KI in der Musik – Computer lernen komponieren

Alexander Reimer und Matteo Friedrich

Gymnasium Eversten Oldenburg

Betreuer: Herr Dr. Glade & Herr Husemeyer

Inhaltsverzeichnis

1 Kurzfassung **3**

2 Einleitung **4**

3 Methode und Vorgehensweise **4**
3.1 Materialien 4
3.2 Vorgehensweise 4
3.2.1 Struktur und Forward Pass – Theorie 4
3.2.2 Struktur und Forward Pass – Umsetzung 6
3.2.3 Backpropagation – Theorie 7
3.2.4 Konfiguration der Trainingsdaten 9
3.2.5 Darstellung der Funktionsweise des Programms 10
3.2.6 Symbole/Syntax – Verzeichnis 10

4 Ergebnisse **10**
4.1 Teilbarkeit – Erster Testversuch 10
4.2 Die Musik 11

5 Diskussion **12**

6 Quellen **14**
6.1 Bücher 14
6.2 Internetquellen 14
6.3 Julia-Packages 14
6.4 Midi-Dateien für Trainings-/Testdaten 14
6.5 Abbildungen 15

7 Unterstützungsleistungen **15**

1 Kurzfassung

In unserem Projekt haben wir eine künstliche Intelligenz (KI) programmiert, die Komponieren lernen soll. Dafür haben wir die Programmiersprache Julia verwendet. Wir haben herausgefunden, dass Machine Learning fast immer mit neuronalen Netzwerken funktioniert. Ein neuronales Netz ist eine Ansammlung von, in mehrere Layer unterteilten, Neuronen. Die Neuronen haben eine Aktivierung und sind mit Gewichten verbunden. Die Aktivierung eines Neurons setzt sich aus den Aktivierungen des vorherigen Layers und den damit verbundenen Gewichten zusammen. Der erste Layer ist der Input Layer. Bei diesem werden die Aktivierungen der Neuronen von Daten bestimmt, z.B. den Pixeln eines Bildes.

Damit die KI nun lernen kann, benötigt sie Trainingsdaten. Diese bestehen aus Paaren von Inputs und den richtigen Outputs dafür (z. B. ein Text und die dazugehörige Textart). Durch sie weiß das Programm, wie ein Netzwerk verändert werden muss, und kann die Leistung eines neuronalen Netzwerkes überprüfen, indem es die Cost berechnet. Um die Cost zu berechnen, haben wir die MSE-Funktion benutzt. Es gilt: Die Cost ist die Summe der Fehler eines neuronalen Netzwerkes, d. h., niedrigere Cost = besseres Netzwerk.

Die KI lernt nun, indem die Gewichte des Netzwerks optimiert werden, das heißt, sie werden so verändert, dass das Netzwerk eine möglichst kleine Cost hat. Um das Netzwerk zu optimieren, nutzen wir Backpropagation.

Anschließend haben wir Midi-Dateien aus dem Internet heruntergeladen und mit Hilfe von ihnen Trainingsdaten erzeugt. In den Trainingsdaten sind 10 aufeinander folgende Töne eines Liedes als Inputs und der, auf den 10. Ton folgende Ton, als richtiger Output angegeben. Am Ende haben wir ein neuronales Netzwerk mit Hilfe dieser Trainingsdaten optimiert. Dieses Netzwerk haben wir genutzt, um neue Lieder zu komponieren. Das Ergebnis war sehr enttäuschend, weil die Ergebnisse für das menschliche Gehör zufällig und nicht schön wirkten. Bei einfacheren Tests hat das Programm jedoch gut funktioniert.

Wir haben auch mehrere Möglichkeiten gefunden, unser Netzwerk zu verbessern. Wir hatten außerdem noch eine andere Idee, wie auf unserem Projekt aufbauen könnte.

2 Einleitung

Die Forscherfrage:

Wir haben es uns als Ziel gesetzt, eine Künstliche Intelligenz zu programmieren, die man mithilfe von Liedern eines Genres trainieren kann, eigene Lieder zu komponieren. Der Grund dafür ist, dass wir uns für Machine Learning interessieren, da es heutzutage eine sehr wichtige Technologie ist, aber trotzdem nur von wenigen verstanden wird. Dafür wollten wir mit Hilfe der Programmiersprache Julia ein eigenes neuronales Netzwerk erstellen und trainieren. Wir haben uns für den Themenbereich Musik entschieden, weil wir es spannend fanden, dass Computer etwas so menschliches wie Musik verstehen und verarbeiten können. Besonders wichtig war es uns dabei, die Teile des Programms über neuronale Netzwerke und Machine Learning selbst zu schreiben, ohne bereits fertige Lösungen zu benutzen. Sachen wie das grafische Darstellen von Daten und das Auslesen von Midi-Dateien innerhalb von Julia haben wir jedoch aus dem Internet kopiert.

