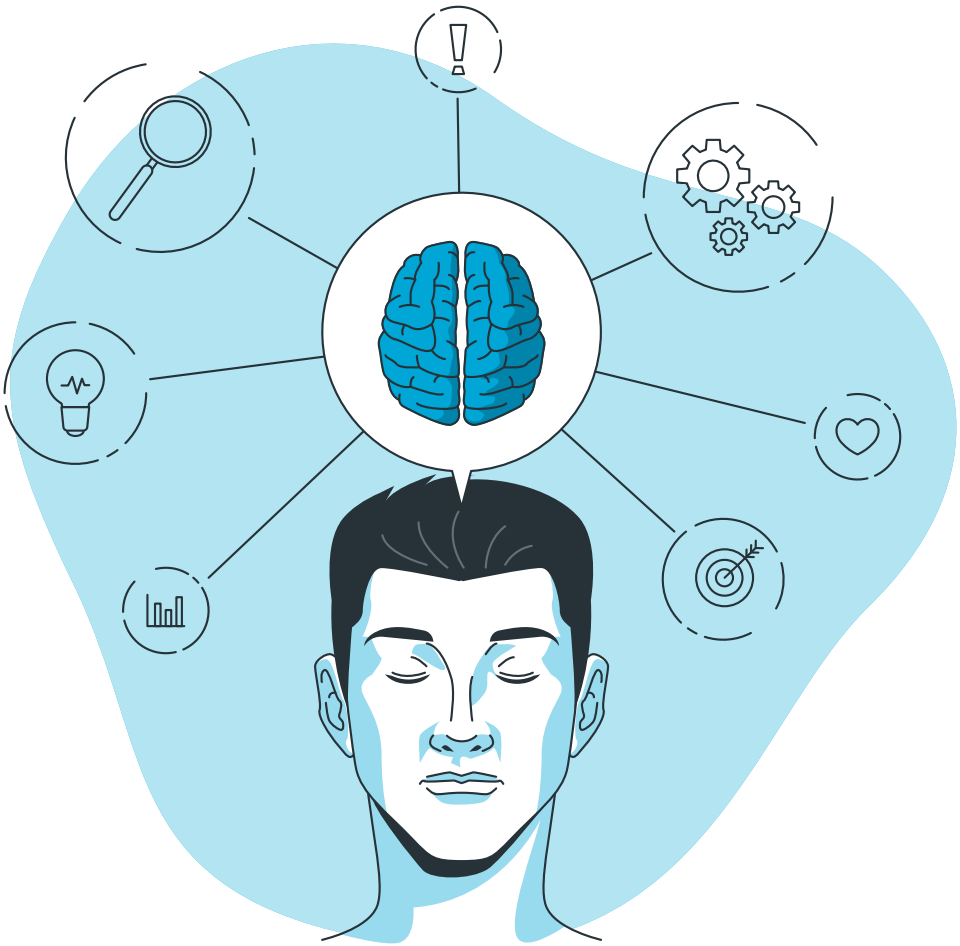
EEG-Based Brain Computer Interface

Decoding: A Deep Learning Approach

V.J. van der Doorn
P. Srivastava



EEG-Based Brain Computer Interface

Decoding: A Deep Learning Approach

by

V.J. van der Doorn
P. Srivastava

to obtain the degree of Bachelor of Science in Electrical Engineering
at the Delft University of Technology.

Students:	V.J. van der Doorn (5557577)
	P. Srivastava (5542707)
Thesis Committee:	Dr. B. Hunyadi (supervisor)
	Dr. F. Fioranelli (jury chair)
	Dr. Ir. M. Fieback

Cover: 'Mind map Cartoon Illustrations' by Storyset, free for personal and commercial purpose with attribution.
Style: TU Delft Report-Thesis Template by Daan Zwaneveld (with additional modifications by Pragun Srivastava).

TU Delft

Abstract

This thesis details the theoretical background and development process of a classification model for electroencephalogram-based (EEG) motor imagery (MI) signals, to be used in a brain-computer interface (BCI) system. This project was undertaken in order to demonstrate the possibility of distinguishing MI-EEG signals acquired using the *g.tec Unicorn Hybrid Black* EEG measurement cap. The classification model devised in this thesis is a hybrid deep neural network model, which combines a convolutional neural network (CNN) and long short-term memory (LSTM) recurrent neural network (RNN) in parallel, closing with a fully-connected (FC) layer. Much experimentation and research was needed to create this model, and this is extensively discussed within the thesis. The classification model produced for this thesis is one part of a complete end-to-end BCI system, which also entails a measurement and data processing procedure, and the development of a graphical user interface which provides visual feedback to the user.

On publicly available datasets, the classification model produced promising performance, achieving 55 % on the BCI Competition IV-2a (4 class) dataset and 78 % on the BCI IV-2a (2 class) dataset. The model has not yet been tested on self-collected data.

Preface

This thesis has been written in order to obtain the Bachelor of Science in Electrical Engineering at the Delft University of Technology. In this thesis, a classification model for MI-EEG signals has been created, as part of a larger project to develop a BCI system. BCI technology has the potential to improve the lives of many around the world, so we are thrilled to have had the opportunity to work on this project. Moreover, this project has been a great learning experience, allowing us to advance our knowledge and skills in the areas of artificial intelligence and signal processing.

We would like to express our sincere gratitude to our supervisor Dr. Bori Hunyadi for her support and guidance throughout this project. Furthermore, we would like to give our thanks to Prof. Dr. Ir. Leon Abelmann for helping us and providing us with feedback during the project. Last but not least, we would like to thank our fellow team members, Tijn Bakkum, Davi Spiller Beltrao, Jack Chen and Martin Little for the hardwork and dedication over the past 10 weeks.

V.J. van der Doorn
P. Srivastava
Delft, June 2024

Contents

Abstract i

Preface ii

1 Introduction 1
- 1.1 Basics of BCI technology . . . 1
- 1.2 State-of-the-art analysis . . . 1
- 1.3 Goal definition . . . 2
- 1.4 Structure of thesis . . . 2

2 Program of requirements 3
- 2.1 General requirements . . . 3
- 2.2 Decoding requirements . . . 4

3 Basics of neural networks 5
- 3.1 Artificial neurons . . . 5
- 3.2 Neural networks . . . 5
- 3.3 Deep learning . . . 6
- 3.4 Activation functions . . . 7
- 3.5 Training neural networks . . . 8
- 3.6 Loss functions . . . 9
- 3.7 Optimizers . . . 10

4 Neural network architectures 12
- 4.1 Convolutional neural networks . . . 12
- 4.2 Recurrent Neural Networks and LSTM . . . 15
- 4.3 Attention and self attention . . . 17
- 4.4 Kolmogorov-Arnold networks . . . 18

5 Model design and development 19
- 5.1 Algorithm approach . . . 19
- 5.2 Algorithm design . . . 20
- 5.3 Training . . . 22
- 5.4 System integration . . . 23

6 Results 25
- 6.1 Testing definitions . . . 25
- 6.2 BCI IV-2a . . . 25
- 6.3 Our dataset (static) . . . 26
- 6.4 Complications . . . 26

7 Conclusion 29
- 7.1 Conclusion . . . 29
- 7.2 Discussion . . . 29
- 7.3 Future work . . . 30

References 31

A Code 33
- A.1 Model . . . 33
- A.2 B-spline activation . . . 35
- A.3 filtering . . . 36
- A.4 Training Loop . . . 38

1 Introduction

1.1. Basics of BCI technology

A brain-computer interface (BCI) is a system that acquires and interprets brain signals, translating them into commands for computers or other external devices. BCIs enable direct communication between the brain and external devices, without the use of peripheral nerves or muscles. Such technology can provide immense benefit to individuals with disabilities, allowing locked-in or physically limited patients to interact with the world in ways that would otherwise be impossible. In addition to this, BCI technology has potential applications a range of areas, including robotics, education, entertainment, and neuroscience research.

The brain signals required for a BCI can be obtained through various techniques, amongst which electroencephalography (EEG) is the most common, primarily because it is non-invasive and cost-effective. EEG measurements are acquired by applying electrodes to the users scalp, often combined into a single cap [1]. Within this thesis, the EEG cap of choice is the *Unicorn Hybrid Black* by *g.tec* [2], which from now on may be referred to simply as 'the cap'.

One way a user of a BCI can interact with the system is through motor imagery (MI), in which the individual imagines moving a particular body part, without performing any actual movement. These MI signals can be detected on an EEG, and the corresponding intended action can be determined. For example, in a BCI cursor control system, imagining moving one's right hand could correspond to the cursor moving rightwards. Thus, much of the research in the area of BCIs involves the development of signal processing (SP) or machine learning (ML) techniques to decode the intended movement or action from MI-EEG signals [1].

1.2. State-of-the-art analysis

Research in the area of BCIs has been ongoing for several decades. For most of this time, interest in the field has been limited due to the difficulty of classifying EEG signals. In recent years, however, improvements in machine learning (ML) have enabled EEG signal classification with much greater accuracy than previously possible [3].

Traditionally, developing an ML model for EEG signals has required three steps: preprocessing, feature extraction, and finally, classification. The preprocessing step involves signal filtering, artifact removal, and other techniques to clean/prepare the data. Feature extraction involves the identification of relevant features from the data for use in the classification model. For EEG signals, these features largely fall into one of three categories: time domain features (e.g. mean, variance, skewness), frequency domain features (e.g. FFT, wavelet transform, power spectral density), and spatial filtering (e.g. CSP). Regarding the classification step, common models that have been used for EEG classification include support vector machines (SVM) and linear discriminant analysis (LDA) [4].

Traditional ML models have produced mixed results, and are limited in their need for manual feature extraction. This manual process limits the analysis to a predefined set of features, and requires extensive domain-area knowledge in order to succeed. In contrast, deep learning (DL) models, a specific class of ML models, can automatically discover features from the data, eliminating the need for manual feature extraction. This capability makes DL a promising avenue for EEG signal classification [5].

Studies have shown promising results using DL techniques for MI-EEG classification. A landmark paper by Lawhern et al. in 2016 [5] was among the first widely cited works to devise a DL model for classifying EEG signals [3]. Since its publication, research in this area has grown exponentially.

Thus, within this thesis, DL techniques for classifying MI-EEG signals will be explored. Ultimately, the goal is to develop a DL model for a BCI system which is to be produced by our project group.

1.3. Goal definition

The goal of this project is develop a BCI system which *demonstrates the possibility of distinguishing MI EEG signals using the g.tec Unicorn Hybrid Black*. This goal statement was formed in agreement with the project supervisor. In order to achieve this goal, the project was divided amongst three subgroups; the preprocessing subgroup, the decoding group, and the interface group. The responsibilities of each subgroup can be defined as follows:

- Preprocessing subgroup: Recording, filtering, and cleaning data acquired from the cap.
- Decoding group: Developing a classification algorithm which can decode the intended movement from the MI-EEG signals captured by the preprocessing subgroup.
- Interface group: Developing a graphical user interface (GUI) which enables the user to train and test the BCI, and produces plots of the acquired data.

In order to develop a complete final product, which captures the data from the cap and translate this into an on-screen action, it is essential that the three subsystems work in cohesion. The interactions between the three subgroups are presented illustratively in Figure 1.1. In addition, organised collaboration is required between the six team members of the team.

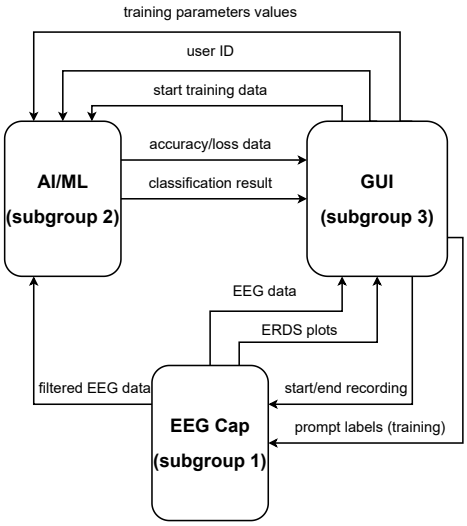


Figure 1.1: Overview of the whole system, divided into three subgroups, with the signals that are sent between them.

The plan for the decoding subgroup in this project is to devise an ML-based classification model. Initially, this model will be trained and tested on a publicly available MI-EEG data set, namely the BCI Competition IV 2a and 2b datasets, which can be found at [6]. Once the three subsystems of the project are close to completion, the model will be trained/tested upon data collected by the preprocessing group. Finally, the three subsystems will be integrated to form a complete BCI system, which is able to take in continuous incoming data from the preprocessing group and translate this into an on-screen actions on the interface.

1.4. Structure of thesis

The structure of this thesis will be divided as follows. In Chapter 2, the program of requirements for this project, both at the system-wide and sub-system level, will be defined. Chapter 3 will introduce the basics of neural networks, which form the basis of the classification model that will eventually be designed. Chapter 4 will expand on this with a detailed consideration of various neural network architectures. In Chapter 5, the process of designing and developing the classification model will be described, as well as the process for integrating the classification model with the remaining subgroups. In Chapter 6, the outcomes and results of the projects will be evaluated. Finally, Chapter 8 will provide some concluding thoughts, and provide a future outlook.

2 Program of requirements

As established, the goal statement for this project is to develop a BCI system which demonstrates the possibility of distinguishing MI-EEG signals using the g.tec Unicorn Hybrid Black. In order to achieve this goal, we have established a set of requirements of our product. These requirements define the necessary features of our final product for it to be considered successful. The requirements are divided in two categories: general requirements for the entire project, and requirements specific to our subsystem.

2.1. General requirements

Functional requirements

[A.1] The system must be able to decode MI-EEG signals to determine the intended movement, in accordance with general requirements B1 – B3.

[A.2] The system must be able to visualize the decoded MI-EEG signals through interactive demonstration(s).

[A.3] The system must be able to plot EEG data in real-time (note: we define a real-time plot as one which satisfies general requirement B4).

[A.4] The data handling performed by the system must be individualized to each user.

Performance requirements

[B.1] The minimal sensitivity of prediction for each of the 4 motor imagery actions separately must exceed 50%.

[B.2] The average overall accuracy of prediction across all 4 motor imagery actions must exceed 70%.

[B.3] The action latency (see definition below) must be less than 5 seconds.

[B.4] The plot latency (see definition below) must be less than 1 second.

Implementation requirements

[C.1] The data utilized by the system must be measured using the g.tec Unicorn Hybrid Black.

[C.2] All software developed for the system must be written in Python. This includes software for measurement, decoding, and visualization.

Definitions

- We define the *action latency* as the time between the moment at which the prompt is shown on the GUI, and the moment at which the corresponding classification is shown.
- We define the *plot latency* as the time between the measurement of a sample and its corresponding output to the plot.

2.2. Decoding requirements

Functional requirements

[A.1] The classification algorithm must be able to receive data from the preprocessing group, and decode the intended movements in accordance with general requirements B1 - B3.

