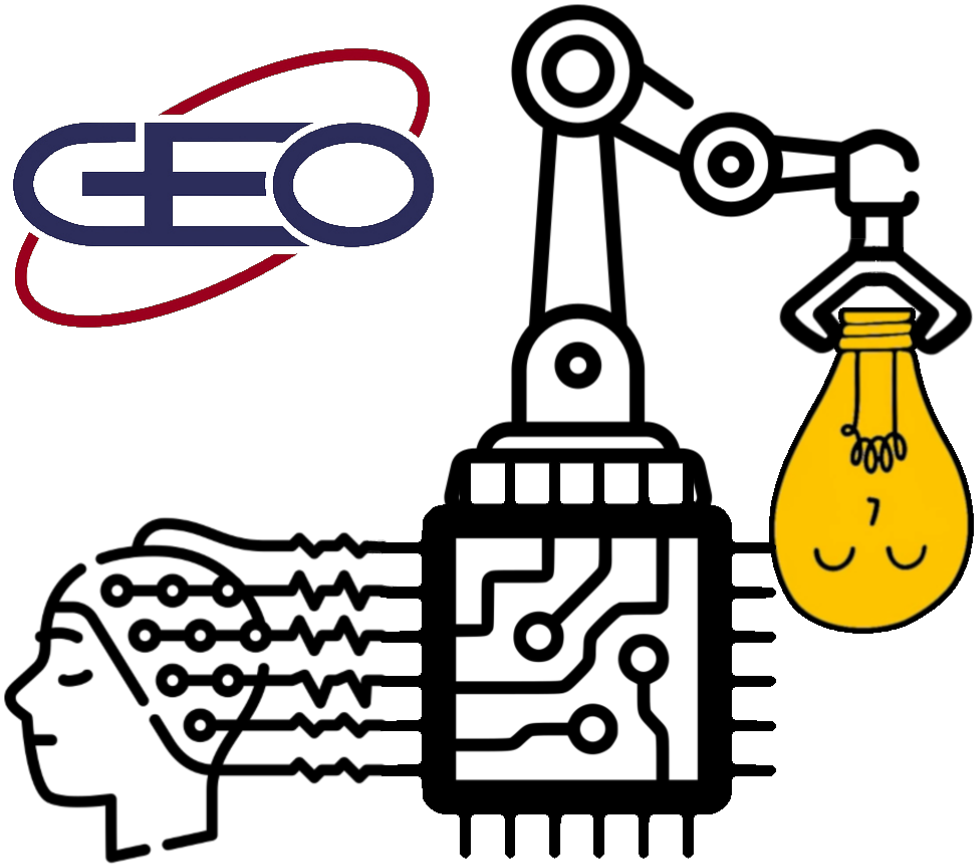
Entwicklung eines Frameworks und eigener Hardware für Brain-Computer-Interfaces

Sammlung, Verarbeitung und Analyse unserer Gehirnsignale



Alexander Reimer, Matteo Friedrich und Mattes Brinkmann

Gymnasium Eversten Oldenburg

Inhaltsverzeichnis

Kurzfassung

In diesem Projekt wollen wir ein Framework für ein Brain-Computer-Interface (BCI) verwirklichen. Dafür entwickeln wir unsere eigene, deutlich günstigere Alternative des benötigten Messgerätes (EEG), welches mithilfe von Elektroden Spannungsdifferenzen misst, die von Neuronen im Gehirn erzeugt werden. Gleichzeitig versuchen wir anhand einer schwierigen Aufgabe (Erkennung der Gedanken an verschiedene Richtungen) ein allgemein verwendbares Interface zu entwickeln, welches die Verarbeitung der Gehirndaten, die Erstellung eines neuronalen Netzes für die Analyse und die anschließende Verwendung des BCI übernehmen kann. Dieses Interface soll möglichst weit abstrahiert und vereinfacht sein, aber dennoch tiefgreifende Anpassungen leicht ermöglichen, sodass es hoffentlich Anfänger den Bereich näher bringen, aber auch in Zukunft eine gute Grundlage für erfahrene Entwickler bieten kann.

1 Einleitung

Ziel des Projektes ist es, ein Framework für BCIs – Brain-Computer-Interfaces – zu entwickeln, welches die Erstellung von BCIs vereinfachen soll, indem der Prozess stark abstrahiert wird. Dabei ist uns wichtig, dass unser Framework trotzdem durch viele Anpassungsmöglichkeiten auch fortgeschrittene Personen ansprechen kann.

BCIs sind Schnittstellen zwischen Gehirn und Computer und sollen Kommunikation zwischen diesen ermöglichen. Sie können für die bewusste Steuerung von Dingen wie Prothesen, Drohnen und Robotern verwendet werden, aber auch für diagnostische Zwecke, wie die Voraussage von Migränen oder die Erkennung von Gehirnstörungen. So wird zum Beispiel gerade daran gearbeitet, BCIs zu verwenden, um die Kommunikation von Menschen mit vollständigem Locked-in-Syndrom (CLIS) mit der Außenwelt zu ermöglichen, obwohl diese keinerlei bewusste Muskelkontrolle mehr haben. [**BCIChaudhary**] Außerdem lassen sich BCIs mit instrumentelle oder die klassische Konditionierung sogar zur Veränderung unserer Gehirnmuster und -aktivitäten verwenden. [**BCIChaudhary**]

Wir hoffen, bei der Erarbeitung unseres Projektes neben dem Erlangen von Erfahrung in diesem interessanten Bereich auch selbst dazu beizutragen. BCIs sind ein relativ neues Thema und Forschung in diesem Bereich wird, soweit wir gesehen haben, größtenteils durch Forschungsinstitut, Universitäten, etc. durchgeführt. Gründe dafür sind vermutlich die hohen Kosten eines EEGs, welches gute Daten liefert, sowie die Komplexität und der Aufwand. Letzteres liegt auch daran, dass Kenntnisse in verschiedenen Bereichen notwendig sind: Programmieren, Künstliche Intelligenz, Psychologie und Gehirn-Physiologie sowie die Technik und Physik hinter einem EEG.

Wir haben uns mit einigen dieser Bereiche beschäftigt und haben uns als Ziel gesetzt, diesen Prozess der Entwicklung eines BCI leichter und zugänglicher zu machen.

Zum einen wollten wir dazu ein eigenes Software-Framework entwickeln. Dieses sollte möglichst viele der oben genannten Aufgaben übernehmen und dem Nutzer ein möglichst abstrahiertes Interface bieten, welches z.B. die Aufbereitung der rohen Daten und die Entwicklung und das Trainieren einer künstlichen Intelligenz übernimmt, ohne dass sich der Nutzer erst detailliert mit diesen Themen auseinander setzen muss.

Gleichzeitig sollen aber Anpassungen und Erweiterungen durch den Nutzer leicht umsetzbar sein, da voraussichtlich nicht in jeder Situation die von uns gewählten Verfahren und Parameter passend sein werden.