3 Methode und Vorgehensweise

3.1 Materialien

Für unser Projekt haben wir folgende Materialien/Applikationen verwendet:

- Hardware
  - Verschiedene Computer
- Software
  - Julia, eine objektorientierte, schnelle, dynamische und übersichtliche Programmiersprache, die speziell für mathematische Berechnungen und Datenverarbeitung ausgelegt ist
  - Julia Packages:
    - MIDI, um mit Musikdateien und Noten zu arbeiten
    - PyPlot, um Plots darzustellen
    - JLD, um Netzwerke und Plots zu speichern
- Midi-Dateien zum Trainieren und Testen:
  - https://bitmidi.com/
  - https://www.midiworld.com/files/

3.2 Vorgehensweise

3.2.1 Struktur und Forward Pass – Theorie

Zuerst wollten wir die Grundstruktur eines neuronalen Netzwerkes programmieren. Nach einiger Recherche haben wir herausgefunden, dass ein neuronales Netzwerk aus drei Teilen besteht: dem Input Layer, den Hidden Layers und dem Output Layer [1, 2, 3].

Der Input Layer ist nur eine Liste aus Zahlen zwischen 0 und 1. Er gibt an, welche Eingaben (Inputs) das Netzwerk bekommen soll, z. B. ein Schwarz-Weiß-Bild oder ein Text.

Die Hidden Layers sind eine Ansammlung von in mehrere Layer (Schichten) unterteilten Neuronen. Jedes Neuron besitzt eine Aktivierung (Activation), die als Zahl zwischen 0 und 1 angegeben werden kann, und einen Bias (Verzerrung), der eine beliebige Zahl sein kann. Die Neuronen verschiedener Layer sind alle durch sogenannte Gewichte (Weights) verbunden, die ebenfalls einen beliebigen Wert haben können. Um nun die Aktivierungen eines Neurons zu

berechnen, gibt es den sogenannten Forward Pass. Dabei beginnt man im ersten Hidden Layer damit, für alle Neuronen den sogenannten Netz Input (auch net input) zu berechnen. Um den Netz Input eines Neurons zu berechnen, werden alle Aktivierungen des vorherigen Layers mit den von dem Neuron dorthin führenden Gewichten multipliziert und summiert. Der Bias ist eigentlich auch ein Gewicht, jedoch ist er mit einem Neuron verbunden, das IMMER die Aktivierung (a_i) 1 hat, weshalb man einfach $+b_i$ statt $+b_i * a_i$ rechnen kann. Um aus diesem Netz Input nun die Aktivierung zu berechnen, benötigt man eine Aktivierungsfunktion, die dafür sorgt, dass die Aktivierung zwischen 0 und 1 liegt. Wir haben dafür eine Sigmoidfunktion benutzt, die eine Zahl nimmt und einen Wert zwischen 0 und 1 ausgibt [2]:

$$sig(x) = \frac{1}{1 + e^{-x}}$$

$$a_i = sig\left(netzinput_i\right)$$

wobei a_i = die Aktivierung des Neurons i und $netzinput_i$ = der Netzinput des Neurons i.

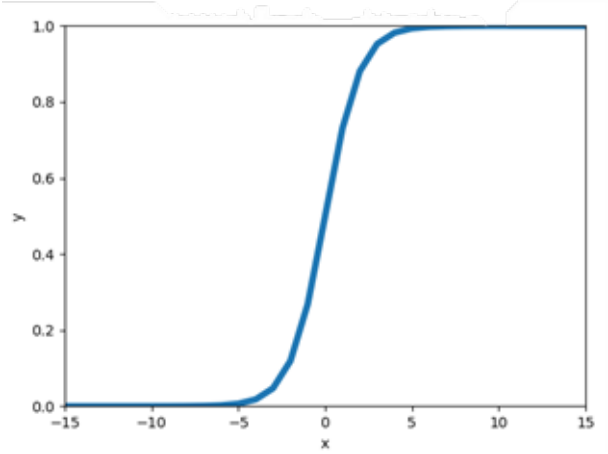

Abbildung 1: Unsere Sigmoidfunktion

Dies wird dann für jedes Neuron in jedem Layer wiederholt. Da man aber immer die Aktivierungen des vorherigen Layers benötigt, muss man das ganze vom Input Layer zum Output Layer machen. Daher auch der Name Forward Pass. Die Outputs sind dann lediglich die Aktivierungen der Neuronen im Output Layer, die Interpretation dieser hängt von den Trainingsdaten ab (s. Backpropagation – Theorie). Die allgemeine Formel für die Aktivierung eines Neurons lautet also:

$$a_j = sig\left(\sum_L (a_L * W_{Lj}) + b_j\right)$$

wobei a_j = die Aktivierung des Neurons j, L = der nächste Layer, a_L = alle Aktivierungen des Layers L , W_{Lj} = alle Gewichte zwischen dem Neuron j und den Neuronen des Layers L, und b_j = der Bias des Neurons j.

Eine Beispielrechnung für ein winziges Netzwerk:

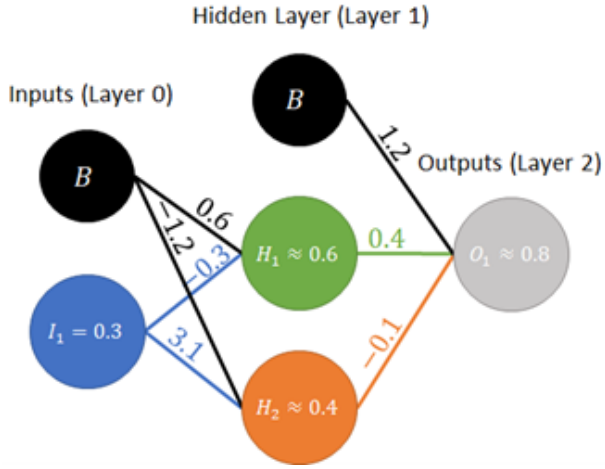

Abbildung 2: Ein einfaches Beispiel für ein neuronales Netzwerk mit 3 Layern (Input, Hidden, und Output Layer), 4 Neuronen und 2 „Bias“-Neuronen. Die Neuronen sind durch Kreise mit der jeweiligen Aktivierung abgebildet. Die Linien zeigen die Verbindungen zwischen den Neuronen mit den dazugehörigen Gewichten. Die schwarzen Kreise zeigen die Bias-Neuronen, deren Aktivierung immer 1 ist.

