Technical University of Denmark

Principles of brain computer interface

Group 19

Enhancing SSVEP-Based Brain-Computer Interfaces: A Deep Learning Approach to EEG Signal Classification

Matilde Correia, s242905
Pedro Vieira, s240181

January 21, 2025

Contents

1 Abstract 1

2 Introduction and Motivation 2

3 Related Work 3

4 Results and Discussion 4

4.1 Signal processing and feature extraction 4

4.1.1 Filtering 4

4.1.2 Time-Domain Features 4

4.1.3 Frequency Domain Features 5

4.1.4 Time-Frequency Domain Features 6

4.2 Deep learning 7

4.3 Architecture 9

4.4 Training and results 10

4.5 Visualizing weights 12

5 Conclusion 14

References 15

A Appendix A - CNN definition, training and evaluation script 16

B Appendix B - model reader and weight visualization script 27

1 Abstract

The report explored the application of convolutional neural networks (CNNs) for the classification of steady-state visually evoked popetentials (SSVEPs) in brain-computer interfaces (BCIs).The main goal was to enhance classification accuracy and demonstrate the potential of deep learning methodologies in processing neural signal data.

Using a benchmark dataset consisting of 64-channel EEG recordings from 35 participants engaged in a cue-guided target selection task. Firstly, we employed signal processing techniques and feature extraction methods, in order to perform profiling and visualize the data. These steps included band-pass filtering, power spectral density (PSD) analysis, and wavelet transforms.

Finally, an end-to-end CNN-based classifier was designed and implemented in PyTorch. The architecture incorporated temporal and spatial convolutions, dropout layers, batch normalization, and data augmentation strategies to improve performance and generalization. Hyperparameter optimization and ablation studies were conducted to validate the effectiveness of various model components.

With the deep learning approach, we obtained validation accuracies of 97.50% and 96.67% for the 2 proposed architectures, with satisfactory training curves, demonstrating very little overfitting, good discrimination between classes and good generalization to unseen data. These results are very satisfactory, and demonstrate the incredible ability of CNNs to perform end-to-end classification on EEG signals, without prior domain knowledge or feature extraction.

2 Introduction and Motivation

Disabilities may arise from genetic conditions, illnesses, accidents, environmental factors, or unknown causes, exhibiting a wide range of symptoms and degrees of severity. Therefore, severe disabilities, especially those leading to complete loss of movement and communication, require the use of advanced assistive technologies. [1]

The field of brain-computer interfaces (BCIs), an emerging human-computer interaction technology used to communicate between the human brain and computers, has experienced significant advancements, due to the increasing availability of high-quality datasets and sophisticated machine learning algorithms. [2]

Usually, BCIs consist of three main parts: brain signals and data acquisition, feature extraction and classification algorithm, and command translation and application. [1]

One prominent domain within BCIs is the study of steady-state visually evoked potentials (SSVEPs), which are elicited by visual stimuli at a range of specific frequencies. SSVEP-based BCIs have garnered attention for their potential in applications such as assistive technologies, gaming, and rehabilitation. While traditional classification methods like support vector machines (SVMs) and linear discriminant analysis (LDA) have been extensively employed for SSVEP data, recent progress in deep learning offers a great opportunity to explore convolutional neural networks (CNNs) as an alternative.

This report was based on the dataset presented in "A Benchmark Dataset for SSVEP Based Brain-Computer Interfaces"[3]. It consisted of 64-channel EEG recordings from 35 healthy participants, which were engaged in a cue-guided target selection task using a 40-target speller interface.

Each participant completed six blocks, with each block containing 40 trials corresponding to distinct targets. During the trials, participants focused on flickering stimuli displayed on a screen at frequencies ranging from 8 Hz to 15.8 Hz, with a 0.2 Hz interval. Each stimulus lasted 5s, and EEG data was synchronized with the onset and offset of the visual stimuli. The dataset was preprocessed to extract 6-second epochs and down-sampled to 250 Hz.

In our case, by using a CNN-based approach, we aimed to enhance classification performance and feature extraction.

The motivation for this work emerged from two main goals: improving classification accuracy in SSVEP-based BCIs and demonstrating the potential of CNNs in processing neural signal data. Conventional methods often rely on manual features and domain expertise, which can limit scalability and adaptability. CNNs, on the other hand, have shown remarkable success in other domains such as image and speech recognition by autonomously learning feature representations.

SSVEP-based BCIs provide a non-invasive communication method, however their practical implementation faces obstacles like noise, neural response variability, and the need for reliable classification techniques. By applying CNNs in this field, we aim to tackle these issues and expand the potential of SSVEP-based BCIs. This study also contributes to advancing deep learning applications in neurotechnology, fostering the development of innovative assistive solutions.