Dies ist bereits das zweite Jahr, dass wir an BCIs forschen. Letztes Jahr hatten wir zwar Erfolg darin, ein BCI zu entwickeln, doch dieses hatte drei hauptsächliche Probleme:

1. Es konnte zwar unsere gewählte zu erkennende Aktivität (Schließen der Augen) erkennen, doch diese ist sehr simpel – die meisten anderen Verwendungszwecke eines BCI sind viel komplexer und wären vermutlich nicht möglich gewesen.
2. Es war nicht nutzerfreundlich, da es kaum dokumentiert war, nicht ohne Weiteres zur Erkennung beliebiger Aktivitäten verwendet oder anders angepasst werden konnte, und schwer in eigenen Programmen zu verwenden war.
3. Wir haben das Ganglion EEG von OpenBCI benutzt, welches ca. 500 € gekostet hat und somit zwar verglichen mit anderen EEGs günstig war, aber für uns noch zu teuer.

Probleme 1 und 2 werden durch das bereits beschriebene Framework abgedeckt; jetzt, da wir etwas mehr Erfahrung haben, achten wir mehr auf die Software-Gestaltung anstelle eines eher Proof-of-Concept artigen Programms.

Das dritte Problem, den Preis des EEG, wollen wir „lösen“ indem wir unser eigenes, günstigeres Selbstbau-EEG entwickeln.

2 Materialien

- Gekauftes EEG aus dem letzten Jahr
  - 4-Kanal Ganglion Board von OpenBCI ⋆
  - 4x Spike-Elektroden ⋆
  - Klettband für die Elektroden ⋆
  - 2x Ohrclips ⋆
  - Lithium-Polymer-Akku und Ladegerät ⋆
  - Plastik-Hülle für das Ganglion Board
- Selbstbau-EEG
  - Raspberry Pi 3B
  - Analog/Digital-Wandler MCP3208
  - Instrumentenverstärker AD620AN
  - 5x Operationsverstärker LM385
  - Diverse Widerstände
  - Keramik-/Tantal-/Elektrolytkondensatoren
  - Steckbrett
  - Kabel, Drähte, Krokodilklemmen
  - Netzteil/Batterie($\pm$ 9 V)
  - Oszilloskop
- Computer: Aorus 15P (Laptop)
  - CPU: i7-11800H
  - GPU: RTX 3060
  - RAM: 64 GB
- Julia als Programmiersprache für unser Projekt mit den Packages:
  - LSL.jl
  - FFTW.jl
  - BSON.jl
  - Flux.jl
  - BaremetalPi.jl
  - CSV.jl
  - PyPlot.jl
  - CUDA.jl
- OpenBCI GUI

3 Vorgehensweise

Für die Entwicklung unseres Framework und EEG haben wir uns ein Ziel gesetzt: Mit diesen ein BCI entwickeln, welches erkennen soll, ob die Testperson gerade an links, rechts oder nichts denkt. Grund dafür ist, dass wir durch ein konkretes Anwendungsbeispiel fortlaufend sehen können, wie Änderungen die Qualität und Nutzbarkeit des Programms und des EEG beeinflussen, indem wir die Leistung dieses spezifischen BCIs sowie Komplexität und Aufwand dieses Programms, welches unser Framework verwendet, betrachten.

Die Inspiration kam von Harrison „Sentdex“: Dieser hat auf GitHub einen frei verwendbaren Datensatz an EEG-Daten hochgeladen, die er mit genau diesem Ziel (Erkennung der Richtung) aufgenommen hat. [**Sentdex**]

Dieses Anwendungsbeispiel passt sehr gut, da es sehr schwer zu erkennen ist und somit viele Verbesserungen benötigt. Bei einem sehr einfachen Beispiel hätten wir bereits eine perfekte Genauigkeit von 100% erreicht und könnten somit nicht die Auswirkungen von weiteren Verbesserungen beobachten.

Außerdem haben wir zu diesem Beispiel durch Sentdex bereits EEG-Daten, wodurch wir bereits mit der Entwicklung der Software beginnen konnten, während wir gerade gleichzeitig das EEG entwickelten.

Da Sentdex selbst daran gearbeitet hat, können wir zusätzlich die Qualität unseres eigenen BCI mit seinem vergleichen und so sehen, wie die Leistung unseres allgemeinen Frameworks im Vergleich zu seinem spezifischen Programm ausfällt.

3.1 Elektroenzephalographie

Bei der Elektroenzephalographie (EEG) werden Elektroden an der Kopfoberfläche platziert. Diese können sehr kleine Spannungen messen, die durch Potentialänderungen in Neuronengruppen im Gehirn entstehen und durch den Schädel dringen. Eine Elektrode kann nur die Summe aller lokalen Potentialänderungen messen – sonst wäre eine Elektrode pro Neuron notwendig. Man kann also auch nur ungefähr sagen, wo genau im Gehirn eine Potentialänderung stattgefunden hat. Mit mehr Elektroden kann jedoch die örtliche Genauigkeit der Daten erhöht werden.

Bei der Messung der Spannung wird immer eine Differenz zwischen einer Elektrode und einem Referenzpunkt gebildet. Der Referenzpunkt wird zuvor invertiert.

Dabei gibt es zwei Verfahren: Bei der unipolaren Ableitung von Gehirnpotentialen wird ein neutraler Punkt wie ein Ohrläppchen, der von Gehirnströmen unbeeinflusst ist, gewählt. Bei jeder Elektrode wird dann die abgeleitete Spannung von dieser abgezogen. Bei diesem Verfahren haben also alle Elektroden den gleichen Bezugspunkt

Bei der bipolaren Ableitung hat jede Elektrode als Referenz den jeweiligen Nachbarn. Bei zwei Elektroden gibt es also nur eine Ausgabe, die Differenz der zwei. Bei drei Elektroden gibt es zwei Ausgaben, einmal die Differenz von Elektroden 1 und 2 und einmal die Differenz von Elektroden 2 und 3. [**EEGHausarbeit**]

Meistens wird die unipolare Ableitung gewählt, da bei der bipolaren Daten permanent verloren gehen: Zum einen ein Daten-Kanal und zum anderen kann es bei zwei entgegengesetzten EEG-Spannungen (z.B. Elektrode $1 = 7\,\mu V$, Elektrode $2 = -7\,\mu V$) zur Auslöschung des Signals kommen. [**EEGHausarbeit**]

In allen von uns verwendeten EEG-Daten (sowohl extern als auch selbst erzeugt) handelt es sich um unipolar abgeleitete Potentiale.

Wir untersuchen ereigniskorrelierte Potentiale (EKPs). Diese sind bestimmte Spannungsschwankungen („Potentiale“), welche in Zusammenhang mit einem internen oder externen Ereignis stehen, wie z.B. einem lauten Ton, einer Körperbewegung oder hoher Konzentration.