$$H_1(netzinput) = 0.3 * (-0.3) + 0.6 = 0.51$$

$$H_1(Aktivierung) = sig(0.51) \approx 0.62480647$$

$$H_2(netzinput) = 0.3 * 3.1 + (-1.2) = -0.27$$

$$H_2(a) = sig(-0.27) \approx 0.432907$$

$$O_1(ni) \approx 0.4 * 0.62 + (-0.1) * 0.43 + 1.2 = 1.406631880283917$$

$$O_1(a) = 0.803234159160199$$

Ergebnis: [0.803234159160199]

Jedoch ist das Beispiel ein sehr kleines Netzwerk, das so in der Praxis nicht angewendet werden würde. Neuronale Netzwerke können (theoretisch) unendlich groß sein und die Anzahl der Gewichte vergrößert sich schon bei ein paar zusätzlichen Neuronen drastisch:

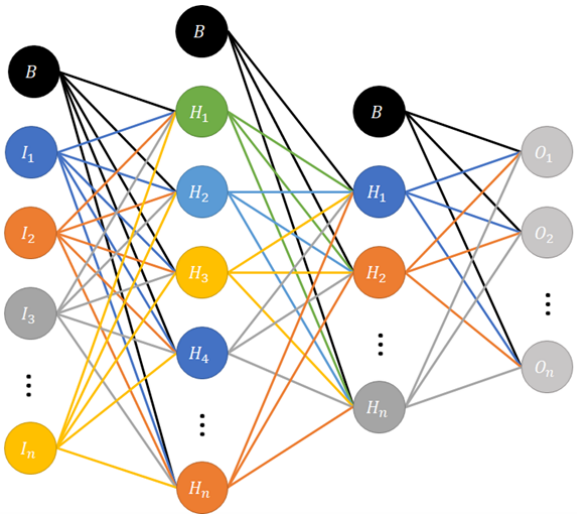

Abbildung 3: Ein größeres neuronales Netzwerk mit 4 Layern

3.2.2 Struktur und Forward Pass – Umsetzung

Um die Struktur eines neuronalen Netzwerkes in Julia umzusetzen, haben wir sogenannte *Mutable Structs* benutzt. Mit diesen kann man eigene Datentypen definieren, die selbst erstellte „Felder“ haben. Hier am Beispiel eines Datentyps zum Speichern eines Neurons:

```
mutable struct Neuron
    netinput::Float64
    activation::Float64
    δ::Float64
end
```

Nachdem so ein Datentyp einmal definiert wurde, kann man ganz einfach mit z.B. Neuron.activation auf die Aktivierung oder Neuron.netinput auf den Netz Input zugreifen.

Man kann Mutable Structs auch ineinander verschachteln, was wir ausgenutzt haben, um unsere Netzwerkstruktur zu erstellen.

Neuronales Netzwerk vom höchsten bis zum niedrigsten Level:

network →Das gesamte neuronale Netzwerk

- allInputs (als Array von Float) →Input Layer
- allLayers (als Array von Layer) →Alle Layer des Netzwerkes außer Inputs
  Layer →Ein Layer
  - neurons (als Array von Neuron) →Alle Neuronen eines Layers
    Neuron →Ein Neuron
    - netinput (als Float) →Der Netzinput eines Neurons
    - activation (als Float) →Die Aktivierung eines Neurons
    - δ (als Float) →Der δ-Wert des Neurons
- allWeights (als Array von einem 2D Array von Float) →alle Weights

Mithilfe der von uns programmierten Funktion `createNetworkStructure(...)` können wir ganz einfach ein neuronales Netzwerk mit beliebig vielen und großen Layern erstellen. Wie üblich werden die Gewichte und Biases mit zufälligen Werten initialisiert, die in unserem Fall zwischen -2 und 2 liegen. Mit der Funktion `forwardPass(inputs...)` kann man ganz einfach alle Aktivierungen des Netzwerkes berechnen.

3.2.3 Backpropagation – Theorie

Doch woher weiß man, welche Werte die Gewichte und Biases haben sollen? Immerhin haben normale Netzwerke tausende von Gewichten und Biases, deren Verbindungen meist für Menschen nicht nachvollziehbar sind.

Nach einiger Recherche im Internet sind wir auf ein Optimierungsverfahren gestoßen: Backpropagation (auch Backward Pass genannt), letztendlich eine effizientere und vergleichsweise einfach anwendbare Version des Gradient Descents, mit der das lokale Minimum gefunden werden kann. Für Backpropagation muss das Programm jedoch wissen, wie gut eine bestimmte "Konfiguration" (Gewichte und Biases) des Netzwerkes ist. Dafür gibt es die sogenannte Cost-Funktion (auch Loss-Funktion genannt), die mithilfe von Trainingsdaten funktioniert. Trainingsdaten bestehen aus einer Liste aus Trainingsdatensätzen. Jeder dieser Datensätze beinhaltet Inputs für das Netzwerk und die richtigen Outputs dafür. [1, 2] Machine Learning, das mit solchen Trainingsdaten arbeitet, wird Supervised Learning genannt. [1] In diesen Datensätzen liegt meist auch die Schwierigkeit von Machine Learning (s. Konfiguration der Trainingsdaten).