[A.2] The classification algorithm must work locally on a Windows PC.

[A.3] For every sample received by the preprocessing group, the output of the algorithm must be a 1×4 vector, describing the probability of each class.

[A.4] The algorithm must be efficient enough that decoding requirement B1 is satisfied when running on a system with less than 4GB of VRAM.

Performance requirements

[B.1] The computational latency of classifications produced by the model must be less than 2.5 seconds.

[B.2] The accuracy of the classification algorithm when applied to the BCI Competition IV-2A dataset must exceed 80%.

[B.3] The accuracy of the classification algorithm when applied to data collected by the preprocessing subgroup must exceed 70%.

Implementation requirements

[C.1] The classification algorithm must be based machine learning methods.

[C.2] The algorithm must be implemented in Python using PyTorch and other associated libraries.

Definitions

- We define the *computational latency* as the time between the moment at which the sample is received by the classification model, and the moment at which the output classification vector is produced.

3 Basics of neural networks

Neural networks (NNs) are machine learning (ML) models which seek to mimic the structure and function of the human brain. NNs use training data to *learn* the patterns of a dataset, and once trained, are able to apply this learning to make predictions or classifications based on unseen input data. The unique capabilities of NNs make them well suited for a wide range of problems in artificial intelligence (AI) and signal processing (SP). In this section, the basic concepts and terminology of neural networks will be explored.

3.1. Artificial neurons

To understand how NNs work, it is first essential to understand *artificial neurons*, which are the fundamental building blocks of NNs. An artificial neuron is analogous to a biological neuron found in the brain, but is not a direct equivalent. Throughout this thesis, artificial neurons will be referred to simply as neurons, following ML convention.

A neuron receives n inputs $\boldsymbol{x}^{\mathrm{T}} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}$. Each input is received through a separate connection, with each connection having an associated weight. These weights form the vector $\boldsymbol{w}^{\mathrm{T}} = \begin{bmatrix} w_1 & w_2 & \dots & w_n \end{bmatrix}$. The neuron determines the weighted sum of its inputs by computing the dot product $\boldsymbol{x} \cdot \boldsymbol{w}$, and then adds a bias term b. The result of this computation, the *pre-activation* z, is passed through an activation function $g(z)$, producing the output (also called the *activation*) y of the neuron. This process can be represented by the expression in Equation 3.1. A neuron with 3 inputs and 3 corresponding weights, can be understood visually as shown in Figure 3.1 [7].

$$\text{output} = g(\boldsymbol{w} \cdot \boldsymbol{x} + b) \tag{3.1}$$

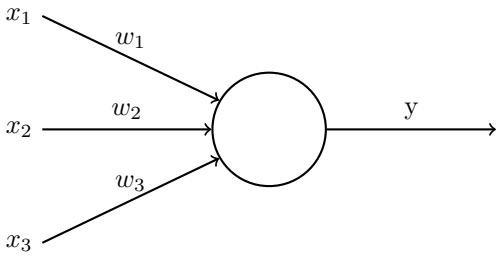


Figure 3.1: Diagram illustrating an artificial neuron.

3.2. Neural networks

A NN is a collection of interconnected neurons organised into groups known as *layers*. These layers are arranged in a sequential structure, where each layer is a function of the previous. A NN typically comprises at least three layers: an input layer, one or more hidden layers, and an output layer. The input layer receives and encodes raw data, and passes it on to the (first) hidden layer without performing any calculations. The hidden layers are the core of the network. They perform linear operations on their inputs, determine outputs by applying activation functions, and pass these outputs to the next layer. The output layer receives its inputs from the (last) hidden layer, and produces the final prediction or classification of the network [8]. When discussing NN architectures, models are mostly considered in terms of layers, rather than individual neurons.

NNs can vary in terms of their structure, including the arrangement and types of layers used, the type of activation function used, the number of neurons within each layer, the connections between

neurons, etc. These factors, together, are called the *architecture* of a neural network. For instance, consider the basic NN architecture given in Figure 3.2. This first example has two unique properties, chosen for sake of simplicity. Firstly, the NN is *feed-forward*, and secondly, all layers in the NN are *fully-connected*. More advanced network and layer types will be discussed in Chapter 3. Feed-forward means that information in the network always flows in one direction. All connections go from one layer to the next, but never back to the same or previous layer. This is in contrast to *recurrent* NNs, where feedback loops are possible. A layer being fully-connected means that all neurons in the layer connect to all neurons in the next layer. This is not the case for all layer types, for example, in the case of *convolutional* layers, which form the basis of convolutional neural networks.

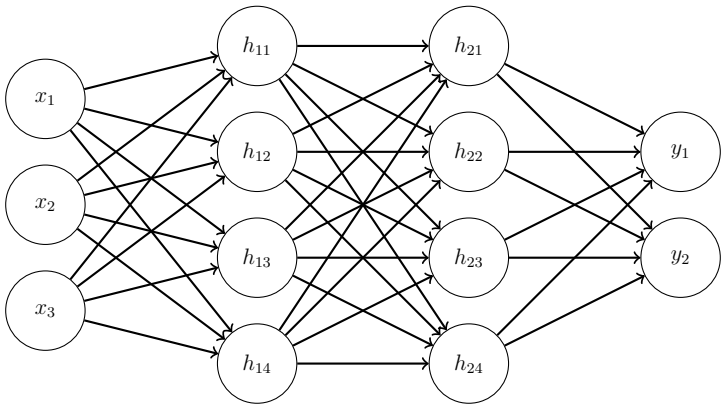


Figure 3.2: A basic neural network architecture with one input layer, two hidden layers, and an output layer.

Mathematically, the layers of a neural network can be described using vectors/matrices of their constituent neurons. For example, in Figure 3.2, the first hidden layer $\boldsymbol{h}^{(1)}$, can be expressed as:

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right) \tag{3.2}$$

In Equation 3.2, $\boldsymbol{x}$ is a column vector containing the values of the input layer, and $\boldsymbol{W}$ is a matrix of weights. Each row of $\boldsymbol{W}$ corresponds to the connections between the input layer and a specific neuron in $\boldsymbol{h}^{(1)}$. The product of $\boldsymbol{W}$ and $\boldsymbol{x}$ results in a column vector, in which each element is the weighted sum for a particular neuron in the hidden layer. Additionally, the bias terms for each neuron are combined into a bias vector $\boldsymbol{b}$, which is then added to the product of $\boldsymbol{x}$ and $\boldsymbol{W}$. The outcome of this operation is passed through an activation function g, which applies element-wise to each neuron in the vector. This produces the vector $\boldsymbol{h}^{(1)}$, representing the activations of the first hidden layer. Similarly, the activations of the second hidden layer can be expressed as a function of the first hidden layer [9]:

$$\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right) \tag{3.3}$$

3.3. Deep learning

A NN with a single hidden layer is considered *shallow*, while one with multiple hidden layers is considered *deep*. Deep NNs form a distinct domain in the field of ML called *deep learning* (DL) [8]. Through the use of multiple hidden layors, deep NNs enable *representation learning*, allowing the network to automatically find relevant features given raw data. This eliminates the need for manual feature extraction, which is typically required in traditional ML approaches. This is particularly useful for applications where manual feature extraction is exceptionally challenging or unintuitive for humans. Moreover, the representation learning performed by NNs is hierachial, meaning that features are found at increasing levels of abstraction as the data passes through the layers of the network. Layers earlier in the network find lower level features, while those further on the network find higher level, more abstract features. The advantages of representation learning make deep NNs an invaluable tool in numerous disciplines, such as image recognition and signal processing [10].

Existing literature has demonstrated successful classification of bio-electrical signals such as MI-EEG via DL approaches. As such, deep NNs will be the primary area of investigation within this project [3] [4] [5], and the classification algorithm developed in this thesis will be a DL model.

3.4. Activation functions

As mentioned in section 3.1, the output of a neuron is determined by passing the computed weighted sum through an activation function (AF). There are several common AFs in the field of ML/DL, each with its own (dis)advantages. In addition, AFs often have an ideal compatibility of cost functions and optimizers they pair particularly well with (these concepts will be introduced shortly). This section will cover some basic, commonly-used AFs, as well as a more novel approach to AFs called B-Splines.

3.4.1. Basic activation functions

- Sigmoid function

$$g(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.4}$$

The sigmoid function maps any real number to the output range $[0, 1]$, and has a smooth gradient. However, a disadvantage is that is suffers from the vanishing gradient problem – the gradient becomes very small for large input values, leading to an almost negligible updates during training of a NN.

- Hyperbolic tangent (tanh)

$$g(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3.5}$$

The hyperbolic tangent function maps real numbers into the range $[-1, 1]$. It has a similar shape to the sigmoid function, but brings the outputs closer to zero, which enables better convergence. This AF also suffers from the vanishing gradient problem.

- Rectified Linear Unit (ReLU)

$$g(x) = \max(0, x) \tag{3.6}$$

ReLU is a piecewise function which outputs the input if the input is positive, and outputs zero otherwise. The advantages of ReLU are its computational efficiency, as well as its mitigation of the vanishing gradient problem. A limitation is the 'dying ReLU problem'. This occurs when the neuron receives negative inputs only, causing the output to be zero. As a result, the neuron becomes inactive, preventing any future gradient updates from taking place.

- Leaky ReLU

$$g(x) = \begin{cases} \alpha x & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{3.7}$$

The Leaky ReLU function is a modification of the standard ReLU, which allows a small, non-zero gradient when the inputs are negative. This prevents the dying ReLU problem from occurring. The hyperparameter α must be set manually. This can be seen as an advantage, since it provides additional flexibility. It can also be seen as a disavantage, since it can be difficult to choose what value to set [11].

- Swish

$$g(x) = x \cdot \sigma(x) \tag{3.8}$$

Swish is a relatively new AF established by Google Brain researchers in attempt to create a standardised function to compete with ReLU and its variants. It is defined as the product of x with $\sigma(x)$, the sigmoid of x. It is more computationally expensive, and occasionally more performant than ReLU [12].

- Softmax

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} \tag{3.9}$$

The softmax AF transforms an input vector of values $z = [z_1, z_2, \ldots, z_n]$ into an output vector of probabilities, ensuring that the outputs are in the range $[0, 1]$ and sum to 1. Each element of the output vector is calculated as shown in Equation 3.9. Each element of the input vector z is exponentiated, and then divided by the sum of all exponentiated values as a normalization [13]. As a result of its probabilistic nature, softmax is often used in the output layer of networks built for multi-class classification tasks. A disadvantage of the softmax AF is its high computational expense.

3.4.2. B-Splines

A spline is a piecewise polynomial function of degree n, most commonly used in computer graphics and computer-aided design (CAD). However, splines are also well-suited for modelling arbitrary functions. The spline function maps values from an interval $[a, b]$ to the set of real numbers $\mathbb{R}$.

$$S : [a, b] \to \mathbb{R} \tag{3.10}$$

The interval [a,b] is divided into a number of sub-intervals by a knot vector T.

$$T = \{t_0 \leq t_1 \leq ... \leq t_k\} \tag{3.11}$$

$$[a, b] = [t_0, t_1) \cup [t_0, t_1) \cup .... \cup [t_{k-1}, t_k) \cup [t_k] \tag{3.12}$$

$$a = t_0 \leq t_1 ... \leq t_{k-1} \leq t_k = b \tag{3.13}$$

on each of these sub-intervals a polynomial is defined.

$$P_i : [t_1, t_{i+1}] \to \mathbb{R} \tag{3.14}$$

The spline function is then defined as:

$$S = P_i(t), \quad t_1 \leq t < t_{t+1} \tag{3.15}$$

A B-spline, also called a basis spline, is a variant of the normal spline function. It has the key property that they and their derivatives may be continuous(which is needed for back-propagation in machine learning). B-splines are defined as

$$S_{n,t}(x) = \sum_i \alpha_i B_{i,n}(x) \tag{3.16}$$

where α is a learnable scaling factor which allows the modulation of the curve and $B_{i,n}$ is a piecewise polynomial function derived by means of the Cox-de Boor formula which is defined as follows:

$$\begin{cases} 1 & if\ t_i \leq x \leq t_{i+1} \\ 0 & otherwise \end{cases} \tag{3.17}$$

$$B_{i,n}(x) := \frac{x - t_i}{t_{i+k} - t_i} B_{i,n-1}(x) + \frac{t_{i+n+1} - x}{t_{i+n+1} - t_{i+1}} B_{i+1,n-1}(x) \tag{3.18}$$

3.5. Training neural networks

The training process for neural networks (NNs) involves three key components: a cost function, the backpropagation ('backprop') algorithm, and an optimizer.

The cost function, or loss function, quantifies the difference between the neural network's predicted outputs and the actual target values. It serves as a measure of the model's performance. Common cost functions include mean-squared error for regression tasks and cross-entropy loss for classification tasks. Common loss functions for classification problems are detailed in Section 3.6. The objective of training a neural network is to minimize the cost function, thereby improving the accuracy of the network's predictions.

The backprop algorithm is used to compute the gradient of the cost function with respect to all weights and biases in the network[1]. Fundamentally, backprop relies on the chain rule of calculus. It involves two steps: the forward pass and the backward pass. During the forward pass, input data is passed through the network to produce an output, which is then used to calculate the cost function. In the backward pass, the gradients are calculated by propagating the cost function backwards through the network. These gradients indicate how much the weights and biases need to be adjusted in order to reduce the cost function.

The optimizer uses the gradients, as calculated by the backprop algorithm, to update the network's weights and biases such that the cost function is minimized. The goal is to find the global minimum of

[1] From now on, the term *gradient* in the context of training NNs will specifically refer to the gradient of the cost function with respect to the network parameters.

the cost function in parameter space, or at the very least a good local minimum. Several optimization algorithms exist, each with its advantages. Common optimizers include gradient descent and ADAM, which are discussed in detail in section 3.7.

With these three concepts, the training of NNs can be understood. The network starts with an arbitrary[2] set of weights and biases, and the first training sample is received. The forward pass then takes place, and the cost function is computed. The cost function is spread through the networks in the backward pass. Finally, the optimizer uses the gradients obtained through backgrop to update the weights and biases of the network. This entire process is repeated for every sample until the training set is exhausted, with the intent to find the optimal combination of weights and biases such that the cost function is minimal.

3.6. Loss functions

This section will discuss two loss functions. One of these, cross-entropy loss is a standard, commonly-used loss function. Meanwhile, the other loss function, triplet loss is a more novel approach, which has been used a lot in computer vision and has recently gained popularity in the field of natural language processing (NLP). As far as is known by the authors, this technique has little published research for MI-EEG signal applications.

3.6.1. Cross entropy loss

Cross entropy loss is one of the most commonly used functions for classification problems in machine learning. It is defined as:

$$L = -\sum_{i=1}^{N} y_i log(p_i) \tag{3.19}$$

Where N is the number of classes y_i is the true probability distribution and p_i is the predicted probability for class i. Cross entropy loss penalizes the model based on how wrong or right its prediction is.