3 Related Work

The development of steady-state visual evoked potential (SSVEP)-based brain-computer interfaces (BCIs) has seen significant advancements, over the last decades, focusing on improving communication for individuals with restricted or none motor capabilities.

A notable contribution is the BCI Speller system developed by DTU [4], which integrates SSVEP responses with dictionary support for efficient text generation. This system utilizes a minimalistic setup with three electrodes and uses a two-stage selection model. Using a dictionary for word prediction, it significantly improves communication speed, achieving an average character production rate of 4.91 characters per minute (CPM). The DTU BCI Speller system demonstrates robust usability, even under non-ideal conditions, emphasizing user-friendly design and intuitive interfaces.

Complementing these efforts, recent advances in SSVEP detection have focused on deep learning methodologies. For example, a 2022 study published in *Frontiers in Computational Neuroscience* introduced a "CNN-based approach to the detection of SSVEP using binaural EEG" [5]. The mentioned method utilizes convolutional neural networks (CNNs) to extract relevant features from ear-centered EEG signals, offering a compact and portable alternative to traditional setups. By targeting the unique challenges of ear-EEG, this approach expands the applicability of SSVEP-based BCIs to scenarios requiring minimalistic and portable designs. The CNN-based model demonstrated superior performance in classifying SSVEP stimuli compared to conventional methods, showcasing the importance and potential of deep learning to improve detection accuracy and system adaptability.

Both of these works, collectively, demonstrate the rapid evolution of SSVEP-based BCIs driven by user-centric innovations and cutting-edge computational techniques. Our study aligns with this trajectory by combining the extraction of features from EEG data with a CNN classifier, with the aim of improving the precision and usability of SSVEP systems.

4 Results and Discussion

A conventional approach to signal processing and descriptive feature extraction was employed primarily to profile the dataset at hand. However, for the specific task of automatically classifying EEG signals, a more comprehensive, end-to-end deep learning methodology was adopted.

4.1 Signal processing and feature extraction

Signal processing and feature extraction are critical steps in EEG analysis since they transform raw, noisy brain activity signals into meaningful representations suitable for interpretation and further analysis.

EEG signals are complex and susceptible to various artifacts such as muscle movements, eye blinks, and environmental noise, which can obscure the underlying neural information.

Therefore, processing techniques, such as filtering and artifact removal, help to clean the data and isolate relevant brain activity. Feature extraction, on the other hand, simplifies high-dimensional EEG data by identifying and quantifying key patterns.

4.1.1 Filtering

The data of the EEG signals was first submitted to a band-pass filter, which removed unwanted noise by isolating the frequency range associated with SSVEP stimuli, this being 8â30 Hz.

Human EEG signals typically range from 1â30 Hz, which is the primary focus of most studies. Low frequency noise appears as slow drifts in the EEG signal over many seconds. In contrast, high frequency noise comes from sources including electromagnetic interference, and muscle contractions (especially facial and neck muscles). High frequency noise looks like spikes in the EEG.

However, since the stimulation frequencies from A Benchmark Dataset for SSVEP-Based BrainâComputer Interfaces ranged from 8-15.8 Hz, the chosen low-cut and high-cut of the band-pass filter were, respectively, set to 7 Hz and 30 Hz.

4.1.2 Time-Domain Features

Time-Domain Features, such as Mean (1), Peak-to-Peak Amplitude (2), Peak Amplitude (3) and Variance (4), were extracted from the raw EEG signal over time.

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{1}$$

$$\text{Peak-to-Peak Amplitude} = \max{[x_n]} - \min{[x_n]} \tag{2}$$

$$\text{Peak Amplitude} = \max{[x_n]} \tag{3}$$

$$\text{Variance} = \sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2 \tag{4}$$

