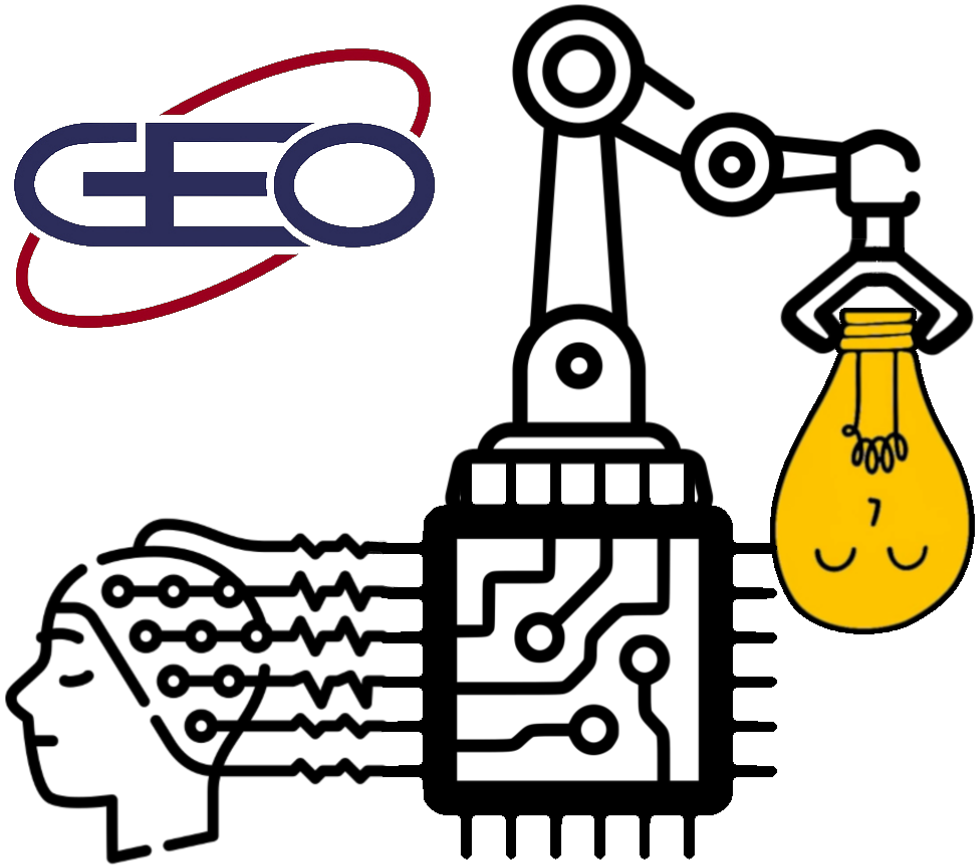
Entwicklung eines Frameworks und eigener Hardware für Brain-Computer-Interfaces

Sammlung, Verarbeitung und Analyse unserer Gehirnsignale



Alexander Reimer, Matteo Friedrich und Mattes Brinkmann

Gymnasium Eversten Oldenburg

Betreuer: Herr Dr. Glade

Inhaltsverzeichnis

Kurzfassung

In diesem Projekt wollen wir ein Framework für ein Brain-Computer-Interface (BCI) verwirklichen. Dafür entwickeln wir unsere eigene, deutlich günstigere Alternative im Vergleich zu den kommerziell erhältlichen Angeboten 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 von Testpersonen an die Richtungen Rechts und Links) ein allgemein verwendbares Framework zu entwickeln, welches die Verarbeitung der Gehirndaten, die Erstellung eines neuronalen Netzes für die Analyse und die anschließende Verwendung des BCI übernehmen kann. Dieses Framework 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.

Wir haben es geschafft, ein Framework zu entwickeln, welches modular aufgebaut und somit gut anpassbar ist. Es bietet außerdem ein Standardkonfiguration, sodass der Nutzer möglichst wenig tun muss. Die Erkennungsgenauigkeit bei der beschriebenen Aufgabe liegt momentan bei 40%; dies wollen wir noch verbessern. Außerdem müssen wir die Standardkonfiguration noch mit anderen Aufgaben testen und verbessern, sodass wir sicher am Ende sicher sein können, dass sie in möglichst vielen Szenarien ohne Anpassung verwendet werden kann.

1 Einleitung

BCIs – Brain-Computer-Interfaces – sind Schnittstellen zwischen Gehirn und Computer und sollen die Kommunikation zwischen diesen ermöglichen. Sie können für die bewusste Steuerung von Geräten wie Prothesen, Drohnen und Robotern verwendet werden, sie eignen sich 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. [1] Außerdem lassen sich BCIs mit instrumenteller oder klassischer Konditionierung sogar zur Veränderung unserer Gehirnmuster und -aktivitäten verwenden. [1]

Ziel des Projektes ist es, ein Framework 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 und für eine möglichst große Spanne an Projekten verwendet werden kann.

Bei der Erarbeitung unseres Projektes möchten wir neben dem Erlangen von Erfahrung in diesem interessanten Bereich auch selbst dazu beizutragen. BCIs sind ein relativ neues Thema und die Forschung in diesem Bereich wird, soweit wir gesehen haben, größtenteils durch Forschungsinstitute, Universitäten und große Technologieunternehmen durchgeführt. Gründe dafür sind vermutlich die hohen Kosten eines hochwertigen EEGs, sowie die Komplexität und der benötigte Aufwand. Außerdem sind EEG Daten sehr schwer zu interpretieren, sodass meistens Methoden der künstlichen Intelligenz (KI), wie zum Beispiel neuronale Netze, zur Verarbeitung der Daten verwendet werden. Dies führt dazu, dass Kenntnisse in verschiedenen Bereichen notwendig sind: Programmierung, KI, Psychologie und Neurophysiologie sowie die Technik und Physik hinter einem EEG.

In unserem Projekt haben wir uns mit einigen dieser Bereiche beschäftigt und haben uns als Ziel gesetzt, den Prozess der Entwicklung eines BCI leichter und zugänglicher zu machen. Dazu haben wir ein eigenes Software-Framework entwickelt. Dieses soll möglichst viele der oben genannten Aufgaben übernehmen und dem Nutzer ein möglichst abstrahiertes Interface bieten, welches z.B. die Aufbereitung der Rohdaten 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 [2]. 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 mit unser damaligen Software nicht realisierbar gewesen.
2. Das BCI 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 unserer Meinung nach noch zu teuer ist, um für private Nutzer zugänglich zu sein.

Durch unser neu entwickeltes Framework werden die Probleme 1 und 2 angegangen. Während das Programm im letzten Jahr eher ein Proof-of-Concept war, konnten wir jetzt, da wir etwas mehr Erfahrung haben, intensiver auf die Software-Gestaltung achten. Um das dritte Problem, den Preis des EEG, anzugehen, haben wir unser eigenes günstigeres Selbstbau EEG entwickelt. Da dieses EEG erst gegen Ende der Forschungszeit fertig geworden ist, haben wir unser Framework mit öffentlich zugänglichen Daten getestet.

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- und 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
- Software von OpenBCI: OpenBCI GUI

3 Vorgehensweise

Für die Entwicklung unseres Framework und EEG haben wir uns das Ziel gesetzt, neben der Programmierung und des Baus gleichzeitig damit ein BCI zu entwickeln, welches erkennen soll, ob eine Testperson gerade an links, rechts oder keine Richtung 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 Kinsley „Sentdex“: Dieser hat auf GitHub einen frei verwendbaren Datensatz an EEG-Daten zur Verfügung gestellt, die er mit genau diesem Ziel (Erkennung der Richtung) aufgenommen hatte. [18] Dieses Anwendungsbeispiel passt sehr gut, da die Erkennung sehr schwer ist. Bei einem sehr einfachen Beispiel hätten wir schnell eine perfekte Genauigkeit von 100% erreicht und könnten somit nicht die Auswirkungen von weiteren Verbesserungen oder Änderungen beobachten.