3.1.1 Bau unseres eigenen EEG

Die Spannungen, die von den Neuronen durch den Schädel dringen, sind zu gering (unterhalb von einigen hundert Mikrovolt), um direkt von einer Messeinheit verarbeitet werden zu können, es muss zunächst eine Verstärkung des aufzuzeichnenden Signals erfolgen. [**EEGHausarbeit**]

Ebenfalls müssen die gewünschten Frequenzen vorab mittels aktiver Filter herausgefiltert werden, um später eine genaue Signalanalyse zu ermöglichen. Je nachdem, was für Phänomene genau untersucht werden sollen, können entsprechend unterschiedliche Frequenzbereiche herausgefiltert werden. Beispielsweise liegen Alpha-Wellen, die ein bestimmtes Signalschema darstellen und eine wichtige Rolle bei der Konzentration sowie visuellen Verarbeitung spielen, in einem Frequenzbereich von 8 bis 13 Hz. [**Praktikum**, **Birbaumer2010**, **wiki:Berger-Effekt**]

Darüber hinaus wird eine Einheit benötigt, die das Ausgangssignal misst und aufzeichnet, damit es weiterverarbeitet und analysiert werden kann.

Sowohl für die Filterung als auch für die Verstärkung werden sogenannte Operationsverstärker (OPV) verwendet. In Abb. **??** sieht man rechts das Schaltzeichen mit den jeweiligen Anschlüssen und links einen OPV in einem DIL-Gehäuse.

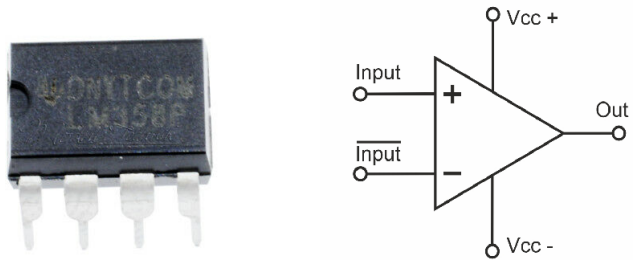


Abbildung 1: Links: 2-fach OPV LM358 in einem DIL-Gehäuse, Rechts: OPV als Schaltbild

OPVs finden vor allem in der Mess- und Regeltechnik Anwendung und bestehen aus Transistoren, die zu integrierten Schaltkreisen zusammengefügt werden. Dabei lässt sich die interne Schaltung eines OPVs allgemein in zwei Schaltungensteile untergliedern, den sog. Differenzverstärker, der später für die Potentialmessung wichtig wird, und die dahinter geschaltete Gegentaktendstufe, welche für die Leistungsverstärkung zuständig ist. Ihr geringer Ausgangswiderstand sowie der hohe Eingangswiderstand als auch der hohe Spannungsverstärkungsfaktor, machen den OPV passend für die Anwendung als EEG-Verstärker.

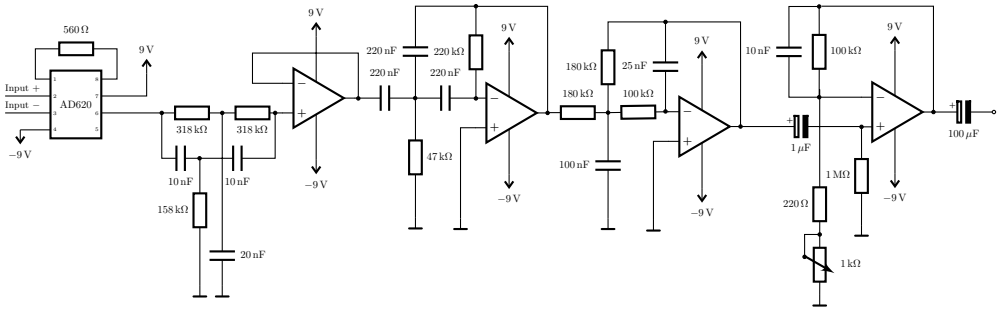


Abbildung 2: Schaltplan

OPVs besitzen, neben den Anschlüssen für die Stromversorgung, drei für ihre Funktion wichtige Anschlüsse, einen invertierenden und einen nicht-invertierenden Eingang sowie einen Ausgang, über welche er angesteuert werden kann.

Legt man nun über beiden Eingängen jeweils unterschiedliche Spannungen U_1 und U_2 an, so wird am Ausgang des OPVs die Differenz der beiden Spannungen ausgegeben. Also gilt: $U_2 - U_1 = U_{\text{Diff}}$. Dies macht man sich nun zu Nutze, um die Spannungen bzw. Potentialdifferenzen zwischen zwei Elektroden zu messen.

Zudem lassen sich mit einem OPV und wenigen weiteren Bauteilen einfach aktive Filter bauen.

Die Grundschaltung für unsere EEG beruht auf einem englischsprachigem Internetartikel, in dem eine Verstärkerschaltung für ein EEG beschrieben war. Die von uns angepasste Schaltung ist in Abb. x zu sehen. Diese mussten wir aber etwas verändern, um den gewünschten Frequenzbereich herauszufiltern. Basis dieser Schaltung bildet der Instrumentenverstärker AD620AN, welcher extra für EEG- und EKG-Anwendungen entwickelt wurde.

3.2 Fourier-Analyse

Die Fourier-Analyse kann die verschiedenen zugrundeliegenden Frequenzen von Datenfolgen, Funktionen, und mehr bestimmen, indem diese in Sinus-Kurven mit verschiedenen Frequenzen zerlegt werden, welche summiert möglichst nah am Ursprung liegen. Dafür kriegt jede Frequenz eine Amplitude zugeordnet. Die Fourier-Analyse dient also zur Spektralanalyse.

Wir nutzen in unserem Projekt die Fast Fourier Transformation (FFT), welche lediglich eine komplexere aber effizientere Form der Diskreten Fourier Transformation (DFT) ist.

Aus der FFT folgt ein Array (eine Liste) an komplexen Zahlen. Der Index eines Wertes in der Liste bestimmt, für welche Frequenz der Wert gilt (erster Wert: 1 Hertz, zweiter Wert: 2 Hertz, etc.). Nun muss für jede komplexe Zahl der Abstand zum Ursprung bestimmt werden, also der absolute Betrag. Dieser entspricht dann der Amplitude der Frequenz. So lässt sich bestimmen, welche Frequenzen am stärksten vorkommen. Außerdem können dann Frequenzen herausgefiltert werden, indem die Werte bei den entsprechenden Indices auf 0 gesetzt werden.

Eine FFT kann Elektroenzephalogramme fast verlustfrei repräsentieren. Dies lässt sich erkennen, wenn man mithilfe der durch die FFT entstandenen Spektralanalyse die Daten rekonstruiert (Inverse Fast Fourier Transformation, IFFT). Es werden dazu die Sinus-Kurven der Frequenzen mit den entsprechenden Amplituden multipliziert und dann addiert.

Wie in Abb. **??** sichtbar, ist diese Rekonstruktion kaum von den originalen Daten unterscheidbar.