Die Funktion für die Cost des Output Layers und somit gesamten Netzwerkes (für einen Trainingsdatensatz) lautet wie folgt [2]:

$$C_0 = (a_L - y)^2$$

wobei C_0 = die Cost des Output Layers, L = der letzte Layer (Output Layer), $a^{(L)}$ = die Aktivierungen des Layers L, und y = die richtigen Outputs für die Inputs, mit denen die Aktivierungen berechnet wurden.

Um die Cost zu berechnen, muss man also für alle Output Neuronen die Differenz der gegebenen und der richtigen Aktivierungen bilden. Danach muss man diese Differenzen quadrieren und am Ende alle Ergebnisse aufsummieren. Dies kann man für alle Trainingsdatensätze wiederholen und von allen Costs den Durchschnitt nehmen, um die allgemeine Performance eines Netzwerkes

zu überprüfen. Diese Art der Cost-Funktion wird Mean Squared Error (MSE) genannt. Zusammenfassend kann man also sagen, dass die Cost die Abweichung von den berechneten und den richtigen Outputs angibt.

Aufgrund des Trainingsdatensatzes weiß man nun, wie der Output Layer verändert werden muss.

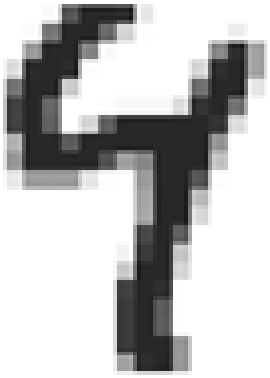

Abbildung 4: Eine handgeschriebene 4[6]

Tabelle 1: Beispielhafte Outputs eines untrainierten neuronalen Netzwerkes mit Abbildung 4 als Input, und die richtigen Outputs dafür – also ein Trainingsdatensatz

entspricht der Zahl	Output	richtiger Output	ändern um:
0	0.867	0.0	−0.867
1	0.598	0.0	−0.598
2	0.0012	0.0	−0.0012
3	1.0	0.0	−1.0
4	0.386	1.0	+0.614
5	0.81	0.0	−0.81
6	0.9976	0.0	−0.9976
7	0.0	0.0	+0.0
8	0.754	0.0	−0.754
9	0.6	0.0	−0.6

Doch wie verändert man nun die Aktivierungen des Output Layers? Es müssen alle Gewichte und Biases davor angepasst werden. Um nun zu wissen, wie ein Gewicht verändert werden muss, gibt es folgende Funktion [3]:

$$\Delta W_{ij} = \epsilon * \delta_i * a_j$$

wobei ΔW_{ij} = um wie viel das Gewicht W zwischen den Neuronen j und i verändert werden muss, ϵ = die Lernrate (meist ein kleiner Wert wie 0.001), $\delta_i \approx$ die Ableitung der Cost des Neuronen i im Verhältnis zum Weight W_{ij}, und a_j = die Aktivierung des Neurons j. Was dabei oft verwirrend ist: j bezeichnet das Neuron, welches zuerst kommt, und i das Neuron, welches danach kommt (Reihenfolge im Forward-Pass), obwohl es bei W_{ij} andersherum steht.

Für den Bias wird die gleiche Formel benutzt, mit der Ausnahme, dass a_j immer 1 ist und so wegfällt. Der Grund dafür liegt darin, dass der Bias, wie in der Struktur beschrieben, eigentlich nur ein Gewicht ist, das mit einem Neuron verbunden ist, welches immer eine Aktivierung von eins hat.

Um nun δ_i für den Output Layer zu berechnen, gibt es folgende Gleichung [3]:

$$\delta_i = sig'(netzinput_i) * (a_i(soll) - a_i(ist))$$

wobei $sig'(x)$ = die Ableitung von $sig(x)$, also $sig'(x) = sig(x) * (1 - sig(x))$, $netzinput_i$ = der Netzinput des Neurons i , $a_i(soll)$ = die Aktivierung, die das Neuron haben sollte (also das gleiche wie y), und $a_i(ist)$ = die Aktivierung, die das Neuron hat.

Mit dieser Formel wird berechnet, welche Aktivierung das Neuron haben sollte, was an $(a_i(soll) - a_i(ist))$ erkennbar ist. Die Aktivierungsfunktion mit dem Netz Input wird als Faktor mit einberechnet, da möglichst nur die Gewichte stark verändert werden sollen, die bei dem Trainingsdatensatz eine hohe Aktivierung haben, also durch diese Inputs besonders angesprochen werden. So werden zum Beispiel beim Sortieren nur die Neuronen miteinander verknüpft, die für ein bestimmtes Muster verantwortlich sind.