Diese Vorgehensweise hatte zwei Vorteile: Erstens konnten wir bereits mit der Entwicklung der Software beginnen, während wir gleichzeitig das EEG entwickelten. Zweitens konnten wir unsere Ergebnisse mit den veröffentlichten Ergebnissen vergleichen, bei denen eine Genauigkeit von ca. 60% erreichen werden konnte. Dadurch konnten wir zum einen die Daten als Fehlerquelle ausschließen, da wir wissen, dass die Erkennung möglich sein sollte, und zum anderen konnten wir sehen, wie die Leistung unseres allgemeinen Frameworks im Vergleich zu dem veröffentlichten spezifischen Programm ausfällt.

3.1 Elektroenzephalographie

3.1.1 Grundlagen

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. Durch eine Vergrößerung der Anzahl an 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. Dieser wird dann als Referenzpunkt für alle Elektroden verwendet. Bei der bipolaren Ableitung hat jede Elektrode als Referenz den jeweiligen Nachbarn. Bei zwei Elektroden gibt es also nur eine Ausgabe – die Differenz der beiden. Bei drei Elektroden gibt es zwei Ausgaben, einmal die Differenz von Elektroden 1 und 2 und einmal die Differenz von Elektroden 2 und 3. [3]

Üblicherweise wird die unipolare Ableitung verwendet, da bei der bipolaren Ableitung Daten permanent verloren gehen, da es einen Datenkanal weniger gibt (siehe oben) und eine Subtraktion keine umkehrbare Operation ist. Außerdem kann es bei zwei entgegengesetzten EEG-Spannungen (z.B. Elektrode 1 = 7 µV, Elektrode 2 = −7 µV) zur Auslöschung des Signals kommen. [3] In allen von uns verwendeten EEG-Daten (sowohl extern als auch selbst erzeugt) handelt es deshalb um unipolar abgeleitete Potentiale.

3.1.2 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. [3] Zweitens müssen die gewünschten Frequenzen vorab mittels aktiver Filter herausgefiltert werden, um unerwünschte Störfrequenzen zu entfernen und so eine genauere Signalanalyse zu ermöglichen. Je nachdem, welche Phänomene untersucht werden sollen, können unterschiedliche Frequenzbereiche herausgefiltert werden. So wären z.B. bei der Analyse von Konzentration oder visueller Verarbeitung vor allem Alpha-Wellen, die in einem Frequenzbereich von 8 bis 13 Hz liegen, relevant. [4, 5, 6] Drittens wird eine Einheit benötigt, die das Ausgangssignal misst und aufzeichnet, damit es weiterverarbeitet und analysiert werden kann.

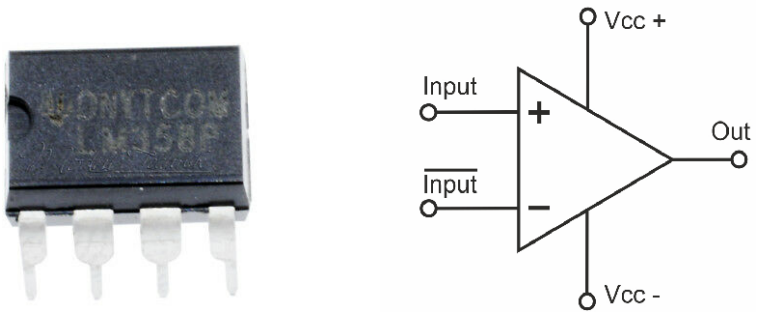


Abbildung 1: Links: 2-fach OPV LM358 in einem DIL-Gehäuse (Dual In-line Package), Rechts: OPV als Schaltbild

Sowohl für die Filterung als auch für die Verstärkung werden sogenannte Operationsverstärker (OPV) verwendet. In Abb. 1 sieht man rechts das Schaltzeichen mit den jeweiligen Anschlüssen und links einen OPV in einem DIL-Gehäuse. 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 Schaltungsteile untergliedern, den sogenannten Differenzverstärker, der später für die Potentialmessung wichtig wird, und die dahinter geschaltete Gegentaktendstufe, welche für die Leistungsverstärkung zuständig ist. Ihr geringer Ausgangswiderstand, der hohe Eingangswiderstand, sowie der hohe Spannungsverstärkungsfaktor machen den OPV passend für die Anwendung als EEG-Verstärker.

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 welchen er gesteuert werden kann (Abb. 1 rechts). Legt man nun bei 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: $U = U_2 - U_1$. Dies macht man sich nun zu Nutze, um die Spannungen bzw. Potentialdifferenzen zwischen zwei Elektroden zu messen. [7] Zudem lassen sich mit einem OPV und wenigen weiteren Bauteilen einfach aktive Filter bauen.

Die Grundschaltung für unser EEG beruht auf einem englischsprachigem Internetartikel, in dem eine Verstärkerschaltung für ein EEG beschrieben ist (siehe Abb. 2). [8] Wir mussten die ursprüngliche Schaltung etwas verändern, um den gewünschten Frequenzbereich herauszufiltern. Basis dieser Schaltung bildet der Instrumentenverstärker AD620AN, welcher extra für EEG- und EKG-Anwendungen entwickelt wurde. Ein

Instrumentenverstärker ist eine Schaltungskonfiguration aus 3 OPVs, wobei zwei OPVs zu einem Differenzverstärker zusammengeschaltet werden und ein weiterer OPV als Impedanzwandler dahinter geschaltet wird. Instrumentenverstärker sind sehr präzise und haben einen nochmal höheren Verstärkungsfaktor. Über Pin 2 und 3 werden die Elektroden angeschlossen, wobei an Pin 2 die Referenzelektrode geklemmt wird.

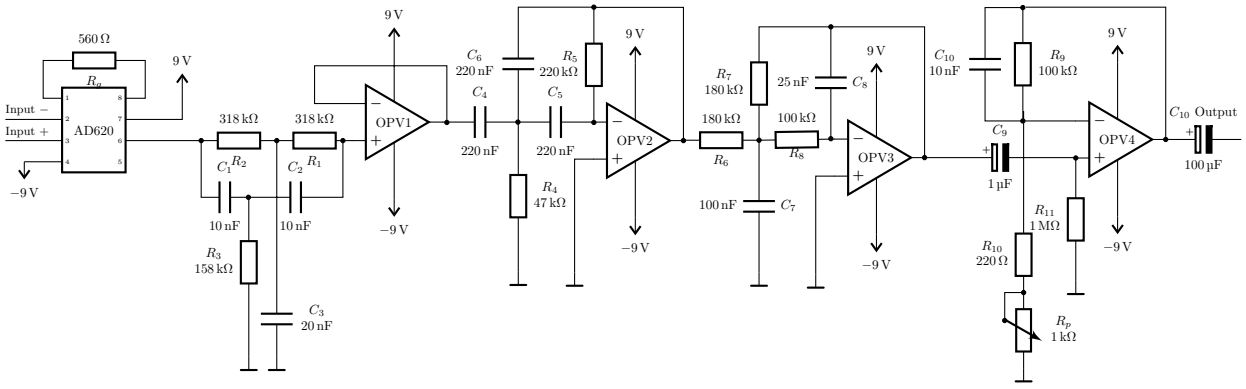


Abbildung 2: Schaltplan des EEG-Verstärkers für eine Elektrode und eine Referenzelektrode, OPV1 bis OPV4 sind alle LM358 (siehe Materialliste)

Der AD620AN ist ein sehr rauscharmes und leicht zu schaltendes IC, da für die Einstellung des Verstärkungsfaktors nur ein Widerstand nötig ist. Über die Formel im Datenblatt kann man den Verstärkungsfaktor berechnen [9]:

$$G_1 = \frac{49{,}4\,\mathrm{k\Omega}}{R_g} + 1 = \frac{49{,}4\,\mathrm{k\Omega}}{0{,}560\,\mathrm{k\Omega}} + 1 = 89.2 \approx 90$$

Hätte man einen größeren Verstärkungsfaktor gewählt, als den der vom Artikel vorgegeben ist, hätte man das, dass dabei das Signal noch absolut ungefiltert verstärkt wird, was bedeutet, dass alle Signale bzw. alle Frequenzen, die das eigentlich gewünschte Signal stören oder überlagern, noch mit verstärkt werden. Die Filter, die nur eine festgelegte Dämpfung haben, können die Störsignale dann nicht mehr ausreichend unterdrücken. Deswegen wird zunächst über den AD620AN nur eine Vorverstärkung vorgenommen. Danach findet eine schrittweise Filterung des Signals statt. Als allererstes wird über den sog. Notch- oder auch Kerbfilter, der aus OPV1 und vorgeschalteten Widerständen sowie Kondensatoren besteht, das 50 Hz-Netzbrummen herausgefiltert. Dieses kann das EEG-Signal erheblich stören, da der menschliche Körper unter anderem als eine Art Kondensator fungiert und die 50 Hz-Netzfrequenz des europäischen Stromnetzes zum Eingang des Verstärkers leitet.