3.3 Neuronales Netz

Ein neuronales Netzwerk besteht aus drei Teilen: der Eingabeschicht, den verdeckten Schichten und der Ausgabeschicht.

Die Eingabeschicht ist eine Liste aus Zahlen. Sie gibt an, welche Eingaben das Netzwerk bekommen soll, z.B. die Helligkeit der Pixel eines Graustufen-Bildes.

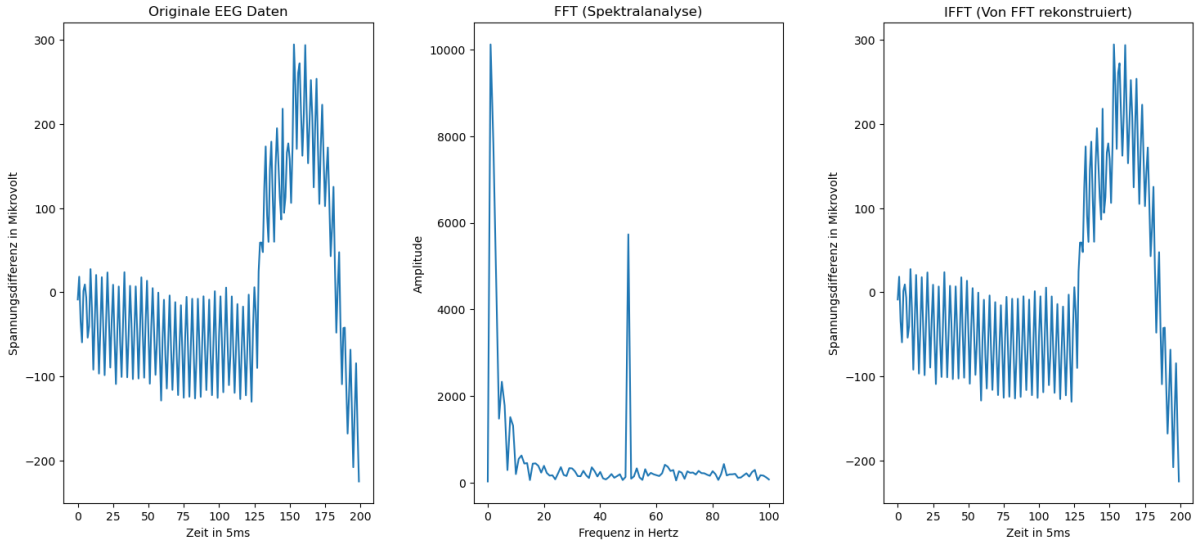


Abbildung 3: Links: Originale EEG Daten, Mitte: Die Amplituden der Frequenzen dieser Daten, Rechts: Die Umkehrung der Frequenzdaten. Da der rechte Graph dem linken sehr ähnlich ist, lässt sich erkennen, dass es bei der Fourier Analyse keinen großen Informationsverlust gibt.

Die verdeckten Schichten sind eine Ansammlung von in mehrere Schichten unterteilten Neuronen.

Jedes Neuron besitzt eine sog. Aktivierung, die meist als Zahl zwischen 0 und 1 oder -1 und 1 angegeben werden kann. Die Neuronen verschiedener Schichten sind durch sogenannte Gewichte (*weights*) verbunden, die ebenfalls einen beliebigen Wert haben können. Die Ausgaben / Vorhersagen des neuronalen Netzwerkes werden von den Aktivierungen der Neuronen in der Ausgabeschicht abgelesen.

Bei neuronalen Netzwerken gibt es zwei relevant Verfahren: Den Forward Pass und die Backpropagation.

Ersteres beschreibt das propagieren von den Eingabewerten durch das Netzwerk. Zuerst werden die Aktivierungen der Neuronen in der Eingabeschicht gesetzt und danach die Aktivierungen der Neuronen der nächsten Schicht berechnet, welche dann wiederum für der nächsten Schicht verwendet werden und so weiter. Dadurch werden die Eingaben von Schicht zu Schicht bis zur Ausgabe weiterverarbeitet. Der Forward Pass ist zur Verwendung eines Netzwerks notwendig.

Er bietet außerdem den ersten Schritt der Backpropagation, die für das Lernen des Netzwerks verantwortlich ist. Nachdem alle Aktivierungen gesetzt wurden, werden die Aktivierung der letzten Schicht, also die Ausgaben, bewertet. Wir verwenden Supervised Learning, heißt wir nutzen zur Bewertung Trainingsdaten, die wir vorher *gelabelt*, ihnen also die korrekten Ausgabedaten zugewiesen haben.

Dann wird jedes einzelne Ausgabeneuron „bewertet“, indem die Differenz zwischen der tatsächlichen Aktivierung nach dem Forward Pass und den zugewiesenen korrekten Ausgaben gebildet wird.

Diese Differenz zwischen ist-Zustand und soll-Zustand wird nun in der Backpropagation verwendet, um die Gewichte zwischen Ausgabe- und vorletzter Schicht anzupassen, denn mit ihm kann der sogenannte delta-Wert eines Neurons bestimmt werden. Dieser entspricht der Änderung der Aktivierung eines Neurons, die notwendig wäre, damit ein Forward Pass von dort aus ein korrektes Ergebnis liefert.

Auf die Berechnung des delta-Werts werden wir dieses Jahr nicht weiter eingehen, da es für uns nicht direkt relevant ist. Interessant ist womöglich nur noch, dass dafür die delta-Werte der nächsten Schicht (in Richtung Ausgabe) benötigt werden und daher der *backward* (dt. rückwärts) Teil im Namen kommt.

Mithilfe dieses delta-Werts kann dann auch bestimmt werden, wie die Gewichte angepasst werden müssten, um diese gewünschte Änderung umzusetzen:

$$\Delta W_{i,j} = \eta * \delta_i * a_j$$

Und entsprechend angepasst werden mit:

$$W_{i,j} = W_{i,j} + \Delta W_{i,j}$$

wobei i das Neuron näher zur Ausgabe, j das Neuron näher zur Eingabe und $W_{i,j}$ das Gewicht zwischen diesen beiden ist.

Die Multiplikation mit a_j findet statt, um zu ermöglichen, dass die Änderung der Gewichte entsprechend ihres tatsächlichen Einflusses geschieht, bedeutet, wenn ein Gewicht mit einem Neuron verbunden ist, welches kaum aktiv ist (eine niedrige Aktivierung hat), wird das Gewicht zu diesem auch kaum angepasst, und wenn es negativ ist, dann wird die Änderung umgekehrt, was auch notwendig ist, um das Ziel zu erreichen.

η ist die sogenannte Lernrate. Sie ist meist ein sehr niedriger Wert (0.01, 0.0001, etc.) und ist wichtig für das Kernkonzept des Lernen eines neuronalen Netzwerks: Sie sorgt dafür, dass bei jedem Trainingsdatensatz die Gewicht nur etwas an diesen angepasst werden, sodass das gesamte Training die Gewichtsanpassungen aller Trainingsdatensätze und somit hoffentlich auch eine Generalisierung dieser darstellt.