3.6.2. Triplet loss and embeddings

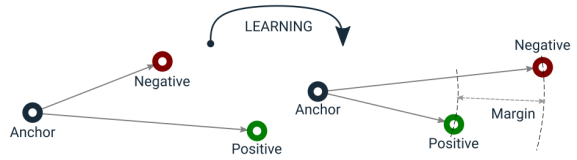


Figure 3.3: Triplet loss

Instead of a model outputting a classifying probability vector for a set of N categories, it can also have it output a vector which embeds the output information the model gives into an N-dimensional Euclidean space. Additionally this Euclidean space is constrained to an n-dimensional hypersphere. A loss function can be defined for this hypersphere.

$$\mathcal{L}(x_i^a, x_i^p, x_i^n) = max(||f(x_i^a) - f(x_i^p)||_2 - ||f(x_i^a) - f(x_i^n)||_2 + \alpha, 0) \tag{3.20}$$

In this loss function, a reference input, also called the anchor vector(x_i^a), is compared to a matching input/positive vector(x_i^p), and to a non-matching vector/negative vector(x_i^n). The loss is then calculated by calculating the distance from the anchor vector to both the Positive and Negative vector. From this a loss is computed and added with a constant margin α. The loss enforces the model to minimize the distance between the Anchor and Positive vector while maximizing the distance between the Anchor and Negative vector. With this loss we ensure that a specific motor imagery task x_i^a is closer to all other samples of that motor imagery task x_i^p than to other different motor imagery tasks x_i^n.

[2]In practice, there are ways to effectively choose these starting values. However, this is beyond the scope of this thesis.

3.7. Optimizers

In this section, several optimizers will be introduced. These optimizers are spread across two categories: gradient descent methods, and Adam (**Ada**ptive **m**oment estimation).

3.7.1. Gradient descent

In gradient descent, the parameters ϕ are updated according to the following rule, where α is a hyperparameter called the *learning rate*, and L is the loss function:

$$\boldsymbol{\phi} \longleftarrow \boldsymbol{\phi} - \alpha \cdot \frac{\partial L}{\partial \boldsymbol{\phi}} \tag{3.21}$$

In order to find the global minimum of the loss function using gradient descent, it would be necessary to either search through the entire parameter space, or repeatedly start the algorithm from different positions until a sufficiently low loss is acheived. However, especially for large NNs, both of these approaches are highly ineffecient. Stochastic gradient descent (SGD), which follows the update rule in Equation 3.22, seeks to address this problem.

$$\boldsymbol{\phi}_{t+1} \longleftarrow \boldsymbol{\phi}_t - \alpha \cdot \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i \left[\boldsymbol{\phi}_t\right]}{\partial \boldsymbol{\phi}} \tag{3.22}$$

In SGD, a random subset of the training data called a *batch* is selected within each iteration. For every item i in the batch $\mathcal{B}_t$, the loss function ℓ_i is determined, and the gradient is taken. Then, the gradients are summed, and the network parameters are adjusted accordingly. This process is repeated for every batch. The batches are drawn at random without replacement, until a complete pass of the training dataset is achieved. One such pass is called an *epoch*. The batch size, as well as the number of epochs are hyperparameters defined by the user. By performing calculations with only a subset of the training data, SGD is less computationally expensive. Furthermore, by updating the parameters in smaller steps, a sensible minimum can be found more easily.

An often used modification to SGD is the inclusion of a *momentum* term, which enables faster convergence and reduces the oscillations of the update. The momentum $\mathbf{m}$ calculates a moving average of the gradients, incorporating past gradient information to guide the current updates. The update rule for the momentum is given in Equation 3.23, where $\beta \in [0, 1)$ is the momentum coefficient, which decides in what ratio to consider the past and current gradients. The update rule for the parameters is then given by Equation 3.24 [8].

$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1 - \beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i \left[\boldsymbol{\phi}_t\right]}{\partial \boldsymbol{\phi}} \tag{3.23}$$

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \mathbf{m}_{t+1} \tag{3.24}$$

In essence, in SGD with momentum, every parameter update takes into account not only the current gradient, but all past gradients. The momentum is calculated recursively such that the further back in time one goes, the less those gradients contribute to the current update.

3.7.2. Adam

Gradient descent methods are limited by the fact that adjustments to parameters are directly proportional to the size of their gradients. Parameters with small gradients see small adjustments, while those with large gradients see large adjustments. This proportional adjustment can lead to slow convergence, especially in deep neural networks where the scale of gradients can vary significantly. The Adam (**Ada**ptive **m**oment estimation) optimizer addresses this limitation by dynamically adjusting the learning rates for each parameter. The update rule for the Adam optimizer is as follows:

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}} + \epsilon} \tag{3.25}$$

To arrive at Equation 3.25, a specific process is followed. First, the momentum $\mathbf{m}_{t+1}$ and its pointwise squared derivative $\mathbf{v}_{t+1}$ are determined with the update rules below. Here, β is the momentum coefficient of $\mathbf{m}_{t+1}$, γ is the momentum coefficient of $\mathbf{v}_{t+1}$, α is the learning rate, and ϵ is a small constant used to prevent division by zero:

$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i \left[\phi_t\right]}{\partial \phi} \tag{3.26}$$

$$\mathbf{v}_{t+1} \leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma) \left(\sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i \left[\phi_t\right]}{\partial \boldsymbol{\phi}} \right)^2 \tag{3.27}$$

As mentioned, the momentum terms takes into account all past gradients. Initially, the $\mathbf{m}_{t+1}$ and $\mathbf{v}_{t+1}$ are unusually small because the previous gradients are close to zero. To address this, the momentum terms are modified with the following rules:

$$\tilde{\mathbf{m}}_{t+1} \leftarrow \frac{\mathbf{m}_{t+1}}{1-\beta^{t+1}} \tag{3.28}$$

$$\tilde{\mathbf{v}}_{t+1} \leftarrow \frac{\mathbf{v}_{t+1}}{1-\gamma^{t+1}} \tag{3.29}$$

Once these modifications are made, we can use the adjusted momentum terms $\tilde{\mathbf{m}}_{t+1}$ and $\tilde{\mathbf{v}}_{t+1}$ in Equation 3.25 to update the parameters. This approach allows the Adam optimizer to effectively handle varying scales of gradients, leading to more efficient and faster convergence compared to traditional gradient descent methods [8].

While Adam is popular given its efficiency and adaptive learning rate, it faces limitations due to the ineffectiveness of L2 regularization[3] on adaptive gradient methods. Specifically, L2 regularization (also called weight decay) does not work well on Adam as it scales the gradient of both the loss function and the regularizer by the learning rate. This results in insufficient regularization for parameters with large past gradients, causing poor generalization performance. This essentially means that even when L2 regularization is utilised, Adam is likely to overfit.

AdamW (abbreviation of 'Adam with weight decay') addresses this limitation by separating the weight decay from the gradient-based update. To do this, the weight decay is subtracted from the parameters when an update takes place. This ensures that weight decay acts purely as a regularization term, without impacting the adaptive learning rates. The update rule of AdamW is the following, where λ is a hyperparameter called the weight decay coefficient [14]:

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \left(\frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}} + \epsilon} + \lambda \boldsymbol{\phi}_t \right) \tag{3.30}$$

In this section, several optimizers have been discusses, each seeking to improve or address a limitation of the previous, culminating in AdamW. By addressing the limitations of its predecessors, AdamW represents the most advanced optimizer amongst those covered. Hence, AdamW will be the choice of optimizer in the classification algorithm that will eventually be designed.

[3] Regularization techniques in ML are a group of methods used to enhance the generalizability of a model, i.e. reduce the likelihood of overfitting. The details of regularization techniques are beyond the scope of thesis. As such, we refer to Chapter 9 of [8] for further information on this topic.

4 Neural network architectures

In the previous chapter, a simplistic NN architecture was considered. This chapter will explore three more advanced architectures. First, convolutional neural networks (CNNs) will be discussed. Next, recurrent neural networks (RNNs), including a particular type of RNNs called Long Short-Term Memory networks (LSTMs), will be described, as well as the attention mechanism. Finally, a newer and more novel architecture, Kolmorogov-Arnold networks (KANs), will be introduced.

4.1. Convolutional neural networks

A CNN is a type of deep NN specifically designed for high dimensional data with spatial dependencies. As a result, CNNs perform exceptionally well in image recognition, computer recognition, and other signal processing applications. The architecture of a CNN typically consists of three layer types; convolutional layers, pooling layers, and fully-connected layers.

4.1.1. Convolutional layers

A convolutional layer transforms an input matrix $\mathbf{x}$ into an output matrix $\mathbf{z}$, where each element of $\mathbf{z}$ is computed as the weighted sum of a local region of elements in $\mathbf{x}$. This transformation is achieved by applying a *convolutional filter* to $\mathbf{x}$. A convolutional filter is a n-dimensional array of learnable weights that slides across the input matrix with a specified stride S. At each position, the filter performs a convolution[1] with the corresponding local region of $\mathbf{x}$, and a bias term is added, resulting in a single value of the output matrix. This process is repeated across all positions, generating the complete output matrix $\mathbf{z}$. Finally, the output matrix undergoes an activation function, and result is fed into the next layer.

In the case of a 1-dimensional (1D) input and filter, the i-th element of the output vector can be found by the convolution presented in Equation 4.1. In this expression, i represents the index of the output vector, and j is an index which iterates over the elements of filter from 1 to K, where K is the size of the filter. The activation is then found as given in Equation 4.2, where β is the bias term and g is the activation function. Visually, a 1D-convolutional filter can be understood as illustrated in Figure 4.1, which demonstrates the computation for determining the output element z_3. In this instance, the convolution is centered at x_3 with a filter size $K = 3$ and stride $S = 1$. After determining z_3, the filter moves S units down. Then, this process is repeated for z_4, z_5, and so on.

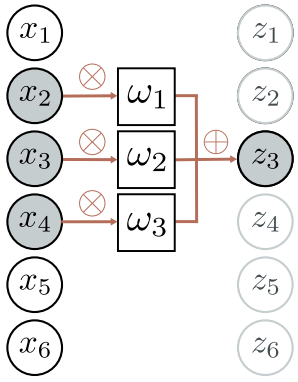


Figure 4.1: Illustration of a 1D conv. filter [8]

$$z_i = \sum_{j=1}^{K} w_j x_{i+j-K} \tag{4.1}$$

$$h_i = \mathrm{g}\left[\beta + z_i\right] \tag{4.2}$$

[1] In strict mathematical terms, the operation performed by a convolutional filter is a cross-correlation, not a convolution. However, we use the term convolution following ML convention.

Now consider a convolutional layer for 2D inputs with a filter size of 3×3, illustrated in Figure 4.2. Again, the filter slides over the input matrix, performing a convolution at each position. The computation performed by the filter in the 2D case is similar to the cross-correlation of two 2D discrete random variables, as shown in Equation 4.3. Just as the 1D case, once the filter is applied, a bias term is added, and the result is passed through an activation function. After performing the computations for one position, the filter moves to the right with stride S. This repeats until the whole row has been processed. Upon completing a row, the filter returns to the leftmost element of the next row, continuing this process until the entire input matrix has been processed [8].

$$h_{ij} = \mathrm{a}\left[\beta + \sum_{m=1}^{3}\sum_{n=1}^{3} \omega_{mn} x_{i+m-2,j+n-2}\right] \tag{4.3}$$

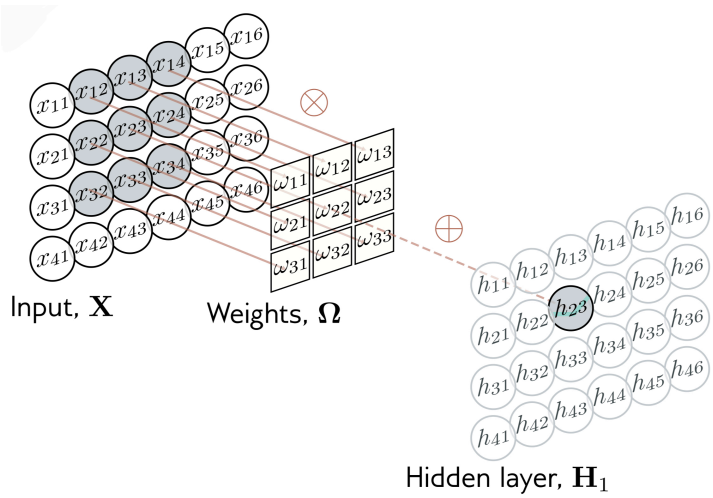


Figure 4.2: Diagram illustrating the computation performed in a 2D convolutional layer [8]

For both the 1D and 2D case, when the convolution is centered at an outer position of the matrix, parts of the filters go over the 'edge' of the input matrix. Zero padding is used to fill in these empty elements.

Note that 3D convolutional layers are also commonly used (in fact, they are used in our own model). However, these will not be elaborated upon, in light of their relative mathematical complexity and the fact that, conceptually, the same core principles apply.

4.1.2. Pooling layers

The next layer type critical in a CNN is a pooling layer, which is most commonly applied after a convolutional layer, but can in theory, also be applied individually. A pooling layer performs a downsampling operation in order to reduce the number of parameters, and correspondingly, the computational load, for subsequent layers. The pooling layer uses a sliding window which traverses the input matrix. The size, and stride, of this window are predefined hyperparameters. At each position in its traversal, a pooling operation is applied. This can either be a max pooling layer, which selects the maximum value of the elements within the window, or an average pooling layer, which computes the mean of the elements within the window. Max pooling is ideal for capturing the most prominent features within a local region, while average pooling is more effective at smoothing and reducing noise [8].

4.1.3. Full architecture

A CNN typically contains several convolutional and pooling layers in sequence. It is valuable to consider what particularly about these layer types, and their combination, makes them well-suited to high-dimensional, spatially-dependent data.

Convolutional layers use filters that slide over the input data, performing operations not only on individual elements but also on their surrounding local regions. This local interaction is crucial for

capturing spatial features, which often have local correlations. For instance, in image classification, features such as specific shapes or textures depend on region of pixels rather than individual pixels. Similarly, in the context of MI-EEG applications, the waveform of the signal is likely to have components which are internally interdependent.

A fundamental advantage of the convolution operations, which are performed by the filter, is their shift-invariance. This means that the filters can identify features consistently throughout the input, regardless of the position of these features. In image classification, an object should be identified regardless of which part of the image it is in. Similarly, MI-EEG signals contain multiple channels of input. The ability to recognise features in a similar way across different channels ensures the spatial features are captured successfully.

Pooling layers complement this by performing dimensionality reduction, keeping the most important spatial features while discarding the rest. This allows earlier layers to detect lower level features, while subsequent layers can focus on higher, more abstract features, preserving only the most significant outcomes of the previous layers. Additionally, by taking the maximum or average value within each local region, pooling layers can ensure that small distortions in the data do not impede the overall feature detection or classification.