Die Dimensionierung des Verstärkers ist dabei recht einfach. Über die gewünschte Grenzfrequenz $f_g = 50\,\mathrm{Hz}$ sowie die Kapazität der Keramikkondensatoren ($C_1 = C_2 = 10\,\mathrm{nF}$), siehe Abb.2) lässt sich der Widerstandswert der beiden Widerstände R_1 und R_2 berechnen. Dazu wird diese Formel verwendet [10, 11]:

$$f_g = \frac{1}{R_1 \cdot C_1 \cdot 2 \cdot \pi}$$
$$\Leftrightarrow R_1 = \frac{1}{C \cdot 2 \cdot \pi \cdot f_g} = \frac{1}{10\,\mathrm{nF} \cdot 2 \cdot \pi \cdot 50\,\mathrm{Hz}} \approx 318{,}31\,\mathrm{k\Omega}$$

Weiterhin gilt für die Schaltung $C_3 = 2 \cdot C_1, R_3 = R_1 \cdot 0.5$. Es ergibt sich also:

$$C_3 = 2 \cdot 10\,\mathrm{nF} = 20\,\mathrm{nF}, \quad R_3 = \frac{318{,}31\,\mathrm{k\Omega}}{2} \approx 159{,}16\,\mathrm{k\Omega}$$

Zudem sind die möglichen Widerstandswerte, je nach Baureihe der Komponenten (E12/E24), beschränkt. Man muss sich also durch geschickte Reihen- und Parallelschaltung der Widerstände bzw. Kondensatoren den Werten annähern. Die 318,31 kΩ-Widerstände R_1 und R_2 bestehen z.B. jeweils aus den drei Widerständen 270 kΩ, 33 kΩ und 15 kΩ, was mit einem Gesamtwiderstand von 318 kΩ für uns so nah wie möglich am Soll-Wert liegt.

Der OPV2 und die davor geschalteten Komponenten bilden den sog. aktiven Hochpassfilter mit Mehrfachgegenkopplung. Dieser Filter arbeitet mit einer Gegenkopplung, was bedeutet, dass das Ausgangssignal über ein passives Filterglied wieder zurück auf den Eingang zurückgeführt wird. Dies sorgt für eine bessere Steilheit im Frequenzgang eines Filters. Ein Hochpassfilter lässt nur Frequenzen, die oberhalb seiner festgelegten Grenzfrequenz liegen, durch. Der Frequenzgang beschreibt den Frequenzverlauf in Abhängigkeit zur Dämpfung und wird in einen Sperr- und einen Durchlassbereich sowie einen dazwischenliegenden Übergang gegliedert. Der ideale Filter würde bei der Grenzfrequenz eine unendliche Dämpfung erreichen. In der Realität verläuft der Übergang vom Durchlass- in den Sperrbereich nicht abrupt, sondern, wenn eine logarithmische Skalierung verwendet wurde, linear. In Abb. 3 kann man dies gut sehen.

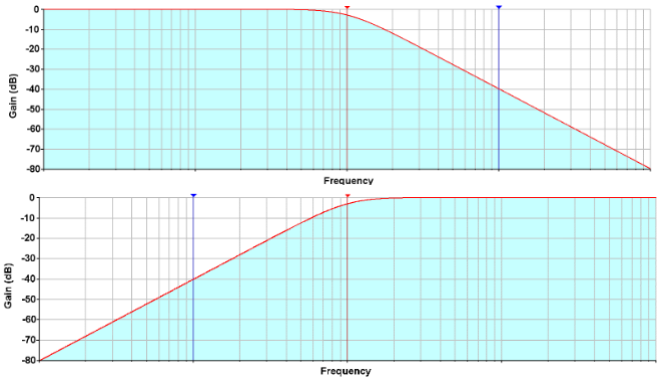


Abbildung 3: Typischer Frequenzgang eines Tiefpassfilters (oben) und Hochpassfilters (unten), Rote Linie ist die Grenzfrequenz, Blaue Linie eine Dekade Abstand zur Grenzfrequenz, um Steilheit zu bestimmen

Die Steilheit beschreibt die Steigung dieser Übergangsgerade und wird in dB/Dekade angegeben. Eine Dekade beschreibt den Abstand zwischen 10^xHz und 10^{x+1}Hz. Also z.B den Abstand zwischen 10 Hz und 100 Hz oder 1000 Hz und 10 000 Hz. Je steiler die Übergangsgerade, desto besser ist der Filter. Des Weiteren ist an der Grenzfrequenz immer ein Abfall von ca. 6 dB zu beobachten, wenn eine Mehrfachrückkopplung verwendet wird. Bei aktiven Filtern, wie sie hier beschrieben sind, erreicht man eine Steilheit von 40 dB/Dekade. Für die Berechnung der Grenzfrequenz des Hochpasses mit Mehrfachgegenkopplung benötigen wir eine etwas andere Formel, als die, die wir beim Notchfilter verwendet haben. Da alle Werte gegeben sind wird die Berechnung nur bestätigt:

$$f_g = \frac{1}{2 \cdot \pi \cdot C_5 \cdot \sqrt{R_4 \cdot R_5}} \Rightarrow \qquad f_g = \frac{1}{2 \cdot \pi \cdot 220\,\text{nF} \cdot \sqrt{47\,\text{k}\Omega \cdot 220\,\text{k}\Omega}} = 7{,}114\,\text{Hz} \approx 7\,\text{Hz}$$

Der Hochpass hier ist auf eine Grenzfrequenz von ca. 7 Hz eingestellt. Im Anschluss auf den Hochpass folgt ein Tiefpass (OPV3), der die selbe Funktionsweise hat, mit der Ausnahme, dass er nur die Frequenzen **unter** der Grenzfrequenz durchlässt. Mit der Formel zur Berechnung der Grenzfrequenz eines Tiefpasses ergibt sich für uns [12]:

$$f_g = \frac{1}{2 \cdot \pi \cdot \sqrt{R_7 \cdot R_8 \cdot C_7 \cdot C_8}} \Rightarrow \qquad f_g = \frac{1}{2 \cdot \pi \cdot \sqrt{180\,\text{k}\Omega \cdot 100\,\text{k}\Omega \cdot 100\,\text{nF} \cdot 25\,\text{nF}}} = 23{,}72\,\text{Hz} \approx 24\,\text{Hz}$$

Der Tiefpass ist in unserem Fall für eine Grenzfrequenz von 24 Hz eingestellt.

Nach einem RC-Filterglied folgt dann der Endverstärker, welchen OPV4 mit entsprechenden Zusatzkomponenten bildet. Hierbei handelt es sich um einen nicht-invertierenden Verstärker, dessen Verstärkung über den Spannungsteiler, der aus dem 100 kΩ-Widerstand, dem 220 kΩ-Widerstand und dem 1 kΩ-Potentiometer R_{Pot} besteht, eingestellt werden kann. Mit der folgenden Formel lässt sich die minimale ($R_p = 1\,\text{k}\Omega$) und

maximale ($R_p = 0\,\Omega$) Verstärkung berechnen [8]:

$$G_2 = \frac{R_9}{R_{10} + R_p} + 1$$
$$\xrightarrow{R_p = 0\,\mathrm{k\Omega}} G_2 = \frac{100\,\mathrm{k\Omega}}{0{,}22\,\mathrm{k\Omega} + 0\,\Omega} + 1 = 455$$
$$\xrightarrow{R_p = 1\,\mathrm{k\Omega}} G_2 = \frac{100\,\mathrm{k\Omega}}{0{,}22\,\mathrm{k\Omega} + 1\,\mathrm{k\Omega}} + 1 = 83$$