Insgesamt lässt sich also sagen, dass ein neuronales Netzwerk versucht, diese Differenz ist − soll zu minimieren. Diese Differenz ist also ähnlich einer Verlustfunktion und lässt sich tatsächlich auch in einer tatsächlichen Verlustfunktion für Supervised trainierte neuronale Netzwerke, dem Mean Squared Error, wiederfinden:

$$C_0 = (a_0 - y)^2$$

wobei C_0 der Verlust der Ausgabeschicht (Schicht 0, da Indexierung der Schichten bei der Ausgabe beginnt), a_0 der Vektor aller Aktivierungen der Ausgabeschicht, und y ein Vektor der richtigen Ausgaben für die Eingaben, mit denen die Aktivierungen berechnet wurden, ist.

Es gibt verschiedene Arten von Schichten in neuronalen Netzwerken; wir verwenden hauptsächlich Fully Connected und Convolutional Schichten.

3.3.1 Fully Connected Schichten

Bei Fully Connected Schichten sind alle Neuronen eines Layers mit allen Neuronen des nächsten Layers verbunden. Außerdem hat typischerweise jedes Neuron einen sogenannten Bias, der hinzu addiert wird.

Zur Berechnung der Aktivierungen der Neuronen gibt es den sogenannten Forward Pass. Dabei beginnt man in der ersten verdeckten Schicht damit, für alle Neuronen die sogenannte Netzeingabe zu berechnen. Um die Netzeingabe eines Neurons zu berechnen, werden alle Aktivierungen der vorherigen Schicht mit den von dem Neuron dorthin führenden Gewichten multipliziert und aufsummiert.

Der Bias ist eigentlich auch ein Gewicht, jedoch ist er mit einem Neuron verbunden, das immer die Aktivierung 1 hat. [**brotcrunsher:forwardpass**]

Die Formel für die Berechnung der Netzeingabe eines Neurons in einer Fully Connected Schicht lautet also:

$$\text{net}_j = \sum_L (a_L * W_{Lj}) + b_j$$

wobei net_j die Netzeingabe des Neurons j, L die vorherige Schicht, a_L der Vektor aller Aktivierungen der Schicht L, W_{Lj} der Vektor aller Gewichte zwischen dem Neuron j und den Neuronen der Schicht L, und b_j der Bias des Neurons j ist.

Um aus dieser Netzeingabe nun die Aktivierung zu berechnen, benötigt man eine Aktivierungsfunktion, die zum einen dafür sorgt, dass die Aktivierung begrenzt und somit ein unendliches Wachstum der Aktivierungen und entsprechend Gewichte verhindert wird, und zum anderen sicherstellt, dass es keine Linearität zwischen Eingaben und Ausgaben gibt. Dadurch kann das Netzwerk vor allem Klassifizierungsaufgaben besser erlernen.

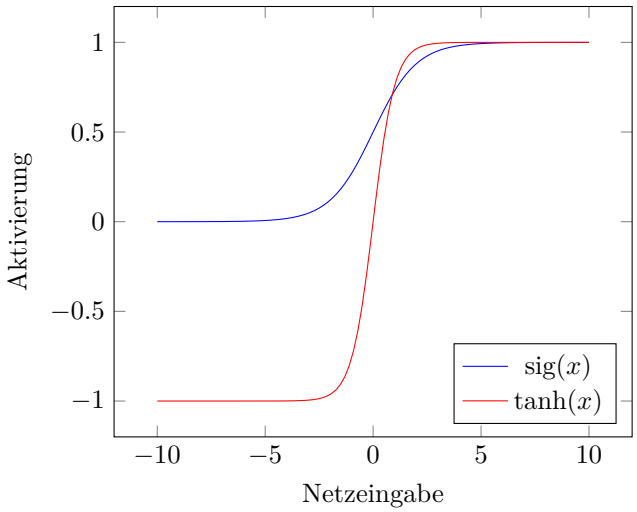


Abbildung 4: Aktivierungsfunktionen im Vergleich

Letztes Jahr haben wir die Sigmoid-Funktion genutzt, welche sicherstellt, dass der Eingabewert immer zwischen 0 und 1 liegt (s. Abb. **??**.

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

Doch dieses Jahr setzen wir auf die tanh-Aktivierungsfunktion, bei der die Werte immer zwischen -1 und 1 liegen (s. Abb. **??**. Mit der Sigmoid-Aktivierungsfunktion war aufgrund der Beschränkung auf positive Zahlen das Abziehen von der Aktivierung eines Neurons durch ein vorhergehendes nicht möglich und somit die Komplexität und Möglichkeiten eines neuronalen Netzwerkes eingeschränkt.

Dieses Verfahren zur Berechnung der Aktivierung wird dann für jedes Neuron in jeder Schicht wiederholt. Da aber immer die Aktivierungen der vorherigen Schicht (a_L) benötigt, muss dieser Prozess von der Eingabeschicht aus zur Ausgabeschicht hin durchgeführt werden, daher auch der Name Forward Pass.

Fully Connected Schichten sind die „Standard"-Schicht bei neuronalen Netzwerken. Sie können und werden vielseitig eingesetzt, doch haben den Nachteil, dass sie „global" sind, d.h. sie

3.3.2 Convolutional Layer

Im Gegensatz zu den den fully-connected Schichten, in denen jedes Neuron einer Schicht mit allen Neuronen der nächsten Schicht verbunden ist, sind die Neuronen der Convolutional Layer jeweils nur mit ein paar Neuronen der nächsten Schicht verbunden. Convolutional Layer nutzen einen sogenannten Filterkernel, um die Aktivierungen der Neuronen der nächsten Schicht zu bestimmen. Der Filterkernel ist eine Liste an Zahlen, hat immer genauso viele Dimensionen wie die Eingaben und ist generell kleiner als sie.

Um die Werte der nächsten Schicht zu bestimmen, „wandert" der Filterkernel über die Eingaben, sodass er jede mögliche Position einmal einnimmt. Bei jedem Schritt wird der Filterkernel „angwendet", indem das Skalarprodukt des aktuellen Ausschnittes der Eingaben und des Filters berechnet wird. Um das Skalarprodukt zu berechnen, muss das erste Element des Filterkernels mit dem ersten Element des Eingabenausschnitts, das zweite Element des Flterkernels mit dem zweiten Element des Eingabenausschnittes usw. multipliziert werden und anschließend die Summe von all diesen ausgerechnet werden. Diese Summen bilden dann die Werte für die nächste Schicht des neuronalen Netzwerkes.

Beim Trainieren des neuronalen Netzwerkes werden bei Convolutional Layern dann die einzelnen Werte der Kernel als anzupassende Gewichte behandelt.