Through these unique set of strengths, convolutional and pooling layers can be utilised to produce powerful models. However, these layers are not able to perform actual classification/prediction. To this end, a CNN usually closes with one or more standard, fully-connected layers to produce the final classification/prediction of the network.

4.1.4. Residuality

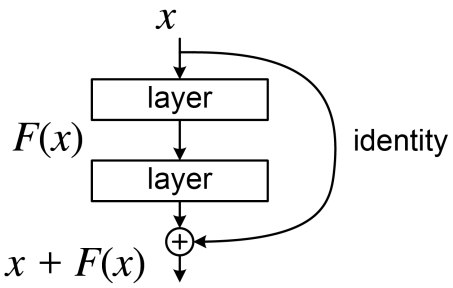


Figure 4.3: Residual learning block

When building very deep CNNs, problems start to appear. During training, instead of getting more accurate, the accurate stops at below that of a shallower CNN, or even starts to increase. This is due to the vanishing gradient problem – the gradients propogated through the network become increasingly small as they move through the layers, making it difficult for the network to correctly update its weights and biases.

The concept of residual learning for CNNs was first introduced in the paper 'Deep Residual Learning for Image Recognition' [15] which sought to combat the vanishing gradient problem. In residual CNNs, instead of a stack of layers fitting to a desired underlying mapping, the stack of layers fits a residual mapping. If we say that x is the input of the network and $\mathcal{H}(x)$ the identity mapping it needs to approach, then the residual function $\mathcal{H}(x) - x$ can be approached as well. To let a convolution block approach the residual function, a bypass is added for x to the output of the convolutional block. thereby getting $\mathcal{F}(x) + x$ as our output mapping. The function approaches the mapping $\mathcal{F}(x) + x := \mathcal{H}(x)$. By rewriting this function, the residual output $\mathcal{F}(x) := \mathcal{H}(x) - x$ is approached. Learning via this residual is both computationally less intense, and solves the vanishing gradient problem due to its skip connections.

4.2. Recurrent Neural Networks and LSTM

Feed-forward networks treat each input as independent of previous inputs, limiting their success in tasks where contextual or sequential information is relevant. This limitation is addressed by recurrent neural networks (RNNs), a specialized class of NNs designed for handling sequential data, such as time series or natural language. RNNs feature cyclical connections that allow hidden units to receive their own outputs from previous time steps as inputs for the current time step. This is enabled by a *hidden state*, which maintains a memory of the previous data seen by the network. The memory capabilities of RNNs are further enhanced through *parameter sharing*, meaning the same weights and biases are applied across all time steps. This ensures temporal consistency throughout the sequence and significantly reduces the number of parameters, making the analysis of sequential data more efficient.

RNNs are useful for classifying MI-EEG signal due to their ability to capture the temporal dependencies and patterns inherent to such data. RNNs are uniquely suited for time series analysis, and in fact, EEG signals *are* time series. In MI-EEG classification, it is crucial to consider not only the current time sample, but also the relationships with previous time samples.

RNNs are typically illustrated as shown in Figure 4.4 (left), where the self-directed connection at the hidden layer represents the hidden state. The diagram of an RNN can also be *unfolded*, as in Figure 4.4 (right), where the network is expanded into a series of feed-forward layers, the length of which is equal to the length of the input sequence. RNNs are trained using a modified backpropagation algorithm called Backpropagation Through Time (BPTT). In BPTT, the sequential network is unfolded, the gradients are then determined through standard backpropogation [9].

4.2.1. Standard RNN

Standard RNNs, such as those depicted in 4.4, consist of a single hidden state. At every time step, the hidden state is updated, and one item in the output sequence is produced. The equations governing the behavior of an RNN are shown in Equations 4.4 to 4.6. $\boldsymbol{b}$ and $\boldsymbol{c}$ are bias vectors corresponding to the hidden and output layers, respectively. The matrices $\boldsymbol{U}, \boldsymbol{V}$ and $\boldsymbol{W}$ denote the weights of the connections between the different layers.

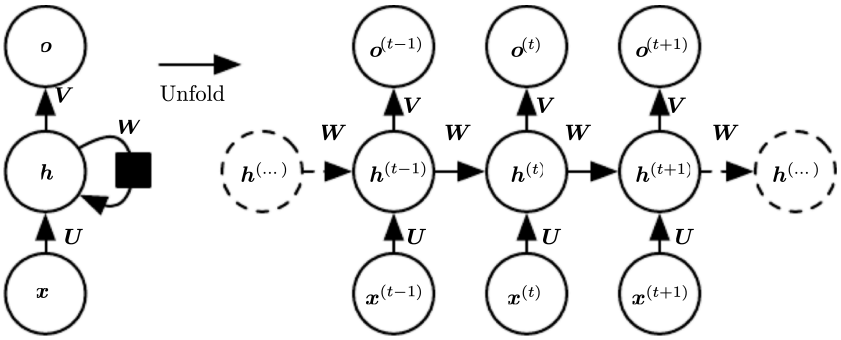


Figure 4.4: A standard graph of an RNN, and its unrolled counterpart [9]. Note: Though $\boldsymbol{x}$, $\boldsymbol{h}$, $\boldsymbol{o}$ are shown as circles, they represent *layers* rather than individual *neurons*. This breaks the typical convention, but this choice has been made for simplicity of illustration.

$$\boldsymbol{a}^{(t)} = \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{b} \tag{4.4}$$

$$\boldsymbol{h}^{(t)} = \tanh\left(\boldsymbol{a}^{(t)}\right) \tag{4.5}$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{V}\boldsymbol{h}^{(t)} + \boldsymbol{c} \tag{4.6}$$

In theory, an RNN should be able to learn any sequence. However, when learning long-term dependencies, RNNs suffer from the vanishing/exploding gradient problem. In training RNNs, when gradients are propogated through time, they are repeatedly multiplied by themselves. For gradients much smaller than one, this causes the gradients to 'vanish'. Similarly, exceptionally large gradients 'explode'. As a result, standard RNNs are unsuccessful for applications requiring a 'memory' of long-term dependencies [9].

4.2.2. LSTM

Long Short-Term Memory networks (LSTMs) are a specific type of recurrent neural network (RNN) architecture first devised by Hochreiter and Schmidhuber in 1997 [16]. LSTMs address the vanishing and exploding gradient problems encountered in traditional RNNs, allowing for long-term dependencies to be learned. While traditional RNNs consist of a chain of repeating tanh layers, LSTMs have a more sophisticated structure composed of memory cells. Each memory cell contains 4 layers which interact through a set of gates.

The structure and internal components of a memory cell within an LSTM network are illustrated in Figure 4.5. As shown, each memory cell takes the output and hidden state from the previous cell, along with an input x_t, and produces an updated hidden state and output for the next module. Each memory cell has a cell state, which uses the output values of the gates to decide which information to pass on to the next cell.

The procedure carried out by a memory cell is described by Equations 4.7 to 4.12. Each cell contains an input gate, a forget gate, and an output gate, which determine how much information from the previous cell to keep, how much to forget, and how much to pass on to the next module, respectively. Furthermore, a candidate cell state is produced, representing the potential information to be added to the cell state. With these components, the memory cell determines the updated cell state (see Equation 4.11), and finally, the updated hidden state to feed the next memory cell (see Equation 4.12) [17].

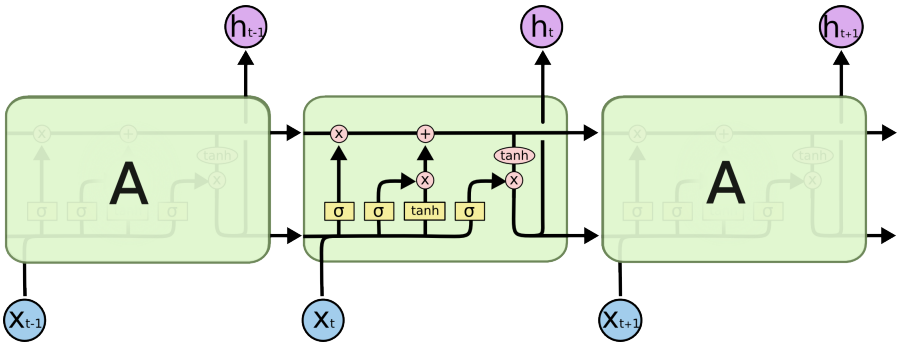


Figure 4.5: Illustration of an LSTM (source: Christopher Olah)

$$\text{Input gate: } \boldsymbol{i}^{(t)} = \sigma\left(\boldsymbol{W}_{ih}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{ix}\boldsymbol{x}^{(t)} + \boldsymbol{b}_i\right) \tag{4.7}$$

$$\text{Forget gate: } \boldsymbol{f}^{(t)} = \sigma\left(\boldsymbol{W}_{fh}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{fx}\boldsymbol{x}^{(t)} + \boldsymbol{b}_f\right) \tag{4.8}$$

$$\text{Output gate: } \boldsymbol{o}^{(t)} = \sigma\left(\boldsymbol{W}_{oh}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{ox}\boldsymbol{x}^{(t)} + \boldsymbol{b}_o\right) \tag{4.9}$$

$$\text{Candidate cell state: } \tilde{\boldsymbol{C}}^{(t)} = \tanh\left(\boldsymbol{W}_{ch}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{cx}\boldsymbol{x}^{(t)} + \boldsymbol{b}_c\right) \tag{4.10}$$

$$\text{Updated cell state: } \boldsymbol{C}^{(t)} = \boldsymbol{i}^{(t)} \odot \tilde{\boldsymbol{C}}^{(t)} + \boldsymbol{f}^{(t)} \odot \boldsymbol{C}^{(t-1)} \tag{4.11}$$

$$\text{Updated hidden state: } \boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \odot \tanh\left(\boldsymbol{C}^{(t)}\right) \tag{4.12}$$

4.3. Attention and self attention

4.3.1. Attention

Another mechanism that can improve the long-term memory of RNNs is use of an attention mechanism (often abbreviated simply as attention). Attention mechanisms in neural networks try to emulate human attention by assigning different levels of importance to various tokens in a sequence. Attention has become fundamental in many state-of-the-art models, particularly in large language models. Among the various kinds of attention mechanisms, Scaled Dot-Product Attention and Multi-Head Attention illustrated in Figure 4.6, are the most commonly used. These methods improve the model's ability to focus on relevant parts of the input, improving performance on a wide range of tasks. In mathematical

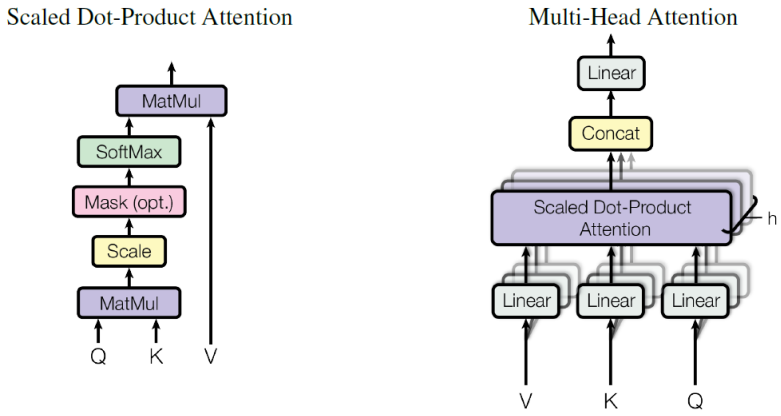


Figure 4.6: Scaled Dot-Product Attention (left) and Multi-Head attention (right) [18]

terms, attention can be expressed as the following, where d_k is the dimension:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4.13}$$

For Multi-Head attention, the expression is:

$$\begin{aligned}\text{MultiHead}(Q, K, V) &= \text{Concat}\left(\text{head}_1, \ldots, \text{head}_\text{h}\right)W^O \\ \text{where head}_\text{i} &= \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right)\end{aligned} \tag{4.14}$$

The input of attention model consists of an $N \times M$ matrix where N is the embedding dimension and M the sequence length. This matrix is then processed through three linear layers to generate the Query (Q), Key (K), and Value (V) matrices. These matrices are of size $d_k \times L$, where L is the sequence length and d_k is the dimensionality. The Key matrix describes what the input tokens represent. The Query matrix describes what the input tokens are searching for and how relevant they are to other positions. The Value matrix stores the values to be weighted at the output of the attention model. By multiplying the matrices Q and K, a weight matrix is created, reflecting the degree of attention each part of the input should receive. This weight matrix is then multiplied with the V matrix to acquire a weighted sum matrix, which now serves as an attention-based representation of the input.

Multi-Head Attention is essentially the same as Scaled Dot-Product Attention but with multiple attention layers running in parallel. These attention layers operate on lower-dimensional spaces, with each head having a dimension of d_k/h, where d_k is the dimensionality of the original Scaled Dot-Product Attention, and h is the number of heads. This lower dimensionality ensures that the computational cost remains similar to that of Scaled Dot-Product Attention (with the same overall dimensionality), while allowing the model to attend information from different representation subspaces at different positions [18].

4.4. Kolmogorov-Arnold networks

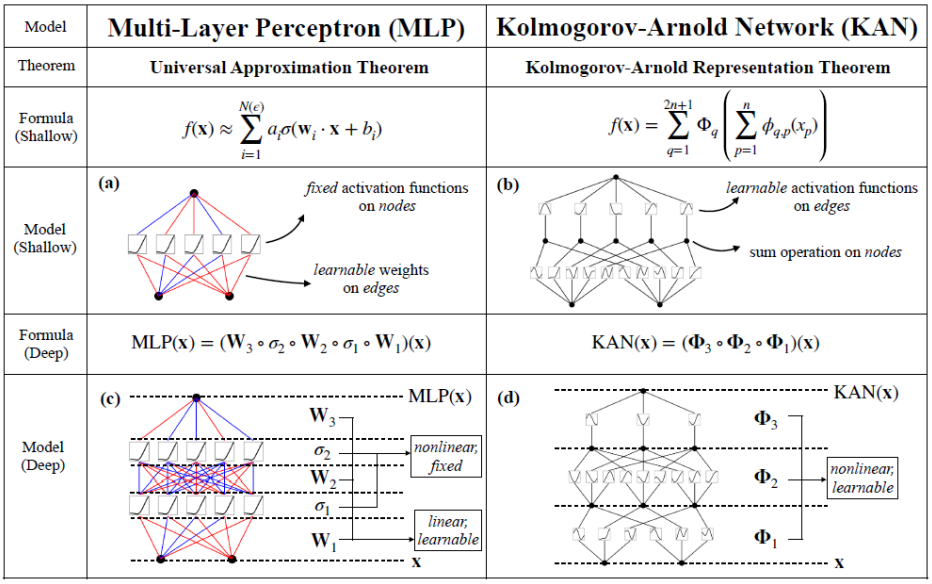


Figure 4.7: Multilayer Perceptrons in comparison with Kolmogorov Arnold Networks

Kolmogorov-Arnold Networks (KANs) represent a novel approach in the field of neural networks, with the implementation we are using only recently published [19]. The NN architectures discussed so far (CNNs and RNNs) are both examples of multi-layer perceptrons (MLPs). MLPs use weights, biases, and activation functions on neurons to approximate a formula, relying on the Universal Approximation Theorem, which can be mathematically described as $f(x) \approx \sum_{i=1}^{N(\varepsilon)} a_1 \sigma(\mathbf{W}_i \cdot \mathbf{x} + b_i))$.