Insgesamt ergibt sich bei unserem EEG also ein Verstärkung von maximal $G_1 * G_2 \Rightarrow 455 \cdot 90 = 40950$. Der Verstärker wurde bereits mittels Oszilloskop getestet, dabei fiel auf, dass das Signal noch durch 50 Hz-Brummen gestört wird. Deswegen wird das ganze EEG nun über 9 V-Blockbatterien betrieben, um dem 50 Hz-Brummen eines Netzteils entgegenzuwirken, und zusätzlich noch über einen Masseanschluss geerdet. Alle diese Dinge hatten ein positiven Einfluss auf das EEG-Signal, reichten aber immer noch nicht ganz aus, um gute Ergebnisse zu erzielen. Deswegen ziehen wir in Erwägung, am Ende der Schaltung noch einen weiteren Notchfilter zu schalten. Der ganze Aufbau ist provisorisch auf einem Steckbrett erfolgt (s. Abb. 4), soll aber fest auf einer Platine verlötet und in ein Gehäuse verbaut werden, sobald es ausreichend getestet wurde und zufriedenstellende Ergebnisse geliefert hat. Unabhängig davon haben wir parallel an einer Mess-

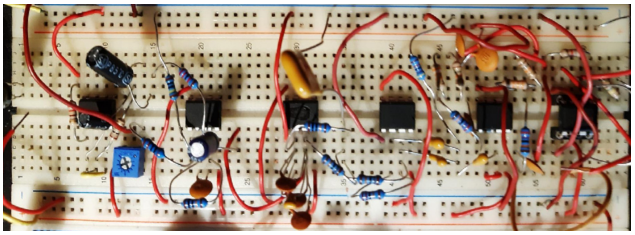


Abbildung 4: Schaltung fertig auf einem Steckbrett aufgebaut

und Ausleseeinheit gearbeitet, die ein wichtiger Teil für weitere Datenverarbeitung darstellt. Diese Einheit besteht aus dem Analog/-Digitalwandler (ADC) MCP3208, der an an die SPI-Schnittstelle eines Raspberry Pi 3B angeschlossen wird. Der MCP3208 besitzt 8 Analoge Eingänge und hat eine Auflösung von 12 Bit sowie eine Abtastrate von 100ksp/s. [13]

Ein ADC funktioniert, indem er einen Spannungsmessbereich, der über einen Referenzpin am IC festgelegt wird, in einzelne digitale Zustände aufteilt. Über die Auflösung kann berechnet werden, in wie viele Zustände der ADC den Messbereich unterteilt, bei 12 Bit ergeben sich $2^{12} = 4096$ Zustände. Das bedeutet, dass bei einer Referenzspannung U_{Ref} von 5 V die Eingangsspannungen in 4096 Zustände unterteilt werden, sodass das kleinste erkennbare Spannungsintervall $U_{\mathrm{Intervall}} = \frac{5\,\mathrm{V}}{4096} \approx 1{,}2\,\mathrm{mV}$. Der ADC schreibt also allen Vielfachen von 1,2 mV eine digitale Ganzzahl zu. So entsprächen 2,4 mV dem Wert 2, 3,6 mV dem Wert 3, usw. Nun lässt sich aus dem Digitalwert, den uns der MCP3208 liefert, ein Spannungswert berechnen:

$$U = U_{\mathrm{Intervall}} \cdot \mathrm{Digitalwert}$$

Die Messgenauigkeit ist in diesem Fall mehr als ausreichend, da durch den EEG-Verstärker bei maximaler Verstärkung die untere Grenzspannung für EEG-Messungen von 1 µV auf ca. 41 mV verstärkt wird.

Ein Programm, um den Analog/-Digitalwandler auszulesen, wurde bereits geschrieben. Der Zusammenbau beider Komponenten steht jedoch noch aus, da sowohl der EEG-Verstärker als auch die Ausleseeinheit noch verbessert und ausreichend getestet werden müssen.

3.2 Fourier-Analyse

Die Fourier-Analyse kann die verschiedenen zugrundeliegenden Frequenzen von Datenfolgen oder Funktionen bestimmen, indem diese in Sinus-Kurven mit verschiedenen Frequenzen zerlegt werden, welche summiert dem ursprünglichem Signal entsprechen. Zu diesem Zweck wird jeder 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 die Frequenz dieses Wertes (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 des Signals bei dieser Frequenz. So lässt sich bestimmen, welche Frequenzen am stärksten im Signal vorkommen. Außerdem können auf diese Weise Frequenzen herausgefiltert werden, indem die Werte bei den entsprechenden Indices auf 0 gesetzt werden.

Eine Spektralanalyse ergibt v.a. bei EEG-Daten Sinn, da diese inherent auf Frequenzen aufgebaut sind, denn viele Neuronen im Gehirn, v.a. mit ähnlichen Aufgaben, sind synchronisiert, sodass sie eine ähnliche Frequenz haben. Es lassen sich verschiedenen Aktivitätsstufen der Neuronen und Aufgaben-/Gehirnbereichen verschiedene Frequenzbereiche zuweisen, wie schon in Bau unseres eigenen EEG beschrieben.

Eine FFT kann Elektroenzephalogramme verlustfrei repräsentieren. Dies lässt sich erkennen, indem 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. 5 sichtbar, ist diese Rekonstruktion nicht von den originalen Daten unterscheidbar.

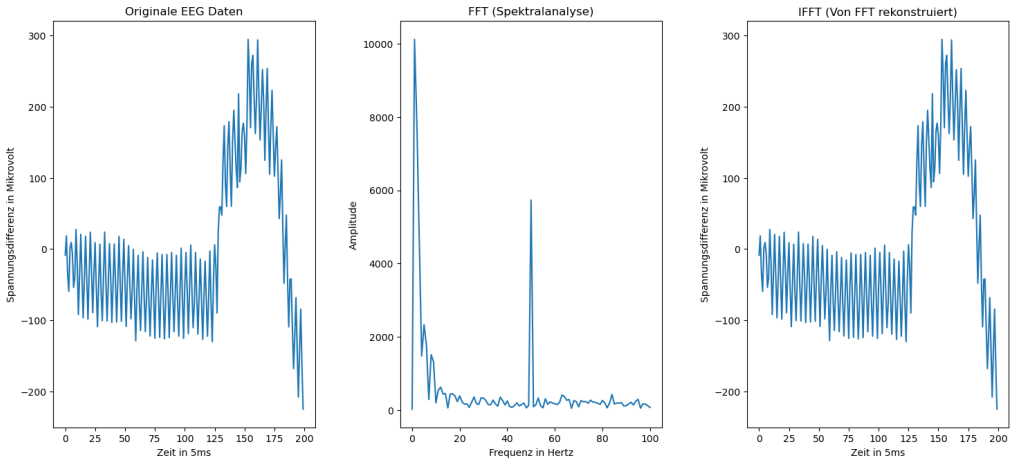


Abbildung 5: Links: Originale EEG Daten, Mitte: Die Amplituden der Frequenzen dieser Daten, Rechts: Die Umkehrung der Frequenzdaten. Da der rechte Graph dem linken sehr ähnlich ist, lässt sich erkennen, dass es bei der Fourier Analyse keinen großen Informationsverlust gibt.

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

Jedes Neuron besitzt eine sogenannte Aktivierung, die meist als Zahl im Intervall zwischen 0 und 1 oder zwischen -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.

3.3.1 Forward Pass

Bei neuronalen Netzwerken gibt es zwei wichtige Verfahren: Den Forward Pass und die Backpropagation. Ersteres beschreibt das Weiterreichen von Eingabewerten durch das Netzwerk. Zuerst werden die Aktivierungen der Neuronen in der Eingabeschicht gesetzt und danach mithilfe der verbindenden Gewichte zuerst die Netzeingaben und dann die Aktivierungen der Neuronen der nächsten Schicht berechnet, welche dann wiederum für die nächsten Schicht verwendet werden und so weiter. Dadurch werden die Eingaben von Schicht

zu Schicht bis zur Ausgabe weiterverarbeitet. Das konkrete Verfahren zur Berechnung der Netzeingabe ist abhängig von Art der Schicht.

Man wendet auf diese Netzeingabe eine Aktivierungsfunktion an, um die Aktivierung zu erhalten. Dies sorgt dafür, dass es keine Linearität zwischen Eingaben und Ausgaben gibt, wodurch das Netzwerk auch nichtlineare Zusammenhänge modellieren kann.