Für die Neuronen der Hidden Layers muss man alle δ's des nächsten Layers mit den von dem Neuron dorthin führenden Weights multiplizieren und dann summieren [3]. Dadurch werden die Änderungen, die die Aktivierungen dieser Neuronen brauchen (δ), zusammengerechnet, da

natürlich die Aktivierungen im nächsten Layer unterschiedliche Änderungen in dem gleichen Neuron benötigen. Durch die Multiplikation mit den dahin führenden Weights werden diese Änderungen gewichtet, da sie auf einige Neuronen größere Auswirkungen haben als auf andere. Wie beim Output Layer auch wird diese Summe noch mit $sig'(netzinput)$ multipliziert, um die Aktivierung durch bestimmte Muster angesprochener Neuronen noch weiter zu erhöhen und weniger/kaum angesprochener Neuronen zu senken, sodass die Ergebnisse besser und eindeutiger werden. Die Formel [3]:

$$sig'(netzinput_i) * \sum_L (\delta_L * W_{Li})$$

wobei L = der nächste Layer, δ_L = alle δ's des Layers L, und W_{Li} = alle Weights, die ein Neuron des nächsten Layers und Neuron i verbinden.

Da immer die nächsten Layer und der Output Layer benötigt werden, ergibt es Sinn, diese Optimierung beim Output Layer zu starten und dann rückwärts die δ-Werte für jeden Layer zu berechnen und für die nächsten Berechnungen zu speichern – daher auch der Name Backpropagation.

3.2.4 Konfiguration der Trainingsdaten

Die Trainingsdaten für ein neuronales Netzwerk zu finden ist einer der schwierigsten Teile, denn wenn man diese automatisch generieren könnte, wäre keine KI mehr notwendig. Also müssen diese Trainingsdaten oft von Menschen erstellt werden. Das Problem: Um gut trainiert zu werden, benötigen neuronale Netzwerke tausende Trainingsdatensätze. Für Musik ist das einfacher, weil es schon sehr viele Lieder gibt.

Nach einiger Recherche wurde klar, dass wir die KI keine eigenen Lieder komponieren lassen können, denn dann könnten wir das Machine Learning nicht mit Supervised Learning, also Trainingsdaten, die die richtigen Outputs für bestimmte Inputs enthalten, durchführen, sondern müssten auf Reinforcement Learning zurückgreifen, das heißt, nach jedem Trainingsdatensatz, für alle Wiederholungen der Backpropagation, müsste sich ein Mensch das erzeugte Lied anhören und es bewerten. Das wäre nicht realistisch machbar. (s. Diskussion für Unsupervised Learning als Alternative)

Also sind wir stattdessen auf die Idee gekommen, das neuronale Netzwerk immer nur eine weitere Note bestimmen zu lassen, nachdem es x viele vorherige als Inputs bekommen hat. Denn diese könnte man automatisch kontrollieren, da man ja die nächste Note in einem Lied kennt. Und sobald das Netzwerk gut genug trainiert ist, kann man es auch neue Noten über die des originalen Liedes hinaus bestimmen lassen, sodass auch „originale“, neue Teile komponiert werden.

Das große Problem hierbei ist die Tatsache, dass neuronale Netzwerke nur Zahlen als Input oder Output bekommen können. Deshalb mussten wir eine Möglichkeit finden, einzelne Noten in Zahlen umzuwandeln. Um eine gute Umwandlungsmethode zu finden, mussten wir zuerst definieren, was eine gute Konfiguration einer Note ausmacht. Damit ein neuronales Netzwerk gut mit einem Trainingsdatensatz lernen kann, sollte dieser nicht zu lang sein, weil sonst zu viele Gewichte angepasst werden müssen. Außerdem sollten kleine Änderungen in der Konfiguration auch nur kleine Auswirkungen auf die zugehörige Note haben, weil ein neuronales Netzwerk meistens eine kleine Fehlerquote hat.

Am Ende hatten wir uns dazu entschieden, die Tonhöhe, die Dauer, die der Ton erklingt, und die Position relativ zum vorherigen Ton in unsere Konfiguration einzubauen. Da Ungenauigkeiten bei zum Beispiel der Dauer von kurzen Noten schlimmer sind als von langen Noten, haben wir diese Werte logarithmisch umgewandelt.

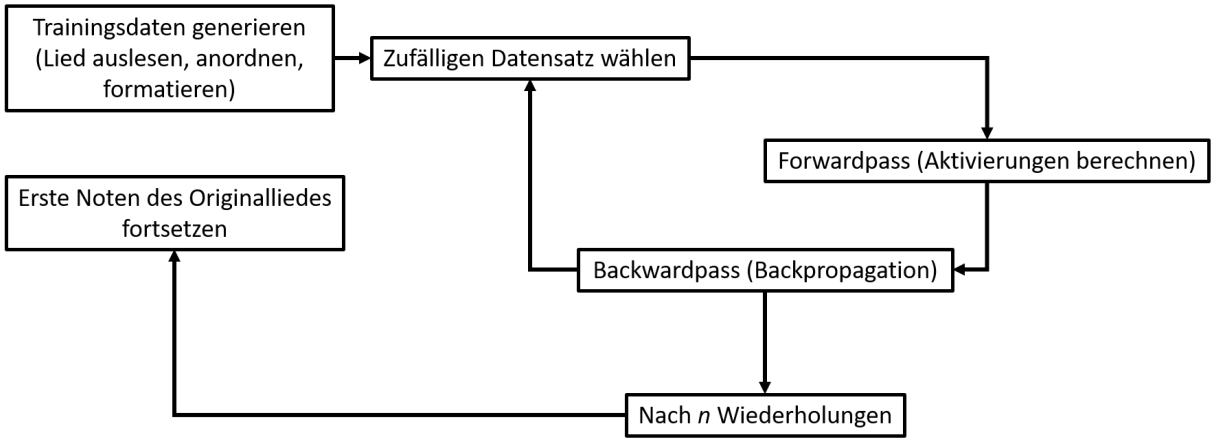

Abbildung 5: Die grobe Funktionsweise unseres Programms

3.2.5 Darstellung der Funktionsweise des Programms

3.2.6 Symbole/Syntax – Verzeichnis

Die wichtigsten Symbole:

Tabelle 2: Verzeichnis der wichtigsten verwendeten Symbole

Symbol	Bedeutung
a_i	Die Aktivierung des Neurons i
b_i	Der Bias des Neurons i
$netzinput_i$	Der Netzinput des Neurons i
a_L	Alle Aktivierungen des Layers L
W_{Lj}	Alle Gewichte zwischen dem Neuron j und den Neuronen des Layers L
W_{ij}	Das Gewicht zwischen den Neuronen i und j. j bezeichnet das Neuron, welches zuerst kommt, und i das Neuron, welches danach kommt (Reihenfolge im Forward-Pass), obwohl es bei W_{ij} andersherum steht.
C_0	Die Cost des Output Layers
ϵ	Die Lernrate (meist ein kleiner Wert wie 0.001)

4 Ergebnisse

Der ganze Code (mit englischen Kommentaren) kann unter https://github.com/AR102/JuFo2021 gefunden werden.

4.1 Teilbarkeit – Erster Testversuch

Zuerst haben die Formeln der Backpropagation in Julia als Funktionen eingebaut und eine Variable ϵ definiert, die für die Lernrate steht. Um das Programm zu testen, haben wir die temporäre Funktion `generateTrainingData(...)` implementiert, die die nötigen Trainingsdaten ausgibt, um ein neuronales Netzwerk darauf zu trainieren, die Teilbarkeit von Zahlen durch 2 zu überprüfen.

Unser erstes Netzwerk dafür hatte folgende Konfiguration:

Inputs	Hidden	Outputs
32	6	1

wobei die Inputs der Repräsentation einer ganzen Zahl in binär entsprechen und der Output 1 sein soll, wenn diese ganze Zahl durch 2 teilbar ist, 0 wenn nicht. Die Trainingsdaten waren die

Zahlen 1 bis 200, also gab es insgesamt 200 Trainingsdatensätze. Die Testdaten, mit denen die Genauigkeit des Netzwerkes am Ende getestet wurde, waren 2000 zufällige Zahlen im Bereich von 1 bis 2000.
ϵ (also die Lernrate) = 0.01

Tabelle 3: Beispielhafter Trainingsdatensatz

Zahl:	99
Inputs:	binäre Darstellung von 99 auf 32 Stellen, also [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1]
Outputs:	falls 99 teilbar durch 2 →[1], sonst →[0], also hier [0]

Die Backpropagation wurde 1000-mal durchgeführt.
Das Ergebnis:

Tabelle 4: Performance des Netzwerkes nach 1000-mal Backpropagation

Genauigkeit:	2000/2000 richtig →100%
Gesamt-Cost (durchschnitt der Costs für alle Trainingsdaten):	$C = 0.0011679187483701488$
Gesamt-Cost für Testdaten:	$C = 0.0017070257611734073$

