('.')
            notes = []
            for current_note in notes_in_chord:
                new_note = note.Note(int(current_note))
                new_note.storedInstrument = instrument.Piano()
                notes.append(new_note)
```

```
            new_chord = chord.Chord(notes)
            new_chord.offset = offset
            output_notes.append(new_chord)
        # pattern is a rest
        elif 'rest' in pattern:
            new_rest = note.Rest(pattern)
            new_rest.offset = offset
            new_rest.storedInstrument = instrument.Piano()
            output_notes.append(new_rest)
        # pattern is a note
        else:
            new_note = note.Note(pattern)
            new_note.offset = offset
            new_note.storedInstrument = instrument.Piano()
            output_notes.append(new_note)
        # increase offset each iteration so that notes do not stack
        offset += convert_to_float(duration)

    midi_stream = stream.Stream(output_notes)

    midi_stream.write('midi', fp='test_output.mid')


def convert_to_float(frac_str):
    try:
        return float(frac_str)
    except ValueError:
        num, denom = frac_str.split('/')
        try:
            leading, num = num.split(' ')
            whole = float(leading)
        except ValueError:
            whole = 0
        frac = float(num) / float(denom)
        return whole - frac if whole < 0 else whole + frac


if __name__ == '__main__':
    generate()
    train_network()
```

Ich erkläre hiermit, dass ich die Seminararbeit ohne fremde Hilfe angefertigt habe und nur die im Literaturverzeichnis angegebenen Quellen und Hilfsmittel benutzt habe.

Vaterstetten, den 06.11.2022 Justin

(Unterschrift des Schülers)