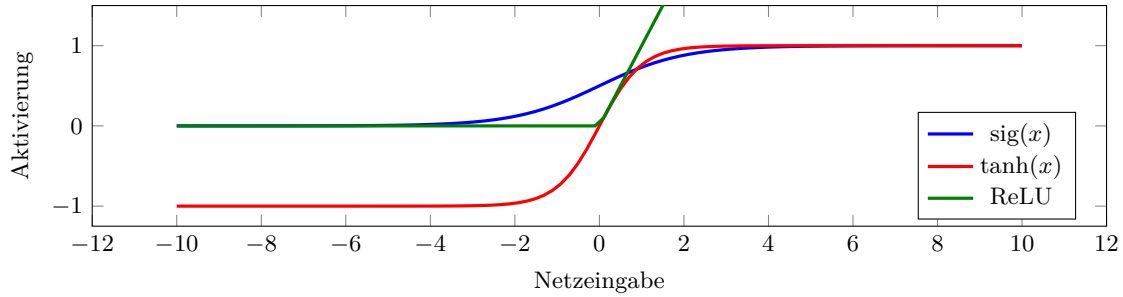


Abbildung 6: Aktivierungsfunktionen im Vergleich

Letztes Jahr haben wir dafür die Sigmoid-Funktion genutzt, welche die Aktivierung auf Werte zwischen 0 und 1 beschränkt (s. Abb. 6).

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

Doch dieses Jahr setzen wir auf die tanh-Aktivierungsfunktion (Hyperbeltangens) [14], bei der die Ausgabe immer zwischen -1 und 1 liegt (s. Abb. 6), da sie eine bessere Lernleistung für neuronale Netze bietet. [15, 16]

$$\tanh(x) = 1 - \frac{2}{\mathrm{e}^{2x} + 1}$$

Außerdem nutzen wir die ReLU-Aktivierungsfunktion, da sie besonders gut für die Convolutional Schichten geeignet ist. Die ReLU-Aktivierungsfunktion gibt für alle negativen Eingabewerte 0 aus und lässt alle positiven Eingabewerte unverändert.

3.3.2 Backpropagation

Der Forward Pass bildet den ersten Schritt der Backpropagation, die für das Lernen des Netzwerkes verantwortlich ist. Dabei werden, nachdem alle Aktivierungen gesetzt wurden, die Aktivierung der letzten Schicht, also die Ausgaben, bewertet. Da wir Supervised Learning verwenden, benötigen wir dafür Trainingsdaten, denen wir vorher die korrekten Ausgabedaten zugewiesen haben. Mit diesen wird dann jedes einzelne Ausgabeneuron mit einer sogenannten Verlustfunktion "bewertet". Je weiter die Ausgaben des neuronalen Netzwerks von dem Ideal (im Trainingsdatensatz vorgegebene Ausgabedaten) abweichen, desto höher der sogenannte Verlust, der mit der Verlustfunktion berechnet wird.

Diese Differenz zwischen ist-Zustand und soll-Zustand wird nun in der Backpropagation verwendet, um die Gewichte anzupassen, denn mit ihr kann der sogenannte δ-Wert der Ausgabeschicht bestimmt werden. Dieser entspricht der Änderung der Aktivierung eines Neurons, die notwendig wäre, damit ein Forward Pass ein besseres Ergebnis.

Um diese δ-Werte für die Neuronen der verdeckten Schichten zu berechnen, werden natürlich die δ-Werte der dahinter liegenden Schichten benötigt, da die benötigte Änderung nur von den Ausgaben aus bestimmt werden kann. Aufgrund dieses Zurückpropagierens der δ-Werte zur Eingabeschicht hin kommt auch der Name Backpropagation. Auf die Berechnung des δ-Werts werden wir dieses Jahr nicht weiter eingehen, da sie für uns nicht direkt relevant ist und wir für unsere neuronale Netzwerke das Julia-Package Flux einsetzen, welches diese Arbeit übernimmt.

Mithilfe dieses δ-Werts wird dann bestimmt, wie die Gewichte des Netzwerks angepasst werden müssten, um diese gewünschte Änderung umzusetzen:

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

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. Das 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.

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. Umgesetzt wird diese Änderung einfach mit

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

3.3.3 Verlustfunktion

Insgesamt lässt sich sagen, dass ein neuronales Netzwerk versucht, die Verlustfunktion zu minimieren. Es gibt viele verschiedene Arten von Verlustfunktionen und wir haben uns dazu entschieden, den Mean Squared Error (kurz MSE) als Verlustfunktion zu nutzen, welcher sich mit folgender Formel berechnen läßt:

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

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.

3.3.4 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 (s. Abb. 7).

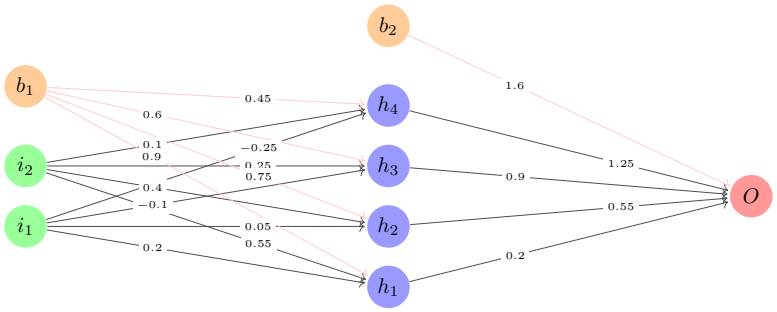


Abbildung 7: Ein Beispiel für ein neuronales Netzwerk bestehend aus einer Eingabeschicht(i_1 und i_2) und zwei Fully Connected Schichten (Neuronen h_1 bis h_4 mit Bias b_1 und Neuron O mit Bias b_2). Die Kreise stellen Neuronen dar; die Zahlen auf den Pfeilen die Gewichte zwischen den jeweiligen Neuronen.

Um die Netzeingabe eines Neurons in einer Fully Connected Schicht 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. [19]

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.

Fully Connected Schichten sind die „Standard“-Schicht bei neuronalen Netzwerken. Sie können und werden vielseitig eingesetzt, doch haben zwei Nachteile: Da jede Eingabe und jedes folgende Neuron ihr eigenes, selbst angepasstes Gewicht hat, ist bei der Erkennung von Mustern bei Klassifizierungen die „Position“ der zu erkennenden Sache in der Eingabe relevant. Wenn also das Netzwerk zum Beispiel mit vielen Hundebildern trainiert wird, in den Trainingsdaten der Hund aber nie in der rechten oberen Ecke war, dann können die Gewichte, die mit den Eingabeneuronen in dieser Ecke verbunden sind, auch nicht trainiert worden sein und somit bei der tatsächlichen Anwendung Hunde dort nie erkannt werden.

3.3.5 Convolutional Layer

Im Gegensatz zu Fully Connected Schichten, in denen jedes Neuron einer Schicht mit allen Neuronen der nächsten Schicht verbunden ist, sind die Neuronen einer Convolutional Schicht jeweils nur mit ein paar Neuronen der nächsten Schicht verbunden.

Convolutional Schichten 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. Die Werte eines Fiterkernels sind die Gewichte einer Convolutional Schicht.

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 Filterkernels 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 (s. Abb. 8).

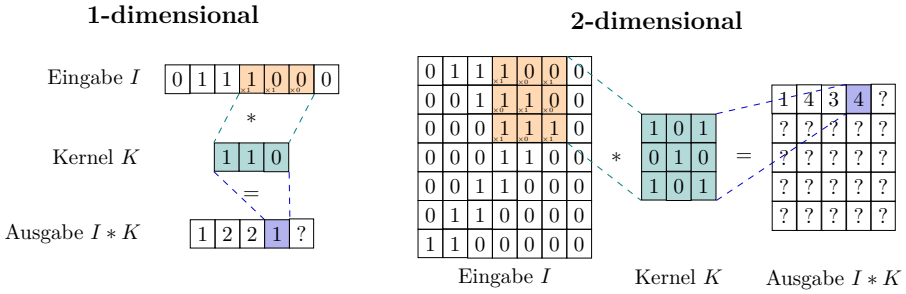


Abbildung 8: Eine 1-dimensionale und eine 2-dimensionale Convolutional Schicht. Die farbig hervorgehobenen Felder stellen einen Schritt des Kernels dar; als nächstes würde er ein Feld weiter nach rechts rücken und die Ausgabe auch.