Der Vorteil von Convolutional Layern ist, dass sie lokal und nicht global arbeiten. Damit ist gemeint, dass eine normale Fully Connected Schicht ein erwartete Muster nur erkennt, wenn es an seiner Position so auch

schon in den Trainingsdaten vorkam. Da der Kernel jedoch über die Eingaben wandert und alle möglichen Positionen annimmt, kann ein trainierter Kernel das antrainierte Muster überall erkennen, egal, wo es vorkommt. Deswegen werden Convolutional Schichten sehr gerne bei Bild- und Videoverarbeitung genutzt, wo z.B. zu erkennende Hunde nicht immer auf der gleichen Stelle im Bild sind.

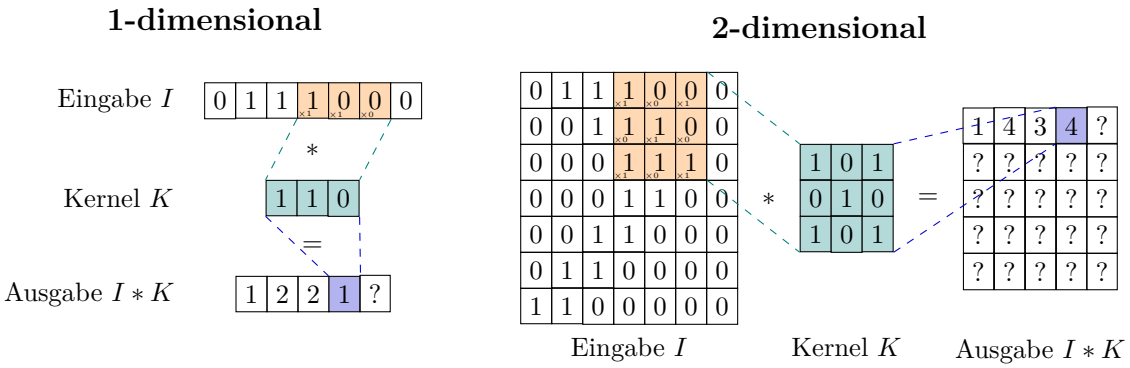


Abbildung 5: Eine 1-dimensionale und eine 2-dimensionale Convolutional Schicht. Die farbig hervorgehobenen Felder stellen einen Schritt des Kernels dar; als nächstes würde er ein Feld weiter nach rechts rücken und die Ausgabe auch.

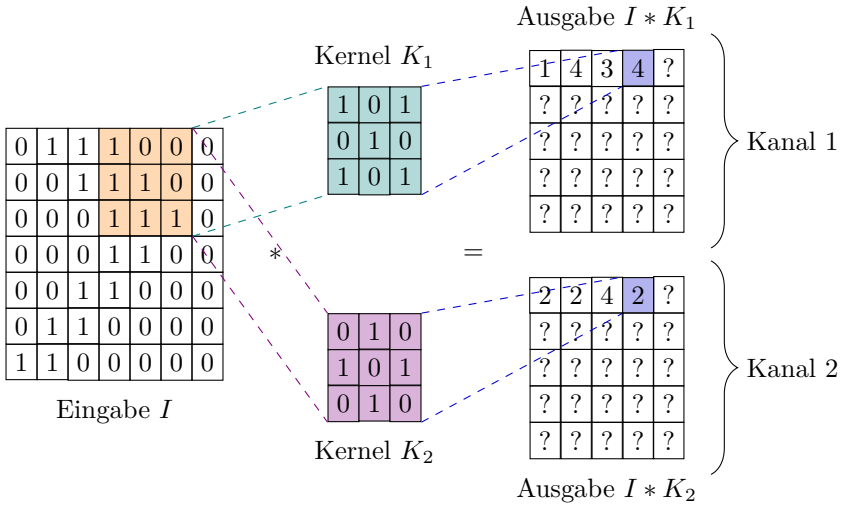


Abbildung 6: Eine 2-dimensionale Convolutional Schicht mit zwei Kanälen und entsprechend zwei Kerneln (K_1 und K_2) und zwei Ausgaben ($I * K_1$ und $I * K_2$). Es kann theoretisch beliebig viele Kanäle geben.

Convolutional Schichten haben außerdem meistens mehrere Kanäle. Bei mehreren Kanälen hat die Schicht mehrere verschiedene Filterkernel, die alle auf die gleichen Eingabewerte angewendet werden. Da jeder Kernel jeweils eine Ausgabe hat, hat die Schicht auch mehrere Ausgaben.

Die Operation ist äquivalent dazu, bei einer n-dimensionalen tatsächliche Eingabe einen $n+1$-dimensionalen Kernel auf eine $n+1$-dimensionale Eingabe anzuwenden. Dabei muss die Eingabe entlang der $n+1$-ten Dimension kopiert werden. Die $n+1$-te Dimension des Kernels und der Ausgabe entspricht dem Kanal.

Mehrere Kanäle können verwendet werden, um mehrere Features gleichzeitig zu extrahieren, z.B. alle Kreise, alle Rechteckt, alle Linien, alle roten Bereiche, etc., wobei für jedes Feature ein Kanal verwendet wird.

Da diese Schicht dann mehrere

Loss

Um zu bestimmen, wie gut ein neuronales Netz ist, gibt es die sogenannte Verlustfunktion (engl. *loss function*), die die Datensätze verwendet, um die Abweichung des Netzwerks vom Ideal zu bestimmen. Diese Abweichung wird auch *loss* genannt.

Trainingsdaten bestehen aus einer Liste aus Trainingsdatensätzen. Jeder dieser Datensätze beinhaltet Eingaben für das Netzwerk und die richtigen Ausgaben dafür. Machine Learning, das mit solchen Trainingsdaten arbeitet, wird Supervised Learning genannt.

Die Funktion für den *loss* der Ausgabeschicht und somit des gesamten Netzwerkes (für einen Datensatz) lautet wie folgt:

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

wobei C_0 der *loss* der Ausgabeschicht, L die letzte Schicht (Ausgabeschicht), a_L der Vektor aller Aktivierungen der Schicht L, und y ein Vektor der richtigen Ausgaben für die Eingaben, mit denen die Aktivierungen berechnet wurden, ist.

Um den *loss* zu berechnen, muss man also für alle Neuronen der Ausgabeschicht die Differenz der durchs Netzwerk gegebenen und der in den Datensätzen vorgegebenen Aktivierungen bilden. Danach muss man diese Differenzen quadrieren und am Ende alle Ergebnisse aufsummieren. Dies kann man für alle Testdatensätze wiederholen und von allen *losses* den Durchschnitt nehmen, um die allgemeine Performance eines Netzwerkes zu überprüfen. Diese Art der Verlustfunktion wird mittlere quadratische Abweichung (engl. *mean squared error*, kurz MSE) genannt.

Zusammenfassend kann man also sagen, dass der *loss* die Abweichung von den aktuellen Ausgaben des Netzwerkes zu den idealen Ausgaben des Netzwerkes darstellt – ein perfektes neuronales Netz hätte einen *loss* von 0. Aufgrund der Trainingsdatensätze und der Verlustfunktion weiß man nun, wie stark die Ausgabeschicht abweicht und entsprechend verändert werden muss.