However, there is another approach to approximation called the Kolmogorov-Arnold representation theorem, which states that if f is a multivariate continous function on a bounded domain, then f can be expressed as a finite composition of continuous functions of a single variable and the binary operation of addition. This can be expressed mathematically as Equation 4.15.

$$f(x) = f(x_i, ..., x_n) = \sum_{q=1}^{2n+1} \Phi_q (\sum_{p=1}^{n} \phi_{q,p}(x_p)) \tag{4.15}$$

where $\varphi_{q,p}(x_p) \rightarrow \mathbb{R}$ and $\Phi_q : \mathbb{R} \rightarrow \mathbb{R}$. A KAN is based on the Kolmogorov-Arnold theorem and does not utilize weights, biases, or activation functions. Instead, it has learnable connections between its neurons. The output of a Kolmogorov-Arnold Network can be described as:

$$f(x) = \sum_{i_L-1=1}^{N_{L-1}} \phi_{L-1,i_l,i_l-1} (\sum_{i_L-2=1}^{N_{L-2}} ... (\sum_{i_1=1}^{n1} \phi_1, i_2, i_1 (\sum_{i_0=1}^{n0} \phi_0, i_1, i_0(x_{i0})))) ...) \tag{4.16}$$

KANs can offer several improvements over MLPs: they may be more accurate with fewer parameters, more interpretable, and robust towards noisy data. However, due to their recent introduction, KANs have some downsides. They have not yet been implemented in an efficient manner, they are highly sensitive to hyperparameters changes, and they are difficult to train.

5 Model design and development

5.1. Algorithm approach

In the initial state-of-the-art analysis, it became evident that machine learning (ML) classification techniques were sufficiently accurate and reliable for deducing the intended cursor movement from MI-EEG signals. Consequently, the decision was made early on to base the classification algorithm on ML methods, and this was included as a requirement in the program of requirements. As research progressed, the advantages of deep NNs in particular became clear. Their ability to perform representation learning is beneficial for analysis of MI-EEG signals, which are high in both dimensionality and variability. For such signals, manual extraction would be limited, as it would confine the analysis to a predefined set of features, hindering the network's ability to fully understand the data.

Representation learning, as performed by deep NNs, uniquely caters to high dimensional data for two reasons. Firstly, the representation learning performed by deep NNs is hierarchical (as a reminder, this means that features are found at increasing levels of abstraction as the data passes through the layers of the network). Secondly, representation learning allows for the most discriminative aspects of the dataset to be acquired automatically, which effectively translates to dimensionality reduction. These two advantages could not be achieved through manual extraction.

Several deep neural network architectures were considered for the algorithm, guided by insights from existing literature. These architectures ranged from simplistic feed-forward neural networks to cutting-edge approaches such as transformers [20]. It was ultimately observed that the highest accuracy values were often achieved by combining multiple neural network architectures into a single model [4]. Notably, models that combined CNNs with LSTMs peformed exceptionally well [21] [22] [23]. The success of this combination can be well understood using the theory provided in the previous chapter.

As established, CNNs excel at learning spatial features and dependencies from raw data. Specifically, for MI-EEG signals, this means CNNs are well-equipped to identify the spatial variations of the signals across multiple channels. Meanwhile, LSTMs specialize in processing, and keeping memory of, sequential data. This is advantageous for MI-EEG signals where temporal patterns are crucial for understanding the data. Essentially, this hybrid approach effectively captures both the spatial and temporal characteristics of the data. Consequently, such an architecture can more easily form a comprehensive model of MI-EEG signals, leading to improved performance.

There are numerous papers which detail the development of a combined CNN-LSTM model for MI-EEG classification. However, the exact details of the architecture varies considerably between cases. Some approaches combine the CNN-LSTM in series, while others combine them in parallel. Furthermore, there are also significant changes in the exact layout and form of the network; the number and ordering of layers, the chosen values of hyperparameters, etc. Accordingly, there are also differences between the accuracy that these models produce. Therefore, even after honing our search to CNN-LSTM models, closer investigation was needed to determine which precise architecture to implement.

Several CNN-LSTM models were considered. The models, as well as the accuracy they obtained on the BCI IV-2a dataset, are listed in Table 5.1. Eventually, it was chosen to base the classification algorithm on the the model devised by Cheng and Hao in [22]. This choice was made in light of the exceptionally high accuracy obtained by the model on the BCI IV-2a dataset, as well as the high degree of transparency in the paper, making the model relatively easy to implement[1].

[1] It should be mentioned that this paper was found on arXiv, and it is thus not peer reviewed. However, it was chosen nonetheless in light of its high accuracy and reproducibility. The authors of this paper are affiliated with well-known institutions, and have several published papers on the topic of EEG signal classification. Hence, we are confident that this paper is still a reliable source.

Paper (first author)	Accuracy (%)
Wang [24]	75.4
Zhang [25]	81.0
Li [26]	87.7
Yang [23]	89.3
Khademi [21]	92.0
Cheng [22]	92.7

Table 5.1: A comparison of CNN-LSTM models for BCI IV-2a MI-EEG classification (sorted from lowest to highest accuracy)

5.2. Algorithm design

5.2.1. Overview

The final model consists of a 3D CNN in parallel with an LSTM network which is concatenated together into a vector and fed into a set of fully connected layer, as shown in Figure 5.1. The output of the model can either be a standard 1×4 vector. or a $1 \times n$ vector if the choice for embedding vectors is made. Its input is a matrix of size $N \times 1 \times L \times C$. Where N is batch size, L is sequence length, and C is the channel dimension. So a sample of length 529 and with 16 channels would look like the following matrix $1 \times 1 \times 529 \times 16$.

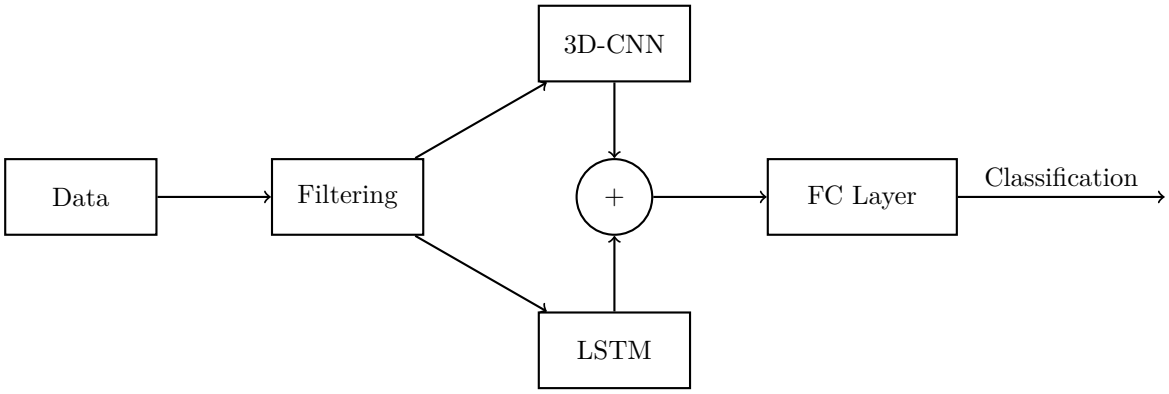


Figure 5.1: Diagram illustrating the data processing pipeline.

5.2.2. Dataset

Dataset Description The validation of our model was conducted on the BCI Competition IV 2a Dataset [6]. The Dataset consists of the EEG measurements of nine participant for four motor imagery classes. The classes involved motor imagery for the left-hand, right-hand, feet, and tongue. Each participant took part in two sessions on two different dates. These sessions were comprised of six sets with each set containing a run of 6 minutes with 48 trials. The final size is therefore 288 samples per participant and 2600 samples overall. The participants were seated in a comfortable chair in front of a task screen. The acquisition paradigm for a trial has the following form. First an auditory cue was heard together with a fixation cross appearing on a black screen, two seconds later a cue with the form of an arrow pointing upwards, downwards, left or right appeared and stayed on the screen for 1.25s. This instructed the participant to perform a specific motor imagery task. this action was carried out until the crosshair disappeared from screen at t = 6s, a short break then follows until the next trial starts.

Why was it chosen The Dataset was chosen based on our Program of requirements which specifies that we have to have an output for four different actions. It was also chosen based on the fact that it is a relatively popular dataset. Therefore it is easy to find and compare our network to other EEG classification networks and learn how our model can be improved and to get a different outlook on the same problem.

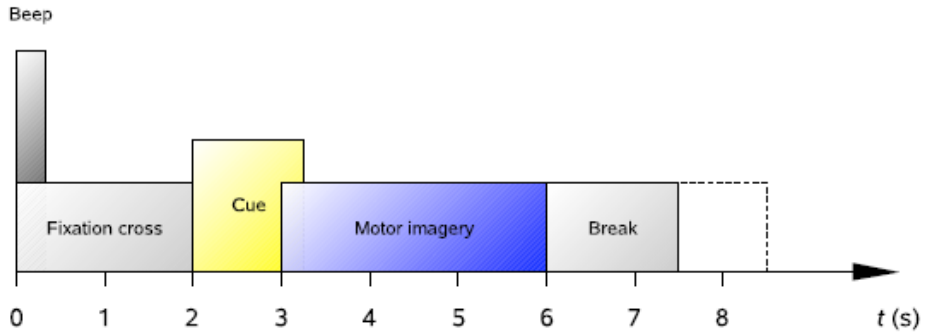


Figure 5.2: Data acquisition paradigm

5.2.3. Filtering

In EEG signal measurement, multiple electrodes are placed on the user's scalp, and their potential differences are recorded relative to a reference electrode positioned on the back of the neck. Each electrode corresponds to its own channel in the EEG reading. However, this setup means that each recording electrode is affected by the same noise present in the reference. Thus, to improve the SNR ratio of the recordings, it was decided to utilise a common average reference (CAR) filter. A CAR filter computes the mean signal value across all channels, and subtracts this mean from each channel. This process essentially isolates the exclusive behaviour on each channel, reducing the 'common noise' from all electrodes, resulting in a higher SNR. In mathematical terms, the function of a CAR filter is described by Equation 5.1. First, the mean signal across all C channels is computed. Then, for each channel i, the mean value is subtracted.

$$x_i^{\mathrm{CAR}}(t) = x_i(t) - \frac{1}{C}\sum_{j=1}^{C} x_j(t) \tag{5.1}$$

In addition to the CAR filter, a frequency filter was implemented, passing through only those frequencies between 0.5 and 38 Hz. This was because, according to existing literature this range of EEG signals produces the highest classification accuracy [27].

5.2.4. CNN module

The convolutional network module as can be seen in figure 5.3 of the model consist of a residual 2 block deep 3D convolutional network, each block employs two of the same convolutional layers in series one with a 3×3 kernel and one with 5×5 kernel and one 7×7 kernel on its own. these convolutional layers with different kernel sizes are placed in parallel with each-other per block. the convolutional layers have a filter size of $5 * 3^n$ where n is the depth of the layer. The filter amount has been chosen based on model testing for various filter amounts. The model differs from the Cheng paper[22]. It's depth and filter amount have been greatly reduced this was done based on the better model performance, to prevent overfitting and to the PoR the max size of the model training should be below 4 GB. Releasing these last 2 layers frees up a large amount of space. After each convolutional block the data is reduced by the means of an average pool block, this was chosen over maxpool because of performance in testing. Lastly the output of a block was normalized by a batchnorm layer.

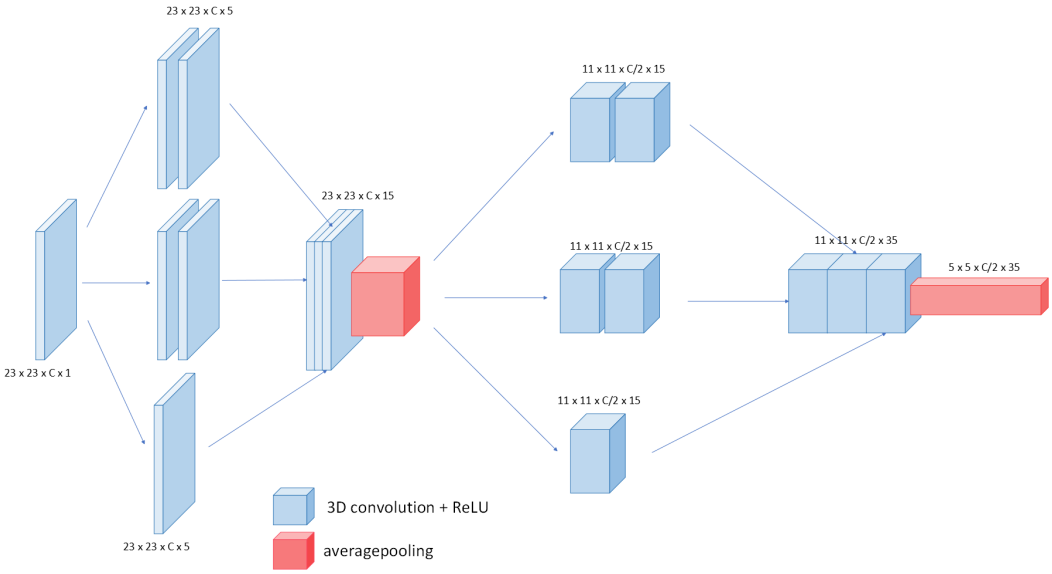


Figure 5.3: 3D convolutional module

The input of the convolutional network is a matrix in the form $(M \times M \times C)$ where C refers to the number of EEG channels. The dimension M is gotten from taking an EEG sequence with length L, splitting this sequence into M parts and stacking these on top of each other to create a matrix of size $(M \times M)$ where $M * M = L$. By structuring an input matrix of this size the network can detect both short time relations, medium time relations and long time relations and spatial relations because of the convolution in 3 dimensions and its different kernel sizes. Another possibility is instead of feeding the 3D convolutional network time series EEG data it is also possible to feed it preprocessed wavelet transformed data. With this approach a shallower less complex net can be used which can improve VRAM usage and latency.

5.2.5. LSTM module

The LSTM module implemented is a standard LSTM consisting of 265 units. It takes in the a matrix of $L \times C$ where L is the length time of the EEG sample and C the channel component. It outputs a vector of size 1×256 relating to the 256 unit input. Afterwards this vector is fed trough a Multi-Head Attention module to denote the most important parts for the feedforward network.