Beim Trainieren des neuronalen Netzwerkes werden bei Convolutional Schichten die Gewichte, also die einzelnen Werte der Kernel, als anzupassende Gewichte behandelt. Bei Convolutional Schichten redet man meistens davon, dass ein Filterkernel eine Eigenschaft (engl. *feature*) extrahiert. So könnte ein Filterkernel zum Beispiel alle vertikalen Kanten eines Bildes, alle scharfen Kanten einer Höhenkarte oder alle runden Objekte in einem LIDAR-Scan erkennen.

Der Vorteil von Convolutional Schichten ist, dass sie das bereits beschriebene Problem von Fully Connected Schichten lösen. Denn dadurch, dass der Kernel 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.

Convolutional Schichten haben außerdem meistens mehrere Kanäle, sodass sie mehre Eigenschaften parallel untersuchen können. 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.

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

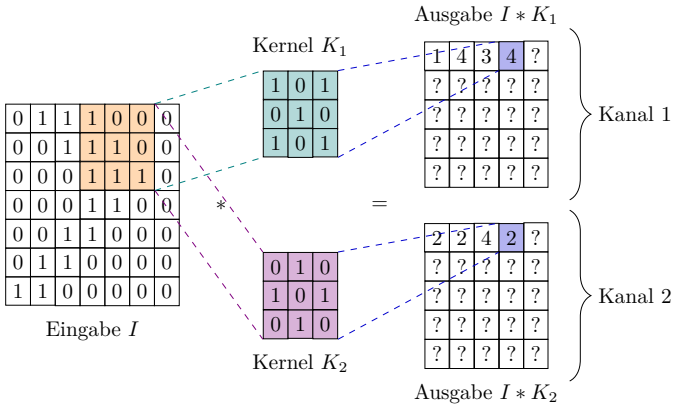


Abbildung 9: Eine 2-dimensionale Convolutional Schicht mit zwei Kanälen und entsprechend zwei Kerneln (K_1 und K_2) und zwei Ausgaben ($I * K_1$ und $I * K_2$). Es kann theoretisch beliebig viele Kanäle geben.

3.3.6 Pooling Schichten

Pooling Schichten haben die Funktion, unnötige Informationen zu verwerfen. Dies funktioniert, indem n Neuronen einer Schicht zu einem einzelnem Neuron in der nächsten Schicht zusammengefasst werden. Es gibt verschiedene Arten von Pooling Schichten, doch die am häufigsten genutzte Art ist die Max Pooling Schicht. Dabei wird von den verschiedenen Neuronengruppen, die zusammengefasst werden sollen, jeweils nur das Neuron mit der größten Aktivierung an die nächste Schicht weitergegeben.

3.3.7 Overfitting

Unterschiede zwischen Testdaten- und Trainingsdaten-Genauigkeit sind ein bekanntes Problem bei neuronalen Netzwerken und werden durch sog. „Overfitting“ verursacht. Dabei passt sich das Netzwerk zu stark an die Trainingsdaten an und „speichert“ die korrekten Ausgaben indirekt in den Gewichten ab. Dies geschieht, da dies die Kosten so am leichtesten minimiert werden. Jedoch ist das abspeichern von Daten schlecht, da das Netzwerk damit den Prozess des Findens von Mustern in den Daten umgeht und somit nicht in der Lage ist, neue, unbekannte Daten korrekt zu klassifizieren. Zwei Hauptursachen für Overfitting sind also eine zu große und komplexe Netzwerkstruktur, der „Speicher“ des Netzwerks, sowie zu wenige oder nicht ausreichend unterschiedliche Daten.

Eine weiter Ursache kann ein schlechtes Signal-Rausch-Verhältnis 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 die Haut stark abgeschwächt werden, bevor sie bei den Elektroden ankommen. Durch großes Hintergrundrauschen ist es für ein neuronales Netz schwer, grundlegende Muster in den Daten zu finden. Dadurch tendiert das Modell bei der Verlustoptimierung der Neuronen trotzdem zum Overfitting. Die Verkleinerung der Netzwerkstruktur kann aus dem selben oben beschriebenen Grund ein Problem darstellen, da eine gewisse Mindestgröße benötigt wird, um ein Problem überhaupt lösen zu können. Das Sammeln von mehr Trainingsdaten kann zwar gegen Overfitting helfen, ist aber sehr aufwändig und sorgt für einen größeren Bedarf an Speicherplatz sowie Rechenleistung.

Bei der Verwendung von Dropout (dt. *fallen lassen*) wird einer Schicht des neuronalen Netzwerkes eine Dropout-Wahrscheinlichkeit P 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 beibehält oder dieser auf 0 gesetzt wird. Bei einer Dropout-Wahrscheinlichkeit von 30% werden also im Schnitt 30% der Neuronen auf 0 gesetzt und somit ignoriert. Dies führt dazu, dass das neuronale Netzwerk die Daten nicht so einfach auswendig lernen kann.

Außerdem gibt es die L1- und L2-Regularisierungen, bei denen eine Penalties-Funktion zum Verlust hinzugefügt wird, die das Ausmaß der Gewichte des Modells beschränkt. Somit wird das Netzwerk gezwungen, für einen niedrigen Verlust eine gewisse Einfachheit beizubehalten, was das indirekte Speichern der Ergebnisse ausschließt.

Die Anwendung von Rauschen (*noise*) auf den Eingaben für ein Neuronales Netzwerk hilft auch dabei, bei Overfitting die Testdatengenauigkeit zu erhöhen. Die Idee dahinter ist, dass dadurch eine größere Menge an unterschiedlichen Trainingsdaten „simuliert“ wird, da die Eingaben so in jeder Epoche unterschiedlich sind.

Eine weitere Methode ist das Pruning. Dabei werden zufällige einzelne Neuronen komplett aus dem Netzwerk entfernt, wodurch die Komplexität des Netzwerkes sinkt. So wird es für das Netzwerk schwerer, die Daten auswendig zu lernen, und Overfitting wird verhindert. Das Pruning lässt sich noch weiter verbessern, indem man einen Forward Pass mit allen Testdaten und einen mit allen Trainingsdaten durchführt und dabei für jedes Neuron jeweils die durchschnittliche Aktivierung bestimmt. Anschließend muss man die durchschnittlichen Aktivierungen beim Ausführen mit Testdaten von den durchschnittlichen Aktivierungen beim Ausführen mit Trainingsdaten subtrahieren. So erfährt man für jedes Neuron, wie sehr es zum Overfitting beiträgt: Nur ohne Overfitting ist die Differenz bei gleicher Fall-Verteilung in Trainings- und Testdaten ungefähr 0. Nun muss man nur noch die Neuronen mit der höchsten Differenz entfernen.

3.3.8 Unsere Implementation

Wir bieten in unserem Framework dem Nutzer eine Standard-Konfiguration, die beliebig abgeändert werden kann. Die Hoffnung ist, dass Nutzer nicht zuerst alle Komponenten verstehen und zusammenbauen müssen, sondern sich auf Standard-Werte und Prozesse verlassen können, die in den meisten Kontexten funktionieren und nur in seltenen Fällen angepasst werden müssen.

Unser Plan, um dieses Ziel zu erreichen, ist es, die Standard-Konfiguration zuerst gut an unser aktuelles Implementationsbeispiel (Gedanken links, rechts) anzupassen. Danach wollen wir weitere Beispiele umsetzen (Augen schließen, akustische Signale, etc. und die Konfiguration ebenso an diese anpassen, sodass wir am Ende hoffentlich einen Satz an Standard-Werten haben, der für alle Beispiele gut funktioniert, oder zumindest mehrere, die auf Gruppen von Problemen anwendbar sind.

Wir verwenden in dieser Standard-Konfiguration selbst nicht alle oben beschriebenen Verfahren gegen Overfitting, doch wir haben sie alle umgesetzt, sodass wir und Nutzer sich jederzeit dazu entscheiden können, sie zu verwenden und zu testen.