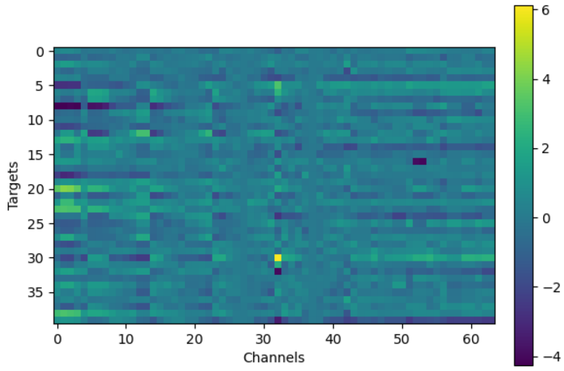

Figure 1: Mean

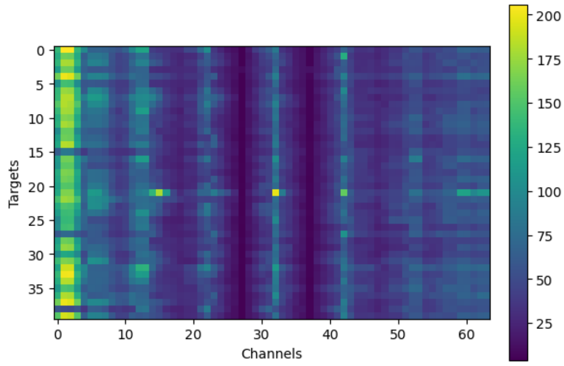

Figure 2: Peak-to-Peak Amplitude

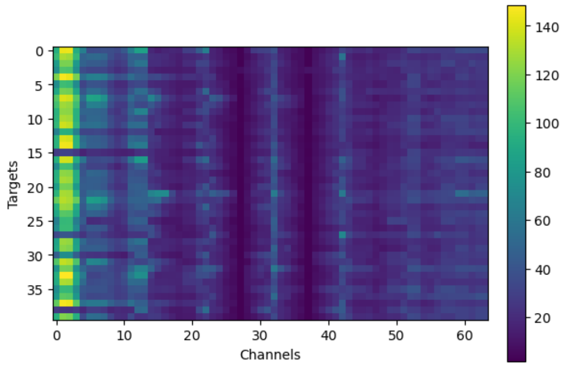

Figure 3: Peak Amplitude

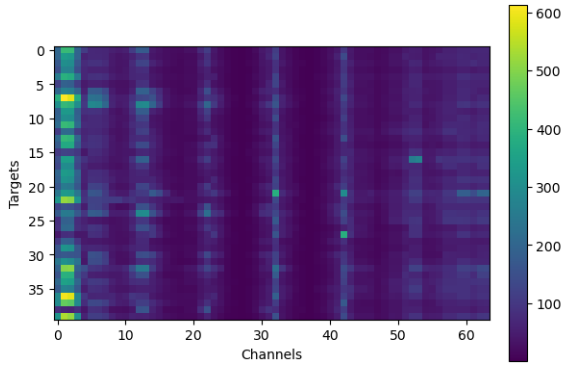

Figure 4: Variance

Figure 5: Time-Domain Features of EEG signals of subject S1, with targets in function of channels

Figure 5 displays the time-domain features extracted from EEG signals of subject S1, plotted as a function of channels. These features, provide insights into the magnitude, range, and stability of the neural signals during the SSVEP task. Channels with higher values in these metrics indicate stronger or more variable brain activity, which could correspond to regions of interest for classification.

The channels 59, 60, 62, and 63, located near the occipital region (critical for visual processing), show notable activity. This aligns with the taskâs focus on visual stimuli, as these areas are predominantly involved in detecting SSVEP responses. Variations in mean amplitude and peak-to-peak values across channels suggest that neural signals are not uniformly distributed, emphasizing the importance of selecting relevant channels for analysis.

4.1.3 Frequency Domain Features

For the Frequency Domain Feature extraction, the Power Spectral Density (PSD) was utilized.

PSD measures the power of EEG signals across different frequencies, enabling the identification of dominant components corresponding to visual stimuli. PSD is particularly crucial in SSVEP-based Brain-Computer Interfaces (BCIs) as it highlights power distribution and leverages harmonics to improve detection accuracy.

Additionally, PSD helps evaluate the strength of SSVEP responses under various conditions, making it a reliable tool for feature extraction and performance assessment. By isolating key frequency