5.2.6. Fully connected layers

After the outputs of the output of the 3D-CNN and LSTM have been flattened and concatenated they are fed into a set of fully connected layers. The first layer Consists of 50 neurons which is then fed trough to a b-spline activation functions(degree = 3, grid = 5, the used implementation is in appendix A.2). This activation function was chosen over the ReLu activation based on its performance in testing and the research done in [28]. It is then fed trough to a 50 by 4 fully connected layer in which the 4 layer is terminated into a softmax activation function for classification.

5.3. Training

This section of the report gives an overview onto how the model was trained and what implementations were used to make the model more accurate, prevent overfitting and speed up training. It also describes the training loop used.

5.3.1. Data augmentation

The first step taken was data augmentation, because of the generally large CNN used it needs to have to have a lot of samples to be able to train correctly. For this a cropping style data augmentation method was used [29] because of its effectiveness at increasing accuracy. Cropping originally comes from computer vision training where the training size of a dataset was increased by cropping out parts

of photos and using theme as a separate training samples. For EEG signals this is implemented by having a window slide over the data with a step size that can be chosen based on how much you want to expand the data. For our data augmentation cropping was done with a step size of 25 allowing us to get 2016 samples out of a dataset of 288 samples.

5.3.2. DROPOUT

Dropout layers were added to the model to prevent overfitting[30]. A dropout layer randomly turns of a certain percentage of neurons in a layer, in this way it prevents networks from overfitting by preventing co-adaptation by neurons. A dropout of 10% was used in the convolutional layers while a dropout of 50% was used for the neurons in the fully connected layer.

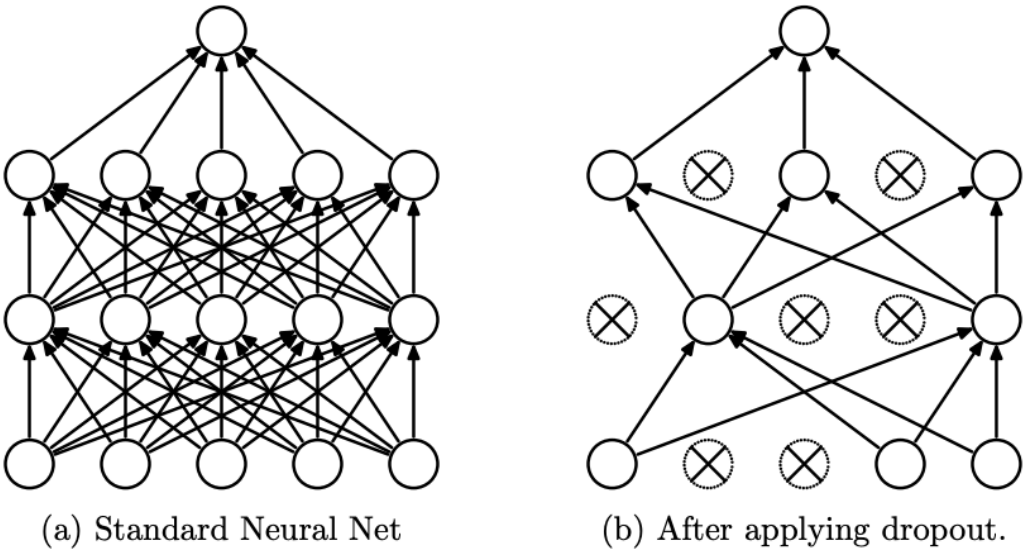


Figure 5.4: Dropout applied to a neural network

5.3.3. BATCH NORMALIZATION

Batch normalization was employed after each layer and before each activation function instead of the order suggested in the original paper, this helped improve our convergence a bit faster but this should not matter that much as said in [31]. Batch normalization helps speed up and stabilize the training process by normalizing the inputs of each layer. It is defined as

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta \tag{5.2}$$

where γ and β are learnable parameters. It reduces covariate shift by using this formula[32] which leads to the benefits of faster training, the reduction of overfitting and regularization.

5.3.4. TRAINING LOOP

Training of the network was done using AdamW as optimizer based on the advantages of this optimizer as mentioned in 3.7.2. It was trained with a learning rate of 0.1 that gets multiplied by 0.7 every 30 iterations and a weight decay of 0.001. Each model for each specific subject in the BCI IV 2a dataset was trained on with the exact same settings and was trained on for 1000 iterations with a batch size of 64. Training was done with a set seed to ensure reproducibility.

5.4. SYSTEM INTEGRATION

At the moment of writing this thesis, the classification algorithm has not yet been integrated with the remaining subgroups in the project; the preprocessing and GUI teams. However, the arrangements required for this integration have already been discussed, and are described below in this section.

5.4.1. Preprocessing subgroup

During the training procedure, live data will be measured by the preprocessing subgroup. This measured data will then be cleaned and provided to the decoding subgroup as a CSV file. During usage (i.e. 'testing), for every time sample, we will receive an array with a dimension suitable for our classification algorithm (this dimension was explained in 5.2.1). The classification algorithm will then produce a classification vector for this sample, and output this to the GUI subgroup.

5.4.2. GUI subgroup

The integration arrangements with the GUI subgroup include several key elements. Select training parameters of our model will be adjustable biew the GUI, these being the learning rate, batch size, max iteration, and margin. The classification algorithm will be classifies amongst 4 classes: left hand, right hand, feet and tongue. These body parts correspond to the directions left, right, down, and up, respectively. For each MI-EEG sample received by the classification algorithm, a 4-element vector will be outputted to the GUI subgroup, with the elements representing the probability of the signal belonging to each of the four subgroups. Furthermore, the GUI must provide the accuracy scores and loss plots to the user. These will be supplied by the classification algorithm in the form of numpy arrays.

6 Results

In this section the evaluation of our network will be provided. The evaluation of the network was done using classification accuracy and kappa score for a 2 class model, a 4 class model and on our own dataset and the BCI IV-2a dataset. The results will be used to analyze the shortcoming of the model and propose future improvements.

6.1. Testing definitions

For the testing, 2 scores will be used accuracy and kappa value. Accuracy is defined as the number of correct predictions divided by the number of total predictions.

$$accuracy = \frac{Correct\ \ predictions}{Total\ \ predictions} \tag{6.1}$$

The kappa value is a statistic that is used to measure the inter-rater reliability for qualitative items. It is defined as

$$\kappa = \frac{p_o - p_e}{1 - p_e} \tag{6.2}$$

p_o denotes the relative observed agreement between a pair of variables, for our model the accuracy, while p_e is the expect agreement in case our variables are assigned labels randomly, in this case because of our 2 classifications $p_e = \frac{1}{2}$.

6.2. BCI IV-2a

To test the model and evaluate its accuracy and kappa value 10-fold cross validation. This was then repeated for a number of different seeds to also take in the account the effect of random weight initialization, the data was also carefully split as to ensure a balanced training and testing set. The training loop for the 10-fold validation stayed exactly the same as was described in the training loop section(5.3.4).

6.2.1. 4 Classes

Table 6.1: Classification accuracy's for four classes

Subject	accuracy	kappa
SO1	0.67	0.56
SO2	0.38	0.17
SO3	0.71	0.61
SO4	0.42	0.22
SO5	0.42	0.22
SO6	0.43	0.24
SO7	0.52	0.36
SO8	0.72	0.62
SO9	0.72	0.62
Mean	0.55	0.40

The model has an average score of 55% and a kappa value of 0.40. During training the latency and VRAM usage of the model were also measured to ensure compliance with the Program of requirements. The final VRAM usage came out to at 2.6GB while sample propagation time(time from input to output of the model) reached a max of 21ms. This means that training and inference satisfy the requirments stated in the PoR.

6.2.2. 2 classes

The two classes used are the right hand class and the tongue class. These where chosen based on the measurements and observations done by the measurements subgroup and based on training with triplet loss. Because of the embedding property from training with triplet loss You can look at the distances between the embedding vectors for the classes to get an accurate outlook at the distinguish-ability of each class. The training loop of the 2 class model was slightly modified. Instead of 100 iterations it now only has 250 iterations because of its faster convergence and the learning rate scheduler has been set to a more aggressive mode with a multiplication every 10 iterations instead of 30.

Subject	accuracy	kappa
SO1	0.93	0.86
SO2	0.58	0.16
SO3	0.87	0.74
SO4	0.70	0.40
SO5	0.64	0.28
SO6	0.70	0.40
SO7	0.76	0.52
SO8	0.87	0.74
SO9	0.96	0.92
Mean	0.78	0.54

Table 6.2: Classification accuracy's for two classes

6.3. Our dataset (static)

The accuracy on our own dataset is not yet included because of a late issue found in the project cycle we ran into time constraint. The dataloader for the csv file put out by the measurement subgroup has been completed and initial testing is underway, Up until now the results have been confusing the models does not seem to learn from it. An investigation is being conducted and with the hopes of presenting the full results during the presentation and demonstration.

6.4. Complications

Sadly a misstep was taken during our initial training and testing of the model by having a fault in our train/test split. We discovered this to late and our accuracy on 4 classes suffered a lot from this. Therefore it was chosen to scale down the model to 2 classes and analyze the shortcomings of the model. From analysis done via both the use of embedding functions and triplet loss training combined with input from the measurement subgroup and research it was found out that our model had a lot of complications with distinguishing left and right hand signals. The section below will give alternative ways that the shortcoming of our model can be mitigated.

6.4.1. wavelets

Because of the high non stationary and frequencies spectrum's in EEG signals another approach of extracting information is via a continuous wavelet transform. By taking a mother wavelet(a function generated from the wavelet function with variable windows of variable widths) you can take advantage of the fact that that low frequencies are propagated over time while high frequencies only appear in short bursts. to get a continuous wavelet transform of the EEG data a window is slid alongside the EEG time data. This window performs a convolutional operation.

$$W(a,b) = \frac{1}{\sqrt{a}} \int_{-\infty}^{+\infty} s(t)\psi^*(\frac{t-b}{a})dt \tag{6.3}$$

ψ is the mother wavelet and is defined as

$$\psi(t) = \frac{1}{\sqrt{\pi f}} e^{j2\pi f t} e^{\frac{t^2}{f}} \tag{6.4}$$

for a Complex Morlet wavelet. By taking the absolute of this continuous wavelet transform the classification can be transformed into an image classification problem like done in this paper[21]. In Figure 6.1 the differences in images between left and right hand, feet and tongue can be seen.

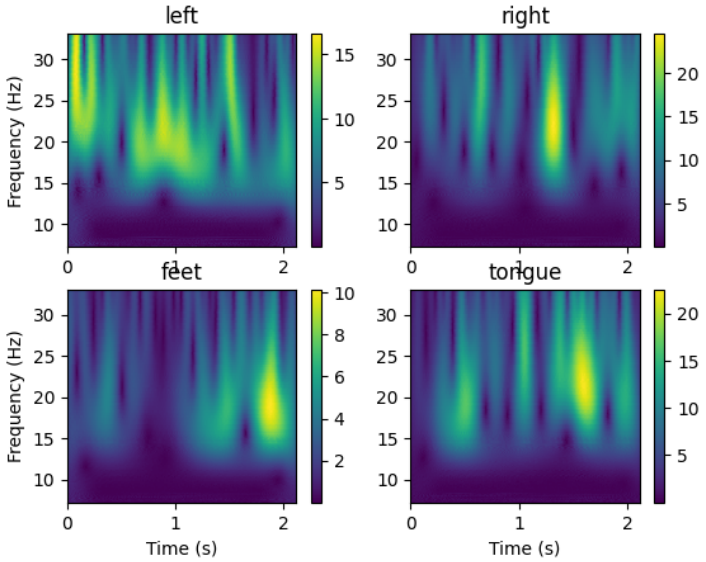


Figure 6.1: Complex vawelet transform of the 4 classes

6.4.2. Filter Bank common spatial pattern

Another approach that could be taken is Filter bank common spatial pattern or FBCSP this was used in a paper by Zhang[25] Together with a convolutional network/lstm structure by taking multiple shorter samples in the same time interval and combining them together two form a 2D matrix. It works as follows you filter your signal for several frequency bands, you then apply spatial filtering and select your features. With these features a vector is created to use as input in either a machine learning algorithm(K-nn,SVM,RNFS) or combined together with its neighbouring time samples into a 2D matrix and fed into a deep learning algorithm.

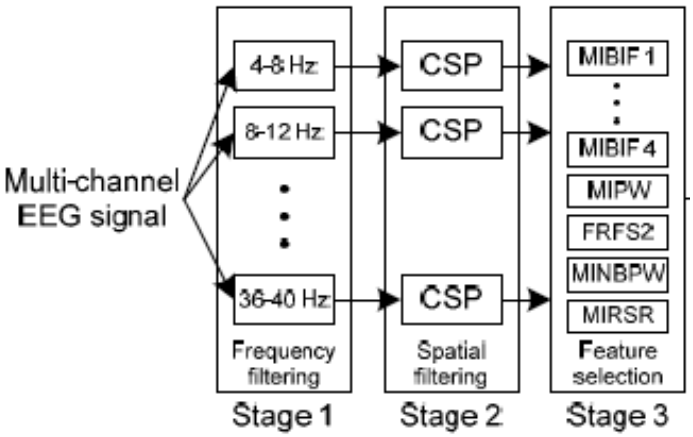


Figure 6.2: Filter Bank Common Spatial Filtering

7 Conclusion

7.1. Conclusion

The primary goal of the decoding subgroup was to develop an algorithm capable of decoding EEG data streamed from the cap to a PC, and make classifications based on this data to send to the interface subgroup.

In conclusion, this thesis presented a CNN-LSTM model to decode MI-EEG signals. Unfortunately, the model performed worse than expected. While a reasonable mean accuracy was achieved of 2-class classifications (78%), the model struggled with the 4-way classification (55%). The issues of the system have been identified and the possible solutions have been determined, but due to time constraints and a late discovery of this problem, we have not been able to implement these yet. As a result of this, some of the decoding requirements (specifically, A1, B2, and B3) could not be reached. On the bright side, the other requirements have largely been met.

7.2. Discussion

Overall, the initial steps of this project went rather well. A comprehensive plan for the project, as well as the necessary research, was formed quickly, and with high-quality outcomes. However, due to complications we experienced in the training and test code files (see section 6.4: Complications), it was discovered that the accuracy produced by the model was lower than what we initially believed. This issue was discovered late into the project cycle, and the necessary changes to adjust this could not be implemented in time for the thesis submission. We hope to able to make the necessary refinements to the model in the upcoming weeks and have a better working version for the presentation and demonstration. Furthermore, due to time constraints, we were unable to fully integrate with the remaining subgroups. Perhaps with a closer attention to detail, these limitations could have been avoided.