4 Ergebnisse

Der gesamte Quellcode unseres Projektes sowie alle gesammelten Daten lassen sich auf unserem GitHub Repository [20] 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 [17] 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 1). Alle von dem Paket benötigten Abhängigkeiten werden automatisch installiert und mitverwaltet.

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 2.

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

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

```
struct Dog end # eigenen Datentypen namens "Dog" definieren
struct Cat end # eigenen Datentypen namen "Cat" definieren
makenoise(animal::Dog) = println("Wuff!")
makenoise(animal::Cat) = println("Meoow!")
```

Programmausschnitt 2: Multiple-Dispatch-Beispiel. Wenn `makenoise(animal)` ausgeführt wird, entscheidet der Typ von `animal`, welche der zwei Funktionen verwendet wird.

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 3 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 Übersichtsichtlichkeit führen könnte.

```
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)
    return rand() # gib Zufallszahl zwischen 0 und 1 zurück
end
device = Device(MyBoard(NUM_CHANNELS) # Das eigene Board verwenden
```

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. Die Filterung ist noch nicht vollständig umgesetzt (momentan können tags nur genutzt werden, um festzulegen, welchen Datensätzen welche Ausgaben zugeordnet sind) und extra infos funktionieren aufgrund von Schwierigkeiten mit CSV.jl noch nicht, doch abgesehen von letzterem erwarten wir, dass wir diese Funktionen sehr bald

implementiert haben werden, da sie größtenteils bereits integriert sind und nur noch öffentliche Methoden zur Verwendung fehlen.

4.2 Das neuronale Netzwerk

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. Bei der Evaluierung des Modells mit Testdaten konnten wir eine Genauigkeit von bis zu ungefähr 44,57% erreichen (s. Abb. 10).

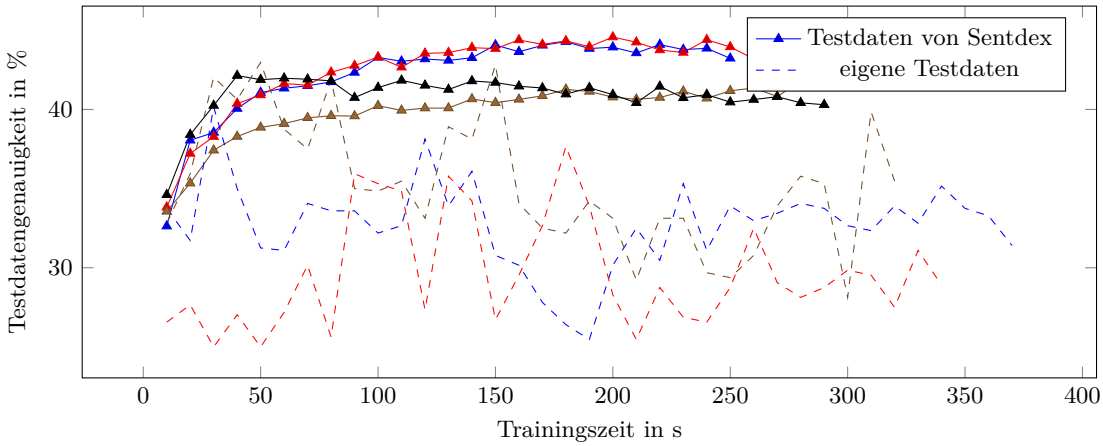


Abbildung 10: Die Testdatengenauigkeit des in Abb. 11 dargestellten Netzwerks im Verlauf. Die durchgezogenen Linien mit Dreiecken repräsentieren die Genauigkeit mit den Testdaten von Sentdex, die gestrichelten Linien die Genauigkeit mit unseren eigenen Testdaten, die mit dem Ganglion aufgenommen wurden. Der unterschiedliche Verlauf der Linien gleichen Typs kommt daher, dass das Netzwerk beim Erstellen mit zufälligen Gewichten initialisiert wird und somit immer einen anderen Verlauf hat.

Da wir drei verschiedene Kategorien von Daten nutzen (Gedanken an Rechts, Gedanke an Links und Gedanken weder Rechts noch Links), würde eine zufällige Klassifizierung der Daten zu einer Genauigkeit von 33% führen.

Die Struktur des Netzwerks, mit dem wir dies erreicht haben, ist in Abb. 11 zu sehen. Wir haben die Struktur größtenteils mit *trial and error* ermittelt und die in Overfitting beschriebenen Verfahren genutzt.

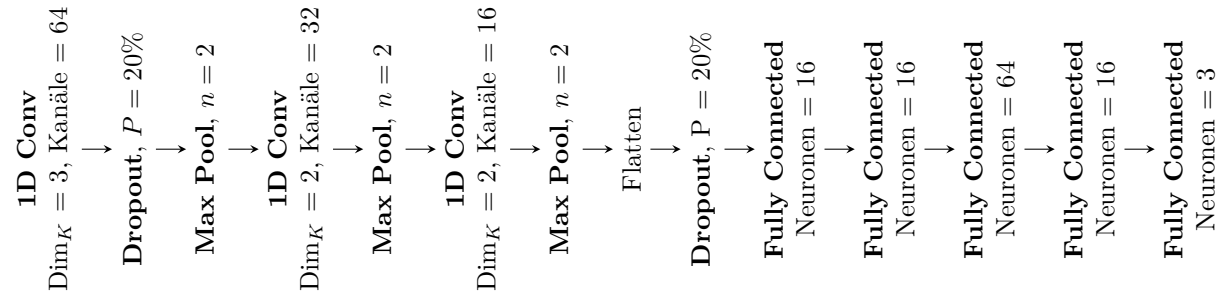


Abbildung 11: Struktur von unserem aktuell besten neuronalen Netzwerk. Dim$_K$ steht für die Größe des Kernels, 1d Conv für eine 1-dimensionale Convolutional Schicht.

Nachdem wir dieses Teilziel erreicht hatten, wollten wir unser Framework mit Daten testen, die wir selber aufgenommen hatten. Dafür haben wir zuerst das EEG von OpenBCI genutzt, da unser eigenes noch nicht fertig war. Da wir noch keine Zeit dafür hatten, konnten wir nicht genug Trainingsdaten aufnehmen, um das Netzwerk effektiv trainieren zu könnnen, da hierfür viele Datensätze benötigt werden. Also haben wir die EEG-Daten von Sentdex zum Trainieren und unsere selbst aufgenommenen zum Testen verwendet und

konnten tatsächlich eine Testdatengenauigkeit von bis zu 43% erreichen (s. Abb. 10). Dies ist zwar weit von ideal, aber dennoch sehr beeindruckend, da die Bedingungen von Trainings- und Testdaten so verschieden sind: Die Daten wurden mit verschiedenen Geräten, von verschiedenen Personen verschiedenen Alters auf verschiedene Weise aufgenommen. Dies unterstützt unsere Vermutung, dass es auch mit unserer Hardware möglich ist ein BCI zu entwickeln, dass Gedanken an verschiedene Richtungen sicher erkennen kann, da das neuronale Netzwerk bereits ein Kern-Muster, welches wirklich nur mit dem Gedanken zusammenhängt, gefunden hat.

5 Diskussion

Ziel unseres Projektes war es, ein Framework zu entwickeln, welches sich leicht auf die eigenen Zwecke anpassen lässt und die Entwicklung von BCIs vereinfacht. Dies haben wir mit unserem eigenen Package erreicht, da es Entwicklung in einer stark abstrahierten Form ermöglicht. Wir arbeiten außerdem momentan an einer ausführlichen Dokumentation mit Beispielen und Anleitungen und, anders als letztes Jahr, versuchen wir alle Funktionen, die wir programmieren, mit Beschreibung, Argumenten etc. zu dokumentieren, siehe unsere Dokumentation [17].

Wir haben zwei Ziele für die direkte Zukunft. Das erste hat mit unserem EEG zu tun: Wir wollen die Qualität der Daten unseres eigenen EEGs testen und diese sowie die erreichbare Genauigkeit des neuronalen Netzwerks mit den kommerziellen Versionen vergleichen. Zweitens wollen wir die Standardwerte verbessern und neue Funktionen implementieren, da wir zwar eine 44,57% Testdatengenauigkeit erreichen konnten, aber wissen, dass 60% möglich sind.