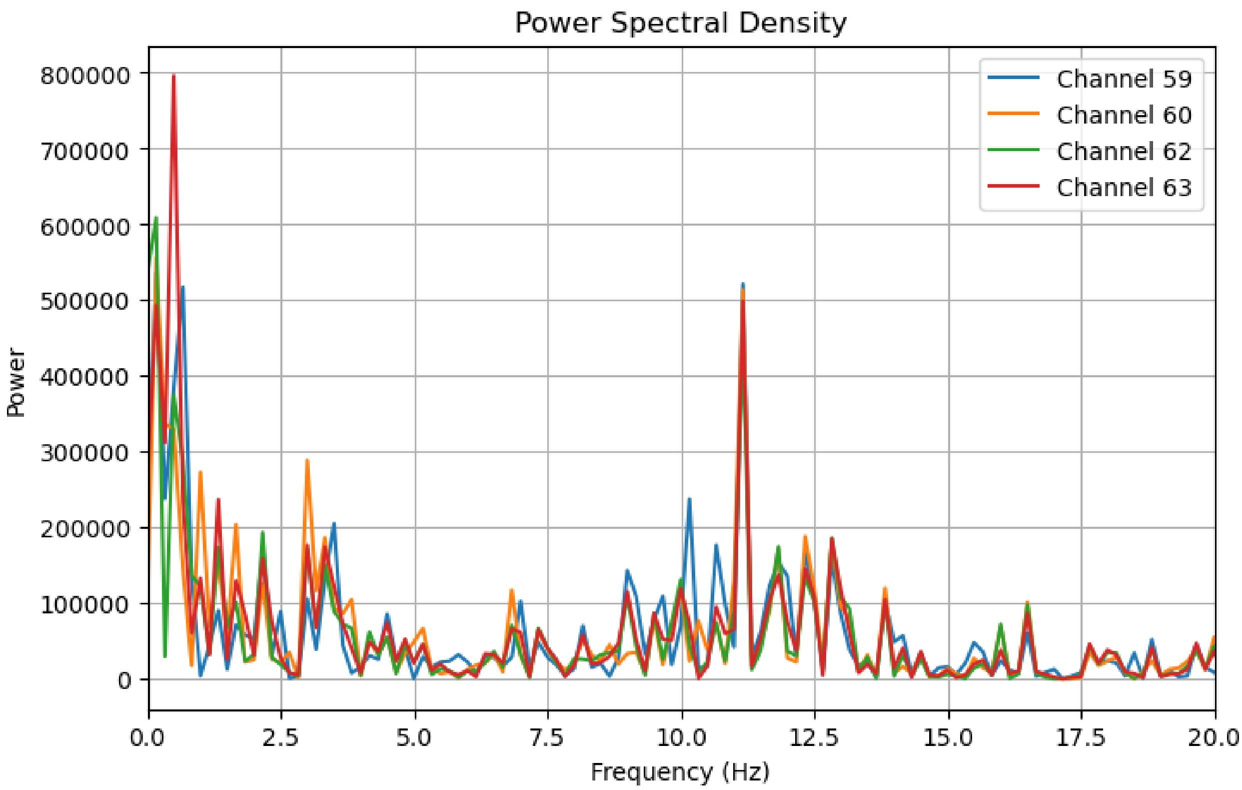

Figure 6: Power Spectral Density of the channels 59,60,62 and 63

components, PSD ensures effective and efficient SSVEP detection, which is critical for real-time control and communication tasks.

According, to the analysis by the CNN (see section 4.5) the most relevant channels (i.e. the channels with the biggest weights associated with them) are: 59,60,62 and 63.

4.1.4 Time-Frequency Domain Features

Lastly, the EEG signals of the data set were analyzed using the Wavelet Transform (WT).

WT is a crucial tool for time-frequency analysis, offering the ability to decompose EEG signals into time-localized frequency components, which is essential for analyzing non-stationary signals like EEG, where the frequency content changes over time.

WT enables detailed time-frequency localization, allowing the detection of transient features such as SSVEP responses, while its analysis ensures adaptive exploration of low and high-frequency components. This is particularly beneficial for identifying fundamental SSVEP frequencies and harmonics, while minimizing noise interference, thereby improving the signal-to-noise ratio (SNR).

The following figure demonstrates the Raw EGG Signal and its respective Wavelet Transform, for channel=61, target=1, and block=1:

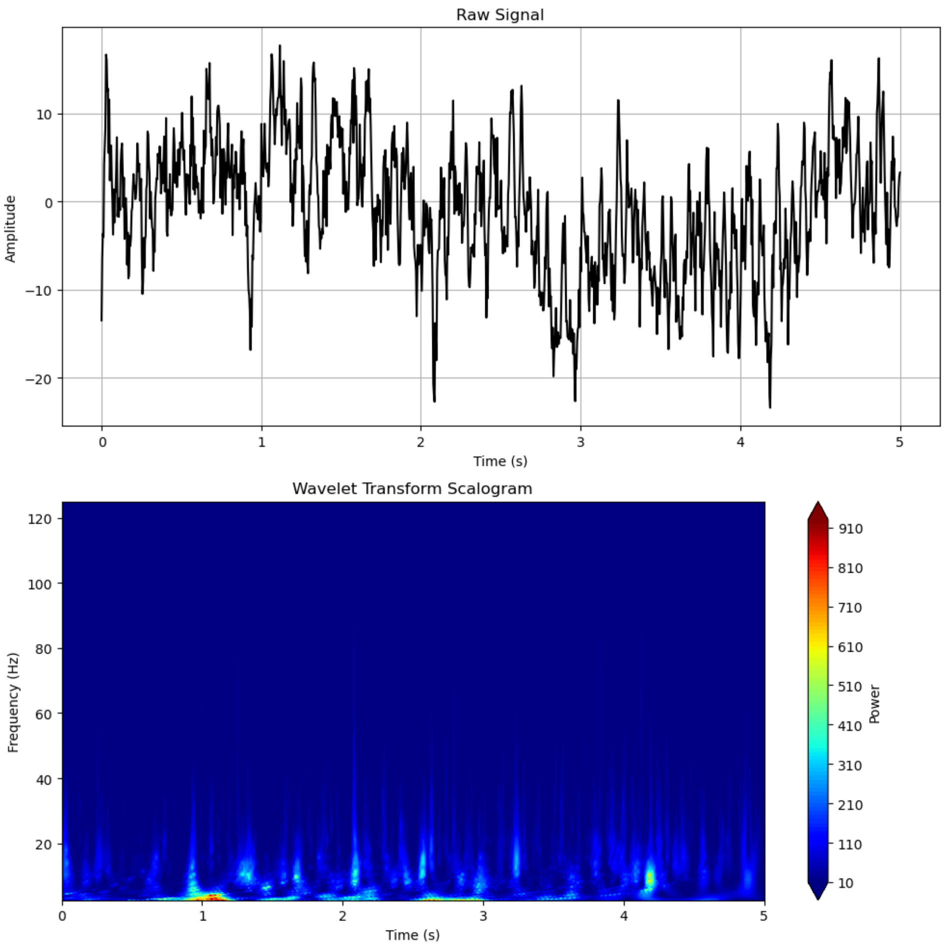

Figure 7: Wavelet Transform Scalogram and EEG Raw Signal

The scalogram revealed that most of the signal's power is concentrated in the low-frequency range (below 20 Hz), with transient bursts of activity around 1 second and 4 seconds. Minimal high-frequency activity above 40 Hz suggests either preprocessing removed noise or the signal lacks such components. Overall, the scalogram effectively captures the time-frequency dynamics, highlighting dominant frequencies and transient events for further analysis.[6][7]

Compared to Fourier-based methods, WT is more effective for analyzing non-stationary signals as its outputs reveal energy distributions across time and frequency, highlighting SSVEP stimulation frequencies and harmonics. This makes

4.2 Deep learning

A more end-to-end, black-box, deep learning approach was taken for the actual task of classification of the signals. The dataset [3] consisted of 6 trials per target per subject, each trial is 64 channels by 1500 time instants. As said, the task was to develop a classification algorithm to map each trial to the corresponding target the person was looking to.

From the literature, and even though there is no consensus on what the optimal architecture is for each EEG deep learning task, the following points were generally true:

- Activation functions - from literature reviews, [8, 9], there was no global consensus on what optimal shape function to use, but ReLU, ELU, sigmoid and tanh were the most common, in this order. These introduce non linearity and more expression on the results. ReLU and sigmoid are more prone to gradient vanishing, so ELU was chosen here.
- Dropout - Almost all the models analyzed in [8] used at least one dropout layer, with varying dropout probabilities. Dropout skips some connections and convolutional layers with a certain probability. It can slow down the learning a little bit, but greatly increases the generalization capability of the model, making it perform better on new data.
- Batch normalization - the paper in [10] first proposed batch normalization layers to greatly increases accuracy of CNNs for EEG P300 tasks. We were also able to prove this for SSVEP (Tab.**??**, line ??). This is because normalization brings the values of each layer closer to a normal distribution, making it easier for the model to learn.
- Augmentation - [11] discusses how data augmentation increases stability, accuracy and reduces overfitting in EEG data. In this study we added Gaussian noise and amplitude scaling data transformations, with a chance of 50% each.
- Pooling - pooling is used to reduce the spacial dimension of the input, reducing the complexity of the learning algorithm. From the literature [8], both max and average pool were commonly used.
- Number of convolutional layers - common literature also uses a range of convolutional layers, from 1 to 4 time convolutions and almost always just one spatial convolutional layer.
- Kernel dimensions - Small kernels allow the network to learn small local changes, and larger kernels infer on more global features. According to [8], the preferred kernel size from the literature is less than 1×25. Bigger kernels were used and reported here if for some reason they provided better results.
- Optimizer, regularization and learning rate - from [8], the most used optimizer is still Adam (with gradient descent and stochastic gradient descent as runner ups). A base learning rate of $10^{-}3$ was used, and the λ parameter for the L2 regularization (*weight_ decay* in Torch) was set to 0.05.

We were able to test most of these with ablation studies on our proposed network to validate the impact of each of the components on the machine learning result. However, the optimal set of hyper parameters for each machine learning task is generally a result of trial and error, following good machine learning practices, with some lucky guesses.