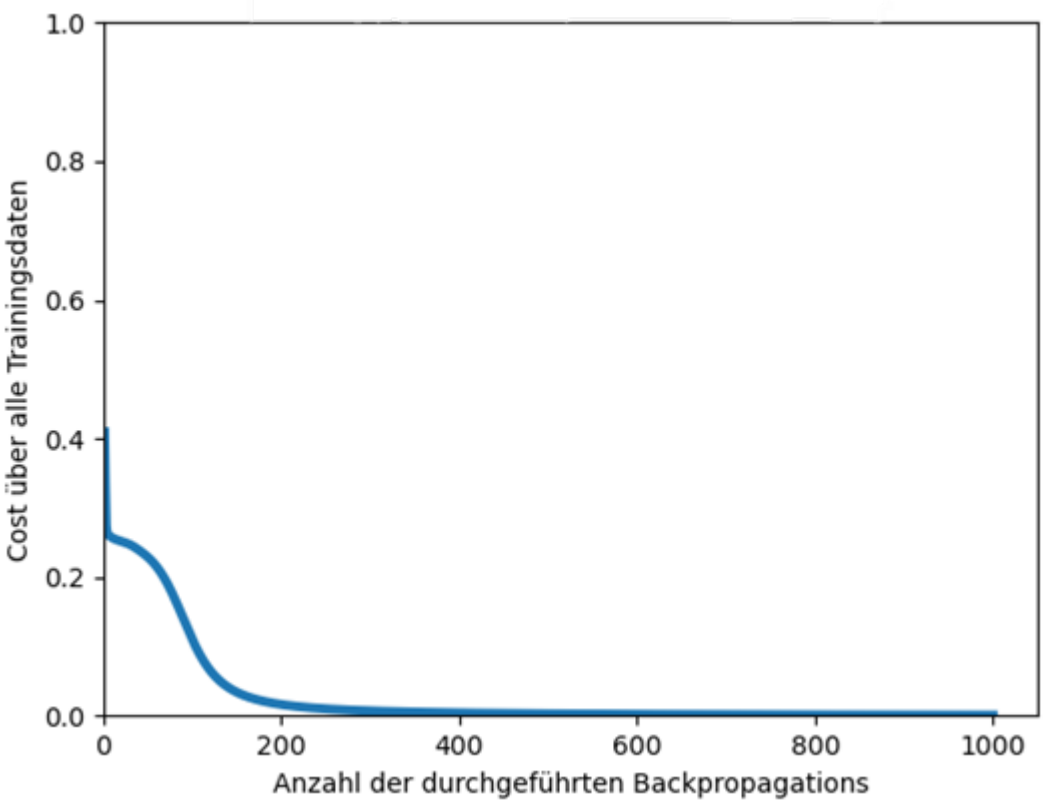

Abbildung 6: Plot der Cost-Entwicklung für die Trainingsdaten (neuer Wert nach jedem Durchlauf der Backpropagation)

4.2 Die Musik

Zuerst haben wir nach einem einfachen Lied gesucht, das eine klare Struktur hat und nicht oder kaum von dieser abweicht. Dabei sind wir auf das Präludium Nr. 1 aus dem Wohltemperierten Klavier von Johannes Sebastian Bach gestoßen.

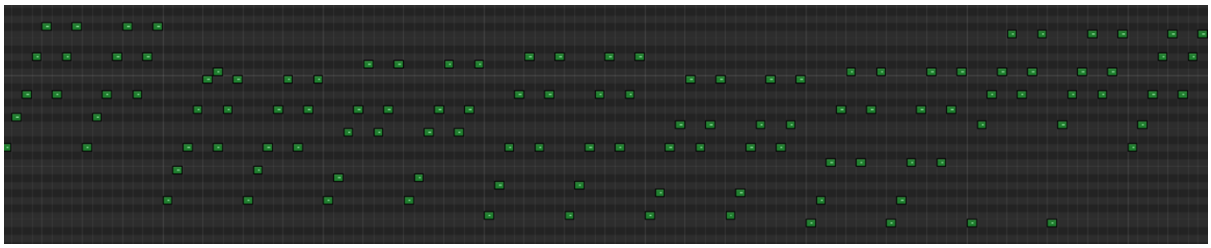

Abbildung 7: Darstellung des Originalliedes

Wie man in Abbildung 7 sehen kann, werden in diesem Stück fast nur gebrochene Akkorde gespielt. Dabei bleibt das Muster, mit dem die Akkorde gebrochen werden, immer gleich. Außerdem wir jeder Akkord einmal wiederholt. Die KI muss also erkennen, wie das Muster funktioniert, mit dem die Akkorde gebrochen werden. Zusätzlich muss die KI erkennen, welche Akkorde und Akkordreihenfolgen (Kadenzen) verwendet werden.

Abbildung 8: Darstellung des von der KI generierten Originalliedes

Man kann in Abbildung 8 erkennen, dass die KI das Muster gefunden hat, jedoch nur das grundlegende, wodurch das Ergebnis sehr repetitiv ist. Außerdem war sie nicht in der Lage, die Akkorde und Kadenzen des Stückes genau zu erkennen. Die Fehler sind zwar sehr klein, meist nur ein oder zwei Halbtöne daneben, führen aber dazu, dass das Lied für das menschliche Gehör dissonant klingt.

5 Diskussion

Nach einiger Überlegung haben wir mehrere Möglichkeiten gefunden, unser neuronales Netzwerk zu verbessern. Eine Möglichkeit wäre es, unseren Computer einfach länger rechnen zu lassen, vielleicht auch mehrere Computer zu benutzen oder ein schnelleres Programm zu programmieren, zum Beispiel, indem unklare Definitionen und unnötige Variablen vermieden werden. Außerdem ist es möglich, noch mehr Trainingsdaten zu generieren oder die vorhandenen Trainingsdaten effizienter zu gestalten, zum Beispiel durch das Berechnen der Anzahl an Noten (Inputs), die zur eindeutigen Vorhersage notwendig sind („Musikalisches Gedächtnis“).

Wir arbeiten im Moment auch an Unsupervised Learning. Unsupervised Learning ist besonders, da man damit eine Künstliche Intelligenz trainieren kann, ohne Label oder die richtigen Outputs zu den Input vorgeben zu müssen. Da er leicht mit der von uns bereits programmierten Struktur und Technik umsetzbar sein sollte, ist uns vor allem die Vorgehensweise mit einem Autoencoder ins Auge gefallen. Dieser funktioniert, indem ein Neuronales Netzwerk x viele Noten als Inputs bekommt, im Hidden Layer aber weniger Neuronen hat. Als Output soll das Netzwerk dann versuchen, etwas möglichst ähnliches zum Input auszugeben. Da das Neuronale Netzwerk aufgrund der beschränkten Anzahl an Neuronen im Hidden Layer jedoch nicht einfach bei jedem Lied eins zu eins die Noten übertragen kann, muss es sogenannte Features, vergleichbar mit Mustern, in den gegebenen Trainingsdaten, in unserem Fall Liedern, finden, und diese über den Hidden Layer weiterleiten. Dieses Vorgehen ist ähnlich zu dem des menschlichen Gehirns, das sich zum Beispiel beim ersten Ansehen eines Schachbretts merkt, dass es ein 8×8 Feld mit immer abwechselnd schwarz und weißen Feldern ist, und nicht jeden einzelnen „Pixel“, also Lichtpunkt. In unserem Fall gäbe es voraussichtlich Features, die zum Beispiel das Tempo, die Dichte an Noten, die Art der Tonleiter, u. Ä bestimmen, jedoch meist nicht intuitiv verständlich sind, da sie von der KI selbst bestimmt werden. Wenn man nun damit ein Lied generieren möchte, kann man alle