In terms of the technical outcomes of this project, we are somewhat satisfied with what was achieved. Combining several advanced topics in the field of deep learning into a single comprehensive model for MI-EEG classification was an insightful and informative experience. Moreover, since we have identified the current technical limitations of our system, we are confident that we will be able to improve upon these areas.

Concerning the *general* requirement, it is difficult to comment on the achievement of these, as the integration process has not yet been completed. We hope to provide an update of this during our thesis defence. Moving on to the *decoding* requirements, some, but not all, are satisfied. Functional requirements A2 through A4 our satisfied: the classification algorithm is able to work locally on a Windows PC, produces a 1×4 probability classification vector as output, and satisfies the VRAM usage requirement. Furthermore, both implementation requirements, C1 and C2, have been satisfied: our model is based on machine learning models, and has been implemented in Python (with PyTorch and other associated libraries). Additionally, requirement B1 was been satisfied: the computational latency of our model is far less than 2.5 seconds. However, as the integration process has not yet taken place, and our accuracy is far less than we expected, requirements A1, B2, and B3 have not been satisfied. Since three of the decoding requirements have not been met, and the general requirements can not yet be tested for at all, the classification algorithm is unsuccessful for the time being. We will *try* to have a fully working system that satisfies all requirements in times for the thesis defence, but there is no guarantee of this.

In summary, our project has been successful in its planning and research. Moreover, in terms of technical outcomes, we are somewhat satisfied with what we have achieved. However, as several product requirements have not been met, our product, the classifical algorithm, can not be considered successful.

7.3. Future work

- Experimenting with other classification systems: Based on the research conducted, a CNN-LSTM model was chosen, but there are many other possible options. For example, the wavelet approach , a combined wavelet time series apporach or an encoder approach like done in [20]. Another possible future is in either transfer learning to improve generalizability or in generative AI and encoder decoder architecture to be able to accurately expand datasets.
- Integrating the system: As of the submission of this thesis, our decoding algorithm has not been fully integrated with the remaining subgroup in our project. We have yet to test on continuous incoming data, and as this is a fundamental feature of our envisioned system, this is a crucial area of further development. Similarly, we have yet to integrate with the interface group, and as such are not able to produce any meaningful visualization to the user. Due to the lack of these features, most general requirements are currently unsatisfied.
- Subject independent classification: Our MI-EEG classification algorithm is subject-specific – it must be trained for the individual user in alignment with general requirement B4. However, several papers have attempted to develop subject-agnostic classifiers which must only be trained once, and can subsequently be attempted by any user. In general, such methods result in reduced accuracy. Regardless, this is an interesting direction to investigate, if only for educational purposes. This would not necessarily be difficult. Our existing system could be slightly modified to experiment with such an approach.
- Distinguishing individual fingers: Conventional MI-EEG classifiers, including our own, typically include the entire hand as one signal, i.e. left hand MI to go left and right hand MI to go right. However, a recent paper has investigated the possibility to distinguish MI signals corresponding to individual fingers. The model proposed by this paper achieved astoundingly poor results, at only 30% accuracy [33]. Though this figure is far promising, it demonstrates that there is much room for research in this area. Outperforming such cutting-edge research is likely out of our capabilities for the time being, but could nonetheless an interesting concept to investigate.

References

[1] J. J. Shih, D. J. Krusienski, and J. R. Wolpaw, "Brain-computer interfaces in medicine," *Mayo Clinic Proceedings*, vol. 87, no. 3, pp. 268–279, 2012. DOI: 10.1016/j.mayocp.2011.12.008.

[2] g.tec. "G.tec." (2024), [Online]. Available: https://www.gtec.at/product/unicorn-hybrid-black/.

[3] F. Lotte, L. Bougrain, A. Cichocki, *et al.*, "A review of classification algorithms for eeg-based brain–computer interfaces: A 10 year update," *Journal of Neural Engineering*, vol. 15, no. 3, p. 031 005, Apr. 2018. DOI: 10.1088/1741-2552/aab2f2. [Online]. Available: https://dx.doi.org/10.1088/1741-2552/aab2f2.

[4] H. Altaheri, G. Muhammad, M. Alsulaiman, and et al., "Deep learning techniques for classification of electroencephalogram (eeg) motor imagery (mi) signals: A review," *Neural Computing and Applications*, vol. 35, pp. 14 681–14 722, 2023. DOI: 10.1007/s00521-021-06352-5.

[5] V. J. Lawhern, A. J. Solon, N. R. Waytowich, S. M. Gordon, C. P. Hung, and B. J. Lance, "Eegnet: A compact convolutional neural network for eeg-based brain–computer interfaces," *Journal of Neural Engineering*, vol. 15, no. 5, p. 056 013, Jul. 2018, ISSN: 1741-2552. DOI: 10.1088/1741-2552/aace8c. [Online]. Available: http://dx.doi.org/10.1088/1741-2552/aace8c.

[6] B. Blankertz, C. Vidaurre, M. Tangermann, K.-R. Müller, *et al.* "Bci competition iv." (2008), [Online]. Available: https://www.bbci.de/competition/iv/#news.

[7] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/.

[8] S. J. Prince, *Understanding Deep Learning*. The MIT Press, 2023. [Online]. Available: http://udlbook.com.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org.

[10] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015. DOI: 10.1038/nature14539.

[11] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, "Activation functions in deep learning: A comprehensive survey and benchmark," *Neurocomputing*, vol. 503, pp. 92–108, 2022, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2022.06.111. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231222008426.

[12] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," 2017. arXiv: 1710.05941.

[13] PyTorch, *Torch.nn.softmax*, 2024. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html.

[14] I. Loshchilov and F. Hutter, "Fixing weight decay regularization in adam," *CoRR*, vol. abs/1711.05101, 2017. arXiv: 1711.05101. [Online]. Available: http://arxiv.org/abs/1711.05101.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. arXiv: 1512.03385. [Online]. Available: http://arxiv.org/abs/1512.03385.

[16] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[17] A. Graves, A.-r. Mohamed, and G. Hinton, *Speech recognition with deep recurrent neural networks*, 2013. arXiv: 1303.5778.

[18] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, *Attention is all you need*, 2023. arXiv: 1706.03762.

[19] Z. Liu, Y. Wang, S. Vaidya, *et al.*, *Kan: Kolmogorov-arnold networks*, 2024. arXiv: 2404.19756 [cs.LG].

[20] W. Liao, *Eegencoder: Advancing bci with transformer-based motor imagery classification*, 2024. arXiv: 2404.14869.

[21] Z. Khademi, F. Ebrahimi, and H. M. Kordy, "A transfer learning-based cnn and lstm hybrid deep learning model to classify motor imagery eeg signals," *Computers in Biology and Medicine*, vol. 143, p. 105 288, 2022, ISSN: 0010-4825. DOI: https://doi.org/10.1016/j.compbiomed.2022.105288. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010482522000804.

[22] S. Cheng and Y. Hao, *3d-clmi: A motor imagery eeg classification model via fusion of 3d-cnn and lstm with attention*, 2023. arXiv: 2312.12744 [cs.HC].

[23] J. Yang, S. Yao, and J. Wang, "Deep fusion feature learning network for mi-eeg classification," *IEEE Access*, vol. 6, pp. 79 050–79 059, 2018. DOI: 10.1109/ACCESS.2018.2877452.

[24] J. Wang, S. Cheng, J. Tian, and Y. Gao, "A 2d cnn-lstm hybrid algorithm using time series segments of eeg data for motor imagery classification," *Biomedical Signal Processing and Control*, vol. 83, p. 104 627, 2023, ISSN: 1746-8094. DOI: https://doi.org/10.1016/j.bspc.2023.104627. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1746809423000605.

[25] R. Zhang, Q. Zong, L. Dou, X. Zhao, Y. Tang, and Z. Li, "Hybrid deep neural network using transfer learning for eeg motor imagery decoding," *Biomedical Signal Processing and Control*, vol. 63, p. 102 144, 2021, ISSN: 1746-8094. DOI: https://doi.org/10.1016/j.bspc.2020.102144. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1746809420302901.

[26] H. Li, M. Ding, R. Zhang, and C. Xiu, "Motor imagery eeg classification algorithm based on cnn-lstm feature fusion network," *Biomedical Signal Processing and Control*, vol. 72, p. 103 342, 2022, ISSN: 1746-8094. DOI: https://doi.org/10.1016/j.bspc.2021.103342. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1746809421009393.

[27] X. Zhao, H. Zhang, G. Zhu, F. You, S. Kuang, and L. Sun, "A multi-branch 3d convolutional neural network for eeg-based motor imagery classification," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 27, no. 10, pp. 2164–2177, 2019. DOI: 10.1109/TNSRE.2019.2938295.

[28] P. Bohra, J. Campos, H. Gupta, S. Aziznejad, and M. Unser, "Learning activation functions in deep (spline) neural networks," *IEEE Open Journal of Signal Processing*, vol. 1, pp. 295–309, 2020. DOI: 10.1109/OJSP.2020.3039379.

[29] R. T. Schirrmeister, J. T. Springenberg, L. D. J. Fiederer, *et al.*, "Deep learning with convolutional neural networks for brain mapping and decoding of movement-related information from the human EEG," *CoRR*, vol. abs/1703.05051, 2017. arXiv: 1703.05051. [Online]. Available: http://arxiv.org/abs/1703.05051.

[30] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html.

[31] M. Hasani and H. Khotanlou, "An empirical study on position of the batch normalization layer in convolutional neural networks," *CoRR*, vol. abs/1912.04259, 2019. arXiv: 1912.04259. [Online]. Available: http://arxiv.org/abs/1912.04259.

[32] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. arXiv: 1502.03167. [Online]. Available: http://arxiv.org/abs/1502.03167.

[33] Y.-M. Go, S.-H. Yu, H.-Y. Park, M. Lee, and J.-H. Jeong, *Fingernet: Eeg decoding of a fine motor imagery with finger-tapping task based on a deep neural network*, 2024. arXiv: 2403.03526.

A | Code

Adding source code to your report/thesis is supported with the package `listings`. *An example can be found below. Files can be added using* `\lstinputlisting[language=<language>]{<filename>}`.

A.1. Model

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchinfo import summary
from attentionmod import blockblock,multihead
from bspline import spline_activation
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#device = "cpu"
from kan import KAN

#convnet decleration/architecture

filer = 32
arch_2 = [
    (5,1),
    (5*3,1)
]

arch_wavelet = [
    (3,32,2,2),
    (3,64,2,2),
]
# freeze escargot 2 architecture#

#rewrite
class L2NormalizationLayer(nn.Module):
    def __init__(self, dim=1, eps=1e-12):
        super(L2NormalizationLayer, self).__init__()
        self.dim = dim
        self.eps = eps
    def forward(self, x):
        return F.normalize(x, p=2, dim=self.dim, eps=self.eps)

#convblock class
class convblock(nn.Module):
    def __init__(self,in_channels,filter_number):
        super(convblock,self).__init__()
        self.conv3 = nn.Conv3d(in_channels = in_channels, out_channels = filter_number,
            kernel_size=3, stride = 1, padding = 1, bias = False)
        self.conv5 = nn.Conv3d(in_channels = in_channels, out_channels = filter_number,
            kernel_size=5, stride = 1, padding = 2, bias = False)
        self.conv7 = nn.Conv3d(in_channels = in_channels, out_channels = filter_number,
            kernel_size=7, stride = 1, padding = 3, bias = False)
        self.conv33 = nn.Conv3d(in_channels = filter_number, out_channels = filter_number,
            kernel_size=3, stride = 1, padding = 1, bias = False)
        self.conv55 = nn.Conv3d(in_channels = filter_number, out_channels = filter_number,
            kernel_size=5, stride = 1, padding = 2, bias = False)
        #self.conv77 = nn.Conv3d(in_channels = filter_number, out_channels = filter_number,
            kernel_size=7, stride = 1, padding = 3, bias = False)
        self.batchnorm = nn.BatchNorm3d(filter_number*3)
        self.max = nn.MaxPool3d(kernel_size=(2,2,2))
        self.avg = nn.AvgPool3d(kernel_size=(2,2,2))
        self.act = nn.LeakyReLU()
        self.dropout3d = nn.Dropout3d(p=0.1)
    def forward(self,x):
        d,i,ei,e2,e3 = x.shape
```

```
        x3 = self.conv3(x)
        x33 = self.conv33(x3)
        x5 = self.conv5(x)
        x55 = self.conv55(x5)
        x7 = self.conv7(x)
        xall = torch.cat((x33,x55,x7),dim = 1)
        x = torch.cat((x,x,x),dim=1)
        if i == 1:
            return self.dropout3d(self.avg(self.act(self.batchnorm(xall))))
        else:
            return self.dropout3d(self.avg(self.act(self.batchnorm(xall+x))))

class CNNBlock(nn.Module):
    def __init__(self,in_channels, out_channels, **kwargs):
        super(CNNBlock,self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, bias=False, **kwargs)
        self.batchnorm = nn.BatchNorm2d(out_channels)
        self.leakyrelu = nn.LeakyReLU()
    def forward(self,x):
        d,i,e2,e3 = x.shape
        if i == 16:
            return self.leakyrelu(self.batchnorm(self.conv(x)))
        else:
            return self.leakyrelu(self.batchnorm(self.conv(x)))

class attention(nn.Module):
    def __init__(self,n_channels,n_head):
        super().__init__()
        head_size = n_channels/n_head
        self.multihead = multihead(n_channels,n_head,head_size)
    def forward(self,x):
        inter = self.multihead(x)
        return inter

class lstmmodule1(nn.Module):
    def __init__(self, input_len = 22, hidden_size=256, num_layers=1):
        super(lstmmodule1, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.attention = attention(256,1)
        self.lstm = nn.LSTM(input_len,hidden_size,num_layers,batch_first=True)
    def forward(self,x):
        hidden_states = torch.zeros(self.num_layers, x.size(0),self.hidden_size).to(device)
        cell_states = torch.zeros(self.num_layers, x.size(0),self.hidden_size).to(device)
        out, (hn,cn) = self.lstm(x,(hidden_states.detach(),cell_states.detach()))
        return self.attention(hn)

class kan(nn.Module):
    def __init__(self):
        super(kan, self).__init__()
        self.kan = KAN([529,40,20])
        self.flatten = nn.Flatten()
    def forward(self,x ):
        N,C,H,W = x.shape
        xa = torch.squeeze(x)
        list = []
        xa = xa.permute((2,0,1))
        for _ in range(W):
            int2 = self.kan(xa[_])
            list.append(int2)
        output = torch.stack(list)
        output = output.permute((1,2,0))
        output = self.flatten(output)
        return output

#model
class escargot(nn.Module):
    def __init__(self,in_channels = 1,in_channels2=16):
        super(escargot,self).__init__()
        self.arch3d = arch_2
        self.arch2d = arch_wavelet
```