4.3 Architecture

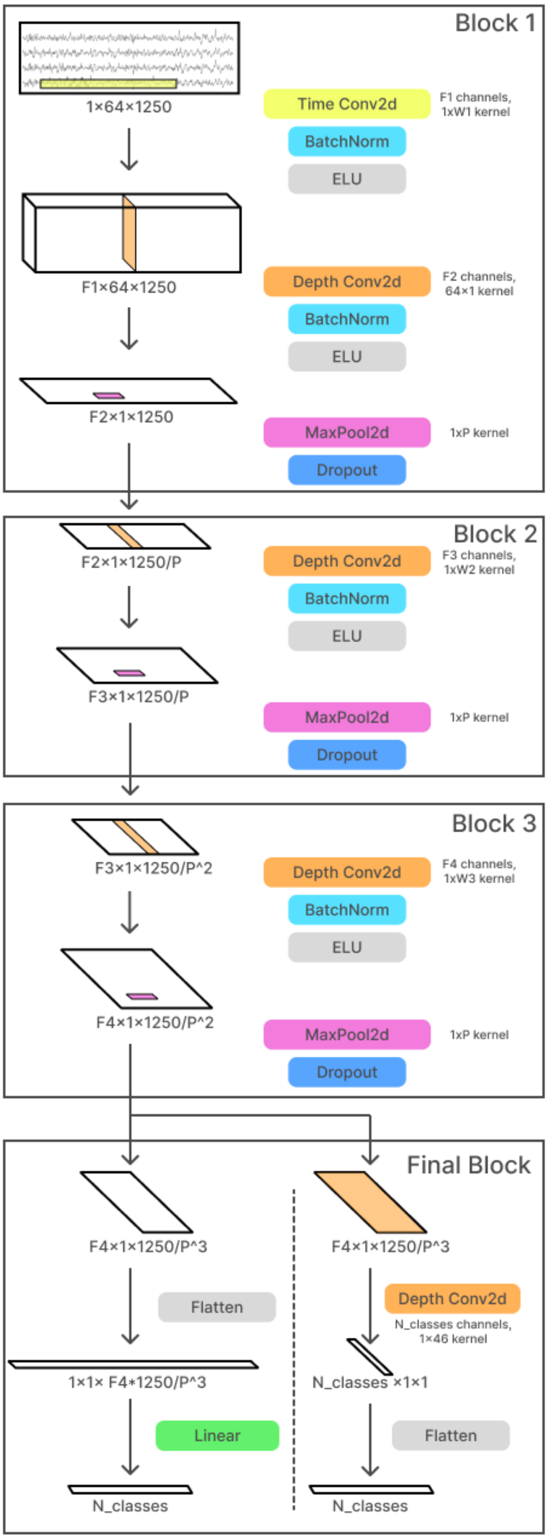

Figure 8: Schema for the proposed network architecture, with general parameters (W_i=size of kernel i, F_i=number of channels of convolution i, P=size of pool kernel

Before arriving at the final architecture, a lot of alternative networks were tested. These were directly proposed from other works, adaptations from those, or proposed by generative AI. The results from these won't be reported, as they were worse and harder to fine-tune. Based in the literature mentioned before, performing some modifications or rewriting new CNN architectures, the final proposed architecture was derived (Fig 8), and implemented in Pytorch. This network is totally end-to-end, with no preprocessing, and mostly inspired by [12], with one less block. It has a very general architecture of 1 temporal convolution, one depthwise spatial convolution, 2 more depthwise temporal convolutions, and a total of 3 pool layers, plus activations, dropout and batch normalization layers, with a final block with 2 options, that will be explained further. The network has a lot of customizable parameters used for hyperparameter fine-tuning, like the size of convolutional and pool kernels, number of channels and the padding.

Regarding the final block, the result from the convolutional layers can be processed in 2 different ways to predict the final classes. The first option consists of a simple fully connected linear layer followed (or not) by a softmax activation (to turn the predictions into probabilities). This is a very common approach in literature, and was also done by [12].

The second approach, inspired by [13] used a final convolutional layer (network with no linear layers), that applies one depthwise kernel on the **entirety** of the previous layer's results per each class. This maps a shape of F_4x1x46 directly into a $F_4 \times 1 \times 46$ $N_{classes} \times 1 \times 1$ vector, used directly for the prediction of the class. This approach used in [13] was for multi-target classification on the same dataset, but it actually proved to have the really good results in our classification task.

4.4 Training and results

The proposed network options were trained for 20 epochs, with only 10 subjects data (due to lack of disk space to run with every subject). With this, we had 2400 trials, with 6 trials per patient per class. A 80-20 train-validation split was performed for the training setup, with a batch size of 16. Data augmentation was preformed: when loading a trial for training, there is a 50% chance of including Gaussian noise or amplitude scaling. Training was performed on DTU's High Performance Computing cluster (HPC), with GPU acceleration and CUDA support.