Layers vor diesem Hidden Layer weglassen und diesen zum Input Layer machen. So kann man nach Belieben die Werte der Features verändern und bekommt immer andere, (wahrscheinlich) komplett originale Lieder. [5]

Eine andere Lösung könnte auch eine andere Art von neuronalem Netzwerk sein – denn wir haben uns auf Deep Learning konzentriert, doch es gibt sehr viele andere Varianten. Außerdem gibt es die Möglichkeiten, mehr oder weniger Biases pro Layer zu benutzen, ϵ dynamisch anpassen zu lassen, und sogar die Struktur des Netzwerkes automatisch optimieren zu lassen. Man könnte auch unsupervised Learning probieren. Auch könnte es möglich sein, die Backpropagation so zu optimieren, dass sie z.B. vorausdenkt und so flache Plateaus verhindert werden. Der Zeitrahmen hat leider nicht gereicht, diese Dinge weiter zu recherchieren, jedoch wäre es eine Option für die Zukunft.

Wir hatten auch eine Idee, wie man auf unserem Projekt aufbauen könnte: Man könnte versuchen, Google Deep Dream nachzuprogrammieren, aber nicht auf Bilder, sondern auf Lieder anzuwenden. Aber wie funktioniert Deep Dream überhaupt?

Abbildung 9: Ein Beispiel für ein Deep Dream Bild [7]

Zuerst programmiert man ein neuronales Netzwerk, das bestimmte Bilder klassifizieren kann (in Abbildung 9 wahrscheinlich Vögel). Um Vögel zu erkennen, muss das neuronale Netzwerk bestimmte Muster erkennen, die in Bildern von Vögeln vorkommen. Danach gibt man dem neuronalen Netzwerk ein neues Bild als Input. Nun guckt man für alle Neuronen eines bestimmten Layers, ob sie eine hohe Aktivierung haben. Wenn die Aktivierung einen bestimmten Schwellenwert überschreitet, bedeutet das, dass das neuronale Netzwerk ein Muster erkennt, welches in Bildern von Vögeln gefunden werden kann. Anschließend kann man mit Hilfe der Backpropagation ausrechnen, wie man das Bild (den Input) verändern kann, damit die Aktivierung, die den Schwellenwert überschritten hat, noch größer wird. Somit wird das Bild „vogeliger“. Auf Lieder angewandt könnte man so zum Beispiel klassische Lieder „jazziger“ machen.

6 Quellen

6.1 Bücher

1 Andreas Zell:
Simulation Neuronaler Netze, Addison Wesley (Deutschland) GmbH, 1994
letzter Zugriff: 02.01.2021

6.2 Internetquellen

2 Grant Sanderson:
Neural Networks-Series
(https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi)
letzter Zugriff: 03.01.2021

3 Brotcrunsher:
Neuronale Netze – Backpropagation
(https://www.youtube.com/watch?v=YIqYBxpv53A,
https://www.youtube.com/watch?v=EAtQCut6Qno)
letzter Zugriff: 31.12.2020

4 Fabian Beck & Günter Daniel Rey:
Neuronale Netze: Eine Einführung – Backpropagation – Probleme
(http://www.neuronalesnetz.de/backpropagation5.html)
Letzter Zugriff: 04.01.2021
Neuronale Netze: Eine Einführung – Backpropagation – Lösungen
(http://www.neuronalesnetz.de/backpropagation6.html)
Letzter Zugriff: 04.01.2021

5 CodeParade:
Computer Generates Human Faces
https://www.youtube.com/watch?v=4VAkrUNLKSo
Letzter Zugriff: 27.02.2021
Generating Songs With Neural Networks (Neural Composer)
https://www.youtube.com/watch?v=UWxfnNXlVy8
Letzter Zugriff: 27.02.2021

6.3 Julia-Packages

- George Datseris und Joel Hobson:
MIDI, seit Aug 23, 2015
(https://github.com/JuliaMusic/MIDI.jl)

- Steven G. Johnson
PyPlot, seit Nov 18, 2012
(https://github.com/JuliaPy/PyPlot.jl)

- Simon Kornblith und Tim Holy:
JLD, seit Aug 23, 2015
(https://github.com/JuliaIO/JLD.jl)

6.4 Midi-Dateien für Trainings-/Testdaten

- https://bitmidi.com/

- https://www.midiworld.com/files/

6.5 Abbildungen

6 https://www.heise.de/select/ix/2017/9/1504455013673842/contentimages/avr.tut.ml-python-2.B1.jpg

7 https://deepdreamgenerator.com/#gallery →Unterkategorie „DEEP DREAM“

Wenn nicht anders angegeben, sind Abbildungen selbst erstellt.

7 Unterstützungsleistungen

Prof. Dr. Bernd Blasius
Hilfe beim / Ideen für das/die:

- Bugfixing (Beseitigung von Softwarefehlern)
- Speichern von Weights in multidimensionalen Arrays statt Dictionaries
  →Performance Verbesserung
  →Verkürzung des Programms
- Genauere Definition von Arrays, um den Datentyp Any zu vermeiden →Performance Verbesserung