Backpropagation

3.3.3 Overfitting

Unterschiede zwischen Testdaten- und Trainingsdaten-Genauigkeit sind ein bekanntes Problem bei neuronalen Netzwerken und werden durch sogenanntes „Overfitting“ verursacht. Dabei passt sich das Netzwerk zu stark an die Trainingsdaten an, wodurch keine Muster in den Daten gefunden werden können und das neuronale Netzwerk nicht in der Lage ist, neue Daten korrekt zu klassifizieren, da es die Trainingsdaten einfach nur „auswendig gelernt“ hat.

Oft lässt sich das Problem darauf zurückführen, dass das Erlernen der Generalisierung der Datenerkennung eines neuronalen Netzwerkes darauf basiert, dass es die Trainingsdaten nicht einfach nur Abspeichern und wieder Aufrufen kann, da dies zwar die Kosten minimieren würde, es aber dafür nicht genug „Speicherplatz“ (Komplexität) hat. Also kann Overfitting durch eine zu große Netzwerkstruktur oder zu wenige Daten verursacht werden.

Außerdem kann ein schlechtes Signal-Rausch-Verhältnis die Ursache sein: Das Signal-Rausch-Verhältnis gibt an, wie stark sich ein Nutzsignal von dem Hintergrundrauschen abhebt. Gerade bei der Datenermittlung mit einem EEG ist dieses Verhältnis sehr schlecht, da die für uns relevanten Signale aus dem Gehirn durch den Schädel und Haut stark abgeschwächt werden, bevor sie bei den Elektroden ankommen. Durch großes Hintergrundrauschen ist es für ein neuronales Netz schwer oder sogar unmöglich, grundlegende Muster in den Daten zu finden, da diese womöglich nicht einmal mehr existieren, sodass es durch fehlende Komplexität zwar keine 100& Genauigkeit durch das direktere Abspeichern der Trainingsdaten erreichen kann, aber die Generalisierung noch schlechter wäre. Dadurch tendiert das Modell bei der Kostenoptimierung der Neuronen zum Overfitting.

Um Overfitting entgegenzuwirken, gibt es verschiedene Möglichkeiten. Diese gehören zu den Verfahren der Regularisierung, die ein Modell Bei der Verwendung von Dropout (dt. *fallen lassen*) wird einer Schicht des neuronalen Netzwerkes eine Dropout-Wahrscheinlichkeit zugeordnet. Bei jedem Trainingsschritt des Modells wird dann für jedes einzelne Neuron dieser Schicht basierend auf der Dropout-Wahrscheinlichkeit zufällig entschieden, ob es seinen Wert beibehalten oder auf 0 gesetzt wird. Bei einer Dropout-Wahrscheinlichkeit von 30% werden also im Schnitt 30% der Neuronen auf 0 gesetzt und somit ignoriert werden. Dies führt dazu, dass das neuronale Netzwerk die Daten nicht so einfach auswendig lernen kann.

Außerdem gibt es die L1- und L2-Regularisierungen. Die Idee hinter ist ähnlich zum Dropout, dass sie verhindert, dass das Netzwerk zu komplex wird und sich zu sehr an die Trainingsdaten anpasst. Im Gegensatz zu Dropout wird bei der L2-Regularisierung eine Penalties-Funktion zum Verlust hinzugefügt, die das Ausmaß der Gewichte des Modells beschränkt. Stattdessen wird es gezwungen, eine gewisse Einfachheit beizubehalten, was dazu beitragen kann, dass das Netzwerk besser auf neue, noch nicht gesehene Daten generalisiert.

Die Verwendung von Rauschen (*noise*) auf dem Input für ein Neuronales Netzwerk kann dazu beitragen, das Overfitting zu vermeiden und die allgemeine Robustheit des Modells zu erhöhen. Die Idee hinter der Verwendung von Rauschen auf dem Input ist, dass das Netzwerk gezwungen wird, sich an zusätzliche Unschärfen in den Eingaben zu gewöhnen und somit besser auf neue, noch nicht gesehene Daten generalisieren kann. Durch die Erhöhung der Robustheit des Modells gegenüber kleinen Änderungen der Eingaben kann es auch weniger empfindlich auf kleine Störungen oder Verzerrungen in den Daten reagieren.

4 Ergebnisse

Der gesamte Quellcode unseres Projektes sowie alle gesammelten Daten lassen sich auf unserem GitHub Repository [**InterpretingEEG**] finden.

Wir werden in diesem Bericht nur einige Design-Konzepte unseres Programms erklären; Informationen über die Verwendung sowie Details lassen sich in unserer Software-Dokumentation [**BCIInterfaceDocs**] finden.

4.1 Nutzerfreundlichkeit

Im Gegensatz zu letztem Jahr, in dem unser Programm eher die Funktion eines Proof-of-Concept erfüllte, haben wir dieses Jahr den Fokus viel stärker auf Nutzbarkeit und Anpassbarkeit unseres Programms gelegt.

4.1.1 Verwaltung

Letztes Jahr war es für die Verwendung unseres Programms notwendig, die einzelnen Dateien herunterzuladen und direkt zu inkludieren. Es gabe keine Versionskontrolle, keine einfache Methoden zum Aktualisieren und die Verwaltung der Abhängigkeiten des Programms hat ebenfalls nicht automatisch funktioniert.

Aber dieses Jahr haben wir das Framework als tatsächliches Julia-Paket entwickelt, welches über die normale Paketverwaltung verwaltet werden kann, und haben dadurch diese Probleme gelöst (s. Programmausschnitt **??**). Alle von dem Paket benötigten Abhängigkeiten werden automatisch installiert und mitverwaltet.

```julia
using Pkg
# Paket installieren
Pkg.add("https://github.com/AR102/Interpreting-EEG-with-AI")
# Paket aktualisieren
Pkg.update("BCIInterface")
# Paket in einem Programm verwenden
using BCIInterface
```

Programmausschnitt 1: Verwaltung unseres Pakets in einem Skript; die Verwendung der Julia REPL ist ebenfalls möglich.

4.1.2 Erweiter– und Anpassbarkeit

Wir haben beim Design des Frameworks starken Gebrauch von der Multimethoden-Unterstützung von Julia gemacht. Dabei können mehrere Funktionen den gleichen Namen haben. Welche Funktion ausgeführt wird, ist abhängig von der Anzahl und den Typen der Argumente. Für ein Beispiel siehe Programmausschnitt **??**.

```julia
# eigenen Datentypen namens "Dog" definieren
struct Dog end
# eigenen Datentypen namen "Cat" definieren
struct Cat end

# Funktion wird ausgeführt, wenn Parameter vom Typ "Dog" ist
makenoise(animal::Dog) = println("Wuff!")
# Funktion wird ausgeführt, wenn Parameter vom Typ "Cat" ist
makenoise(animal::Cat) = println("Meoow!")
```

Programmausschnitt 2: Multiple-Dispatch-Beispiel

Da wir unser Programm modular aufgebaut haben, ist es durch Multimethoden leicht möglich, einzelne Aspekte anzupassen, indem man eigene Typen definiert und dann die intern verwendeten Funktionen, an die diese Typen weitergereicht werden, „überlädt“ (neue Multimethoden definiert). Dies ermöglicht eine leichte Erweiterung und Anpassbarkeit unseres Frameworks.