Training and validation accuracy results from training the network with different parameters are shown in tables 1 and 2.

For the first option, a setting with a 1×2 pool, padding set to "same" and convolutional kernels of roughly 1/10 the size of that layer achieved better results. For the second option for the last block, a pool kernel of size 1 (meaning no pool is performed, like in [13]), with smaller number of channels (small F1, F2, F3 and F4) and small kernels (small W1, W2 and W3) provided better results. Interestingly, this setup uses less weights (10 times less parameters than the best model found for option 1), but achieves very similar results and really good generalization capabilities and relatively fast learning **??**.

Changing the pool size, dropout rate and the inclusion of batch normalization layers had a big impact on the performance for both options. The number of out channels in the last layer (F4) and the type of padding ("valid" or "same") also impacted the results, to a lesser extent. For both options, the huge majority of the model's parameters are concentrated on the last block, either in the fully connected or the convolutional layer.

Model	F_1	F_2	F_3	F_4	W_1	W_2	W_3	Pool	Dropout	Padding	# Parameters	Training Accuracy	Validation Accuracy	ITR
1	10	20	40	80	39	19	19	2	0.5	same	503,490	100.00%	95.21%	57.49
1	10	20	40	80	121	59	19	2	-	same	505,910	100.00%	96.25%	58.72
1	10	20	40	80	121	59	11	2	-	same	505,270	100.00%	**97.50%**	60.25
1	10	20	40	80	121	59	11	2	-	same	504,970	100.00%	93.75%	55.85
1	10	20	40	80	124	61	31	2	-	**valid**	359,780	100.00%	94.17%	56.31
1	10	20	40	40	124	61	31	2	-	same	256,180	100.00%	96.88%	59.47
1	5	10	20	40	124	61	31	2	-	same	249,640	100.00%	96.46%	58.97
1	10	20	40	80	61	61	19	2	-	same	505,390	100.00%	95.63%	57.98
1	10	20	40	80	31	31	19	2	-	same	503,890	100.00%	96.46%	58.97
1	10	20	40	80	124	41	13	**3**	-	same	152,740	95.31%	15.83%	2.92
1	10	20	40	80	39	19	19	**3**	-	same	151,490	85.42%	15.42%	2.77
1	20	20	40	80	59	19	19	**3**	-	same	120,300	69.38%	7.08%	0.50
1	10	20	40	80	39	19	19	**4**	-	same	65,080	53.12%	6.04%	0.32
1	10	20	40	80	39	11	5	**4**	-	same	63,650	39.38%	5.63%	0.26

Table 1: Summary of CNN model configurations and accuracy results for option 1 with best model

Model	F_1	F_2	F_3	F_4	W_1	W_2	W_3	Pool	Dropout	Padding	# Parameters	Training Accuracy	Validation Accuracy	ITR
2	5	5	5	40	101	19	19	1	0.5	valid	46,350	99.90%	96.04%	58.47
2	5	5	5	40	59	19	19	1	-	valid	47,820	99.79%	96.25%	58.72
2	5	5	10	40	101	19	19	1	-	valid	46,455	99.84%	**96.67%**	59.22
2	5	10	20	40	59	19	19	1	-	valid	48,465	99.90%	95.42%	57.74
2	10	20	40	80	59	19	19	1	-	valid	96,930	99.95%	93.75%	55.85
2	10	20	40	80	101	19	19	1	-	valid	93,990	99.90%	94.38%	56.55
2	10	20	40	80	101	39	19	1	-	valid	93,190	99.43%	94.17%	56.32
2	20	40	80	160	59	19	19	1	-	valid	193,860	98.96%	88.75%	50.64
2	5	10	20	40	59	19	19	**2**	-	valid	7,625	38.39%	30.41%	9.09

Table 2: Summary of CNN model configurations and accuracy results for option 2 with best model