```
        self.in_channels = in_channels
        self.in_channels2 = in_channels2
        self.Cnn = self.Create_conv_layers3d(self.arch3d)
        self.nn = self.create_nn()
        self.flatten = nn.Flatten()
        self.lstm = lstmmodule1()
        self.norm = L2NormalizationLayer()
       #self.transoferm = blockblock(1,1,4,529,22).to(device)
        #self.kann = kan()
    def forward(self,x):
        N,C,H,W = x.shape # get shape of input
        #x = torch.permute(x,(0,1,3,2))
        xr = torch.reshape(x,(N,1,23,23,22)) # reshape input for convolutional neural network
        #3d convolutional part time series
        intermediate = self.Cnn(xr)
        intermediate = self.flatten(intermediate)
        #concatenating in one output layer
        xlstmi = torch.squeeze(x)
        xlstm = self.lstm(xlstmi)
        xlstm = torch.squeeze(xlstm)
        intermediate = torch.cat((intermediate,xlstm),dim=1)
        return self.nn(intermediate)

    def Create_conv_layers3d(self,arch): #creates the 3d convolution layers used before the
        lstm
        in_channels = self.in_channels
        list = []
        for x in arch:
            if type(x) == tuple:
                list += [convblock(in_channels,x[0])]
                in_channels = x[0]*3
            elif type(x) == list:
                print("placeholder")
        return nn.Sequential(*list)
    def create_nn(self): #creates the nn used in the model
        return nn.Sequential(nn.Flatten(),
                             nn.Dropout(p=0.5),
                             nn.Linear(5881,50,bias=False),
                             nn.BatchNorm1d(num_features=50),
                             #nn.ReLU(),
                             L2NormalizationLayer(),
                             spline_activation(device = device,grid = 5, input_dim = 50),
                             nn.Dropout(p=0.5),
                             nn.Linear(50,2),
        )


#summary of the model
#model = escargot().to(device)
#summary(model, [(40,1,529,8)])
#a = torch.rand(40,1,529,16).to(device)
#b = torch.rand(40,1,40,40,16).to(device)
#print(model(a,b))
```

A.2. B-spline activation

```
import torch
import torch.nn as nn
import torch.nn.functional as F

#Created/modified B-spline activation function from modification of files in github Repo of
    Kolmogorov-Arnold Networks
#Link: https://github.com/KindXiaoming/pykan/blob/master/kan/spline.py

class Bspline_segment_calc(nn.Module):
    def __init__(self,input_dim,device = "cpu"):
        super().__init__()
        self.device = device
        self.input_dim = input_dim
    #-----Make Grid-----#
```

```
    def make_grid(self,grid = 10):
        grid_knots = torch.linspace(-1,1,steps= grid + 1)
        grid_size  = torch.einsum('i,j->ij',torch.ones(self.input_dim,), grid_knots)
        return grid_size

    #-----Extends grid for accuracy-----#
    def extend_grid(self,grid,extend=True,k=3):
        if extend == True:
            #print(grid.shape)
            h = (grid[:,[-1]] - grid[:,[0]]) / (grid.shape[1] - 1)
            k_extend = k
            for i in range(k_extend):
                grid = torch.cat([grid[:,[0]] - h, grid],dim=1)
                grid = torch.cat([grid, grid[:,[-1]] + h],dim=1)
            grid = grid.to(self.device)
            return grid
        else:
            return grid

    #-----Calculate Bi in B-spline formula-----#
    def Cox_deBoor(self,x,grid,k,extend):
        if extend == True:
            grids = self.make_grid(grid) # make grid
            grids = self.extend_grid(grids,True,k) # extend grid
            grids = grids.unsqueeze(dim=2).to(self.device)
        else:
            grids = grid.unsqueeze(dim=2).to(self.device)
        x = x.unsqueeze(dim=1).to(self.device)
        if k == 0:
            value = (x >= grids[:,:-1]) * (x < grids[:,1:])
        else:
            B_km1 = self.Cox_deBoor(x = x[:,0],grid = grids[:,:,0],k = k - 1,extend = False)
            value = (x - grids[:, :-(k + 1)]) / (grids[:, k:-1] - grids[:, :-(k + 1)]) *
                B_km1[:, :-1] + (
            grids[:, k + 1:] - x) / (grids[:, k + 1:] - grids[:, 1:(-k)]) * B_km1[:, 1:]
        return value

class spline_activation(nn.Module):
    def __init__(self,grid = 10,k = 3,extend = True,device = "cpu",input_dim = 5):
        super().__init__()
        self.grid = grid
        self.k = k
        self.extend = extend
        self.device = device
        self.input_dim = input_dim
        self.Bspline = Bspline_segment_calc(self.input_dim,device = self.device)
        self.size = grid + 1  + (k-1)
        self.coef = torch.nn.Parameter(torch.rand(self.input_dim,self.size))
    def forward(self,x):
        #x = F.normalize(x)
        x = x.T
        B = self.Bspline.Cox_deBoor(x,k=self.k,grid=self.grid,extend=self.extend)
        out = torch.einsum('ij,ijk->ik', self.coef, B)
        return out.T
```

A.3. FILTERING

```
import os
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from Dataloader import DataReader

files = os.listdir("C:\\Users\\Gebruiker\\Desktop\\Bap\\Data")
print(files)
output_directory = "C:\\Users\\Gebruiker\\Desktop\\Bap\\PTData4"

def CAR_filter(logits):
    a,b,c = logits.shape
```

```
    # initialise zero tensor to hold channel average
    channel_average = torch.zeros(1440, 900)

    # compute average for each channel
    for i in range(int(logits.size(0))):
        for j in range(int(logits.size(1))):
            for k in range(int(logits.size(2))):
                channel_average[i][j] += logits[i][j][k]
            channel_average[i][j] = channel_average[i][j] / 22

    # initalise empty tensor to hold filtered logits
    logits_CAR = torch.empty(int(logits.size(0)), int(logits.size(1)), int(logits.size(2)))

    # subtract all readings by channel average to create new 'reference' signals
    for i in range(int(logits.size(0))):
        for j in range(int(logits.size(1))):
            for k in range(int(logits.size(2))):
                logits_CAR[i][j][k] = logits[i][j][k] - channel_average[i][j]

    return logits_CAR

def freq_filter(logits):

    # set parameters
    fs = 250
    T = 1 / fs
    L = 900
    # perform FFT
    fft_logits = torch.fft.fftn(logits, dim=(1,))

    # create frequency mask
    frequencies = torch.fft.fftfreq(L, d=T)
    mask = (frequencies >= 0.5) & (frequencies <= 32)
    full_mask = torch.zeros_like(fft_logits, dtype=torch.bool)
    full_mask[:, :len(mask), :] = mask[None, :, None]
    reversed_mask = torch.flip(mask, dims=[0])
    full_mask[:, -len(mask):, :] = reversed_mask[None, :, None]

    # apply frequency mask
    fft_logits_masked = fft_logits * full_mask

    # perform IFFT
    logits_filtered = torch.fft.ifftn(fft_logits_masked, dim=(1,)).real

    return logits_filtered

for _ in files:

    # set index
    index = _[2]
    print(_)
    # read/load .npz file for user
    npz_loc = "C:\\Users\\Gebruiker\\Desktop\\Bap\\Data\\" + _
    file = DataReader(dataset=npz_loc)
    data = file.load_data()

    # produce tensor for logits and targets
    logits = torch.tensor(np.asarray(data[0]), dtype = torch.float)
    targets = torch.tensor(np.asarray(data[1]), dtype = torch.long)
    logits = logits[:,None,:,:]
    logits = logits.squeeze(1)

    # perform filtering on logits (calling functions written above)
    logits = CAR_filter(logits) # spatial filtering (CAR)
    logits = freq_filter(logits) # frequency filtering
    logits = logits.unsqueeze(1)

    # set file name for output
    user_logits = "logitsU{}.pt".format(index)
    user_targets = "targetsU{}.pt".format(index)

```

```
    # set path for output
    user_logitpath = os.path.join(output_directory, user_logits)
    user_targetpath = os.path.join(output_directory, user_targets)

    if _ == "A01T.npz":
          logitsall = logits
          targetsall = targets
    else:
          logitsall = torch.cat((logitsall,logits),dim = 0)
          targetsall = torch.cat((targetsall,targets),dim = 0)
    # output cleaned/filtered logits and targets
    torch.save(logits, user_logitpath)
    torch.save(targets, user_targetpath)

#torch.save(logitsall, "logits0.5-30-1s")
#torch.save(targetsall, "targets0.5-30-1s")
```

A.4. Training Loop

```
from Dataloader import DataReader
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset
from model4 import model
from escargot_rapidement import kanmodule
from escargot3 import escargot
from torch.utils.data import DataLoader
import numpy as np
import matplotlib.pyplot as plt
import os
from sklearn.model_selection import KFold

seed = 26
torch.manual_seed(seed)
arr = os.listdir("C:\\Users\\Gebruiker\\Desktop\\Bap\\PTData")

#-----HYPERPARAMETERS/training-----#
test_size = 28
batch_size = 64
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
learning_rate = 1e-5
lr_warmup = 1
lr_fin = 1e-3
n_warmup = 0
max_iters = n_warmup + int(1000)
eval_interval = 10
#-----HYPERPARAMETERS/training-----#

#-----DataLoader-----#
#logits = torch.load("C:\\Users\\Gebruiker\\Desktop\\Bap\\PTData3\\logitsU6.pt")
#targets = torch.load("C:\\Users\\Gebruiker\\Desktop\\Bap\\PTData3\\targetsu6.pt")
logits = torch.load("own.pt")
targets = torch.load("owntargets.pt")
logits = logits[:,None,:,:]
t,f,h,l = logits.shape
print(logits.shape)
print(logits)


x = torch.arange(0,72,1)
a = torch.randperm(6)
print(a)
kf = KFold(n_splits =6,shuffle=True)
kf.get_n_splits(a)
train_list = []
test_list = []

for i, (train_index, test_index) in enumerate(kf.split(a)):

```

```
    print(f"Fold␣{i}:")
    train_list.append(train_index)
    print(f"␣␣Train:␣index={train_index}")
    test_list.append(test_index)
    print(f"␣␣Test:␣␣index={test_index}")



#rand_samples = torch.randperm(72)
#samples_test = rand_samples[:int(0.1*72)]
#samples_train = rand_samples[int(0.1*72):]

'''for _ in samples_train:
    for i in samples_test:
        if _ == i:
            print("True")
        else:
            pass'''
acc_list = []
for iii in range(10):
    ind = []
    for i in range(2):
        for _ in train_list[iii]:
            #indixes = _ + (i*72)
            indixes = torch.arange((_*30)+(i*6*30),(_*30)+(i*6*30)+30,1)
            ind.append(indixes)

    indexes = torch.cat(ind)
    logits_train = logits[indexes]
    targets_train = targets[indexes]
    print(logits_train.shape)


    ind = []
    for i in range(2):
        for _ in test_list[iii]:
            #indixes = (_*7)+(i*72*7)
            indixes = torch.arange((_*30)+(i*6*30),(_*30)+(i*6*30)+30,1)# (i*72) #(_*7)+(i
                *72*7)
            ind.append(indixes)

    #print(logits_train.shape)

    t1,l1,h1,f1 = logits_train.shape

    #print(indexes)
    indexes1 = torch.cat(ind)
    #print(indexes1)
    logits_test = logits[indexes1]
    targets_test = targets[indexes1]
    #print(logits_test.shape)
    #print(targets_test)


    for _ in indexes1:
        for i in indexes:
            if _ == i:
                print("True")
            else:
                pass



    torch.save(logits_test,"logits_test.pt")
    torch.save(targets_test,"targets_test.pt")
    from datautils import cropper


    #create dataset for performance
    dataset = torch.utils.data.TensorDataset(logits_train,targets_train)
```

```
    train_size = int(t1*0.9)

    train, test = torch.utils.data.random_split(dataset,[train_size,t1-train_size])
    train = DataLoader(train,batch_size = batch_size,shuffle = True)
    test = DataLoader(test,batch_size = test_size,shuffle = True)


    #-----Warmup-----#
    def warmup(current_step):
            if current_step < 50:
                return lr_warmup
            elif current_step < 100:
                return 1e-1
            elif current_step < 150:
                return 1e-2
            else:
                return 1e-3
    #-----Warmup-----#


    #-----Model-----#
    model = escargot().to(device)
    #model.load_state_dict(torch.load("C:\\Users\\Gebruiker\\Desktop\\Bap\\PTData\\cnnnet2048
        .pt"))
    model.train()
    #-----Model-----#

    #-----optim/loss-----#
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=1e-3)
    loss = torch.nn.CrossEntropyLoss()
    #scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
    #scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer,lr_lambda = warmup)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.7)
    #scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones
        =[600,800,100,1200], gamma=0.1)
    #scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor
        =0.1, patience=10, threshold=0.0001, threshold_mode='abs')
    #-----optim/loss-----#


    llist = []
    tlist = []
    avloss = []
    #-----training loop-----#
    for itere in range(max_iters):
        tf_list,tt_list = logits_test,targets_test
        f_list,t_list = next(iter(train))
        t_list = t_list.type(torch.LongTensor)
        tt_list = tt_list.type(torch.LongTensor)
        if itere % eval_interval == 0 or itere == max_iters - 1:
            with torch.no_grad():
                model.eval()
                out = model(tf_list.to(device,dtype=torch.float))#tf_list.to(device),tff_list
                    .to(device)
                #print(torch.max(out,dim=1))
                values,ind = torch.max(out,dim = 1)
                g = tt_list.shape
                #print(g)
                a = np.sum((torch.eq(ind.to("cpu"),tt_list.to("cpu")).numpy()))
                #print(a)
                accuracy = (a/g)*100
                tlist.append(accuracy)
                avgloss = (np.sum(avloss)/(len(avloss)))
                progress = (itere/max_iters) * 100
                print("accuracy : {}, validation loss : {}, progress : {}%, lr : {}".format(
                    accuracy, avgloss, int(progress), scheduler.get_last_lr()))
                avloss = []
                if itere == 0:
                    print(" ")
                else:
                    llist.append(avgloss)
```

```
188         else:
189             model.train()
190             inputs = model(f_list.to(device,dtype=torch.float))#,ff_list.to(device)batch_list
                    .to(device)
191             print(inputs[0])
192             with torch.no_grad():
193                 val_input = model(tf_list.to(device,dtype=torch.float))#test_list.to(device),
                        tff_list.to(device)
194             lossvalue = loss(inputs, t_list.to(device))
195             print(lossvalue)
196             vallvalue = loss(val_input.to(device),tt_list.to(device))
197             avloss.append(vallvalue.data.cpu().numpy())
198             optimizer.zero_grad(set_to_none=True)
199             lossvalue.backward()
200             optimizer.step()
201             scheduler.step()
202     #-----training loop-----#
203     acc_list.append(accuracy)
204 torch.save(model.state_dict(),'blockblock.pt')
205 print(acc_list)
206 print(np.sum(acc_list)/10)
```