Eine wichtige Funktion, die sowohl uns beim Verbessern der Standardwerte als auch Nutzern stark helfen würde, wäre eine Funktion zur Hyperparameter-Optimierung. Hyperparameter sind bei neuronalen Netzwerken Parameter, die vor dem Trainieren festgelegt werden müssen, wie zum Beispiel die Netzwerkstruktur, die Velust- und Aktivierungsfunktionen, die Lernrate, die Regularisierungsfunktionen, Pruning, etc. Eine Hyperparameter-Optimierung kann automatisch gute Hyperparameter finden, indem sie systematisch verschiedene Konfigurationen ausprobiert und bessere entwickelt. Eine Hyperparameter-Optimierung würde uns helfen, da wir so schneller bessere Standardwerte finden könnten. Denn das *Trial and Error* Verfahren, vor allem per Hand, ist sehr zeitaufwändig – inzwischen haben wir vermutlich schon hunderte verschiedene Konfigurationen manuell erstellt und ausprobiert. Außerdem würde ein Hyperparamter-Optimierung es erlauben, dass unser Framework in einer noch größeren Anzahl an Szenarien ohne manuelle Anpassung benutzt werden könnte.

Wir haben bereits eine Idee für ein solches Verfahren, welches eine evolutionäres Vorgehensweise nutzen würde: Das Programm würde unsere Standardkonfiguration (s. Unsere Implementation) als Basis nehmen, diese mehrfach kopieren und zufällig abändern, und dann alle Kopien für z.B. 20 Sekunden trainieren. Danach würde es sich die beste Konfiguration auswählen, diese wieder kopieren und zufällig verändern und den ganzen Prozess erneut beginnen. Doch wir wollen uns noch weiter mit den verschiedenen möglichen Verfahren beschäftigen und müssen die Hyperparameter-Optimierung natürlich noch implementieren und testen.

Weiter in der Zukunft, nachdem wir hoffentlich mit unserem aktuellen Anwendungsbeispiel eine zufriedenstellende Genauigkeit erreichen konnten, werden wir weitere Anwendungsbeispiele umsetzen und dafür Daten sammeln. Nur so können wir sichergehen, dass unser Framework am Ende nicht einfach nur für einen Zweck verwendbar ist (s. Unsere Implementation).

6 Danksagung

Danke an den Förderverein „Gesellschaft der Freunde des Gymnasium Eversten e.V.“ für die Finanzierung des EEG-Geräts. Ohne diese Hilfe wäre dieses Projekt nie zustande gekommen. Alle finanzierten Teile sind in der Materialliste mit ⋆ gekennzeichnet.

Größter Dank geht an unseren Projektbetreuer Herr Dr. Glade, der unser Projekt begleitet hat. Er hat uns bei der wissenschaftlichen Methodik geholfen und uns wichtige Ressourcen zur Hand gegeben sowie bei Fragen weitergeholfen.

7 Quellen

7.1 Abbildungen

Abb. 8 und 9 Kopiert und abgeändert von `https://tikz.net/conv2d/`, ursprünglicher Author ist Janosh Riebesell

Abb. 1 Linkes Bild: `www.conrad.de/de/ratgeber/handwerk-industrie-wiki/elektronik-bauteile/lm393.html`, rechtes Bild: `https://www.reichelt.de/operationsverstaerker-2-fach-dip-8-lm-358-dip-p10483.html`

Abb. 3 Kopiert, ursprünglich erstellt von Józef Król, siehe `https://docplayer.pl/48121857-Szybkie-metody-projektowania-filtrow-aktywnych.html`

Alle anderen Abbildungen wurden selbst erstellt.

7.2 Literatur

[1] Ujwal Chaudhary, Niels Birbaumer und Ander Ramos-Murguialday. „Brain-computer interfaces for communication and rehabilitation“. In: *Macmillan Publishers Limited* 12 (Sep. 2016), S. 513–525.

[2] Alexander Reimer und Matteo Friedrich. „Erkennung ereigniskorrelierte Potenziale eines Elektroenzephalogramms durch eine KI“. 2. Apr. 2022. URL: `https://github.com/AR102/Interpreting-EEG-with-AI/blob/40ea994780d21ad324c6161d2a361584c1976b01/paper/paper.pdf` (besucht am 06. 01. 2023).

[3] Markus Feser. „Physik der Elektroenzephalographie“. Konzeption eines Schülerforschungstages. 1. Okt. 2013. URL: `https://www.physik.uni-wuerzburg.de/fileadmin/11010700/_imported/fileadmin/11010700/Didaktik/Zulassungsarbeiten/Schriftliche_Hausarbeit_Feser.pdf` (besucht am 11. 01. 2023).

[4] Universität Lübeck. „Elektroenzephalographie (EEG) und Ereigniskorrelierte Potentiale (EKP)“.

[5] N. Birbaumer und R. F. Schmidt. „Biologische Psychologie“. In: Springer Verlag, Berlin Heidelberg, 2010. Kap. 20.5, S. 468–493.

[6] Wikipedia. „Berger-Effekt — Wikipedia, The Free Encyclopedia“. `http://de.wikipedia.org/w/index.php?title=Berger-Effekt&oldid=208486396`. 2022. (Besucht am 29. 03. 2022).

[7] Dieter Nührmann. „Das große Werkbuch der Elektronik-Band 3“. In: Franzis' Verlag, Poing, 1998. Kap. 3.11, S. 1850–1869.

[8] cah6. „DIY EEG (and ECG) Circuit“. 2019. URL: `https://www.instructables.com/DIY-EEG-and-ECG-Circuit/`.

[9] „Low Cost, Low Power Instrumentation Amplifier“. In: (1999). URL: `https://pdf1.alldatasheet.com/datasheet-pdf/view/48091/AD/AD620AN.html`.

[10] Norbert Lieven. „Doppel-T-Filter, Kerbfilter, Notchfilter“. 2014. URL: `www.electronicdeveloper.de/FilterAktiv_Doppel_T.aspx`.

[11] „Passiven Tiefpass 1. und 2. Ordnung berechnen“. Funktionsweise, Formel, Tiefpass Rechner. URL: `https://electronicbase.net/de/tiefpass-berechnen/#grenzfrequenz-tiefpass-berechnen`.

[12] Detlef Mietke. „Filter mit Mehrfachgegenkopplung“. URL: `https://www.elektroniktutor.de/analogverstaerker/aktivflt.html`.

[13] Microchip Technology Inc. „MCP3204/3208“. In: (2008). URL: `https://www.microchip.com/en-us/product/MCP3208`.

[14] Wikipedia. „Tangens hyperbolicus und Kotangens hyperbolicus“. 3. Jan. 2023. URL: http://de.wikipedia.org/w/index.php?title=Tangens%5C%20hyperbolicus%5C%20und%5C%20Kotangens%5C%20hyperbolicus&oldid=229440694 (besucht am 15. 01. 2023).

[15] „Activation Functions in Neural Networks“. 6. Sep. 2017. URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6 (besucht am 14. 01. 2023).

[16] „Activation Functions: Why 'tanh' outperforms 'logistic sigmoid'?“ 23. Dez. 2019. URL: https://mvschamanth.medium.com/activation-functions-why-tanh-outperforms-logistic-sigmoid-3f26469ac0d1 (besucht am 14. 01. 2023).

[17] Alexander Reimer und Matteo Friedrich. „BCIInterface“. Englisch. 2023. URL: https://ar102.github.io/Interpreting-EEG-with-AI (besucht am 06. 01. 2023).

7.3 GitHub-Repositories und Videos

[18] Harrison Kinsley „Sentdex“. „BCI“. 2023. URL: https://github.com/Sentdex/BCI (besucht am 06. 01. 2023).

[19] Brotcrunsher. „Neuronale Netze - Backpropagation - Forwardpass“. 2017. URL: https://www.youtube.com/watch?v=YIqYBxpv53A (besucht am 15. 01. 2022).

[20] Alexander Reimer und Matteo Friedrich. „BCIInterface“. Version 0.0.1. 2023. URL: https://github.com/AR102/Interpreting-EEG-with-AI (besucht am 06. 01. 2023).