Ein Beispiel davon ist in Programmausschnitt **??** zu sehen, wo ein eigenes EEG Gerät verwendet wird. In diesem Beispiel gibt dieses „EEG“ nur zufällige Werte aus, aber der Nutzer könnte in der Funktion `get_voltage` etwas beliebiges schreiben, also auch sich mit dem Internet verbinden, USB-Ports ansteuern, auf einen Netzwerk-Stream zugreifen, etc.

Ein großer Vorteil dieser Vorgehensweise ist, dass man als Nutzer weder auf Funktionen des Pakets verzichten / diese selbst implementieren muss noch den Quellcode unseres Pakets anpassen muss, was aufwändig sein und zu Problemen mit Paketaktualisierungen sowie Übersicht führen könnte.

```julia
using BCIInterface
# eigenes EEG Gerät als eigenen Datentypen definieren
struct MyBoard <: BCIInterface.EEG.EEGBoard
    # data_descriptor beschreibt Form der Daten
    # (Anzahl Elektroden? rohe Daten? Frequenzen?)
    data_descriptor::DataDescriptor
end
NUM_CHANNELS = 8
MyBoard() = MyBoard(RawDataDescriptor(NUM_CHANNELS))
# Überlade die interne Funktion zum Abrufen von Daten
function BCIInterface.EEG.get_voltage(board::MyBoard,
↪   channel::Int)
    # gib zufällige Zahl zwischen 0 und 1 zurück
    return rand()
end

device = Device(MyBoard(NUM_CHANNELS)
...
```

Programmausschnitt 3: Beispiel für die Verwendung eines eigenen EEGs

4.1.3 Weiteres

Weiter haben wir bei der Verwaltung der EEG-Daten ein einfaches Interface zum Speichern, Abrufen und Filtern von Daten implementiert. So können mit den Daten auch noch „tags“ und „extra info“ gespeichert werden, die später für die Filterung und Verarbeitung der Daten sehr nützlich sein können, wie es Programmausschnitt **??** zeigt.

```
...
save_data(experiment, tags = [:room102, :PersonA],
    extra_info = Dict(:temp => 22.3))
...
filter = DataFilter(
    tags = [:PersonA], # nur Daten von Person A
    nottags = [:room105], # keine Daten aus Raum 105
    extra_filter = x -> x.temp < 21 # nur Daten, wo
↪  Temperatur unter 21 war
)
filter!(data, filter)
...
```

Programmausschnitt 4: Geürztes Beispiel, das die Verwendung von „tags“ und „extra info“ zeigt

4.2 Framework

Unser Ziel war es, ein Framework zu entwickeln, das einfach zu benutzen ist und einem Zeit bei der Entwicklung von BCIs erspart. Dafür haben wir ein eigenes Julia-Package geschrieben, mit dem man einfach Daten aufnehmen und speichern sowie ein eigenes neuronales Netzwerk erstellen und in einer Live-Anwendung testen kann.

Dieses Package haben wir anschließend getestet, indem wir es genutzt haben, um ein BCI zu entwickeln, welches erkennen kann, ob man gerade an die Richtung rechts oder an die Richtung links denkt. Diese Klassifizierung ist relativ komplex, da dabei kein einfacher Ausschlag oder Anstieg einer bestimmten Frequenz verursacht wird, anders als z.B. beim Blinzeln oder einem akustischem Signal. Deswegen ist es für einen Menschen auch fast unmöglich, nur anhand der Daten des EEGs manuell festzustellen, ob eine Person gerade an eine bestimmte Richtung denkt.

Um herauszufinden, ob es überhaupt möglich ist mit unserem EEG, das nur 4 Elektroden besitzt, Gedanken an verschiedene Richtungen zu klassifizieren, haben wir zuerst ein neuronales Netz mit Daten, die wir im Internet gefunden haben, trainiert. Diese Daten wurden mit einem 16 Elektroden EEG aufgenommen. Nachdem wir unser Modell mit diesen Daten trainiert haben, haben wir eigene Daten aufgenommen, um zu testen, ob sich dieses Modell auch auf Daten generalisieren lässt, die mit unserem EEG aufgenommen wurden.

Wir waren in der Lage, eine sehr gute Genauigkeit beim Testen des Modells mit Trainingsdaten zu erreichen. Mit der Genauigkeit ist gemeint, welchen Anteil der gegebenen Datensätze das neuronale Netzwerk korrekt klassifizieren konnte. Da die Trainingsdaten dem Modell durch das Trainieren jedoch bereits bekannt waren, haben wir zusätzlich noch die Genauigkeit des Modells bei der Analyse der Testdaten bestimmt. Diese wurden vorher nie zum Trainieren des Netzwerks verwendet und repräsentieren somit viel besser die Genauigkeit des Modell in der späteren Verwendung, wo das Modell mit unbekannten, neuen Umständen und neuen, noch nie zuvor gesehenen Daten klarkommen muss.

Die Genauigkeit lag jedoch bei den Testdaten bei ungefähr ... und somit deutlich niedriger als bei den Trainingsdaten. Dieses starke Overfitting lässt sich wahrscheinlich auf das schlechte Signal-Rausch-Verhältnis zurückführen, das durch die Datenermittlung mit einem EEG entsteht. Man kann also festhalten, dass unsere neuronales Netz zwar besser als zufällig ist, aber trotzdem noch keine sichere Erkennung liefert.

5 Diskussion

Ziel unseres Projektes war es, ein Framework zu entwickeln, welches sich leicht auf die eigenen Zwecke anpassen lässt und das entwickeln von BCIs vereinfacht. Dies haben wir mit unserm eigenen Package erreicht, da es Entwicklung in einer stark abstrahierten Form ermöglicht und aufgrund unserer ausführlichen Dokumentation auch für Anfänger geeignet ist.

TODO:

- Hyper Parameter Optimization
- Mehr Standard Parameter-Sätze
- Integration & Testen des Selbstbau-EEG
- Vergleich Sentdex, unser Ganglion und Selbst-Bau

6 Quellen

6.1 Abbildungen

Abb. **??** Kopiert und abgeändert von https://tikz.net/conv2d/, ursprünglicher Author ist Janosh Riebesell

Abb. **??** Linker Teil: https://www.conrad.de/de/ratgeber/handwerk-industrie-wiki/elektronik-bauteile/lm393.html, Rechter Teil: https://www.reichelt.de/operationsverstaerker-2-fach-dip-8-lm-358-dip-p10483.html?CCOUNTRY=445&LANGUAGE=de