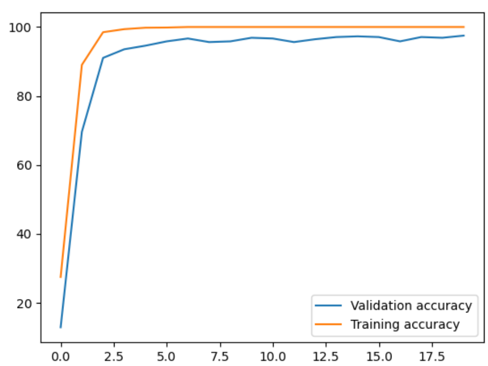

Figure 9: Option 1

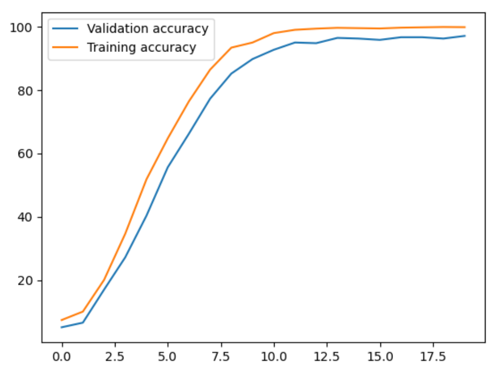

Figure 10: Option 2

Figure 11: Training curves for the best parameter for both options

The learning curves for the best model for both options (Fig.11.a and Fig.11.b) also demonstrate a lack of overfitting, as the training and validation accuracies in each epoch go up steadily. For option 1, with considerably more parameters, the accuracies rose very fast: after 5 epochs the training accuracy was already above 99%. Both accuracies plateaued after 10 epochs for these models, with the validation accuracy staying consistently below the training accuracy.

Finally, we evaluated the best model of both options against a 11th subject's data that wasn't used for training. They both reported an accuracy of >95% on this data, further confirming the lack of overfitting and the ability to classify and generalize the learning process.

4.5 Visualizing weights

In this section, we developed another Python script to load the trained models, extract the learned weights on each layer and visualize them.

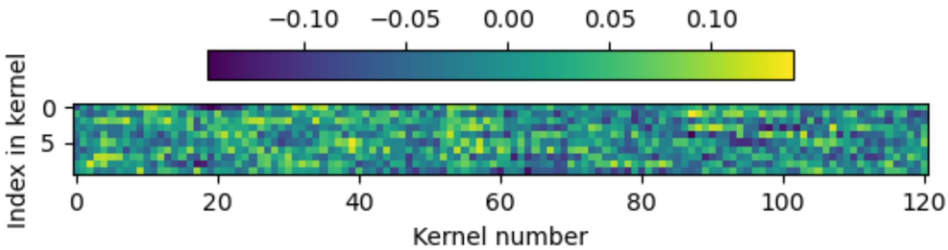

Figure 12: Weights for the first temporal layer for the best setup of option 1

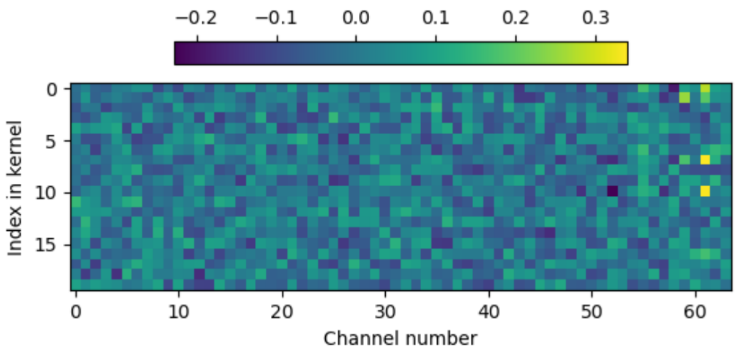

Figure 13: Weights for the first depthwise spatial layer for the best setup of option 1

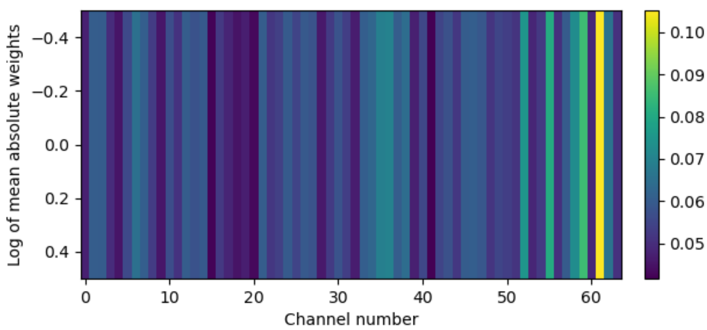

Figure 14: Mean of the absolute values for the weights of the first depthwise spatial layer, for the best setup of option 1

Fig.12 shows the learned weights for the first temporal convolution in the best setup for option 1. Each row represents a kernel. Fig.13 shows the learned weights for the first depthwise convolution. Then, Fig.14 shows the mean of the absolute value of these weights for each channel, in a barcode format. This plot allows us to pinpoint which electrode channel the model pays more attention to. From that, electrodes 59, 60, 62 and 63, with 62 being exceptionally strong. These correspond to electrodes PO8, CB1, Oz and O2 respectively, all on the occipital region, which is responsible for visual perception in the brain. This is a very relevant finding, as it demonstrates that even without imputing prior knowledge of how the brain functions (by only using the signals from the electrodes in the occipital region for the classification, for example), the network is able to learn that on its own. It's hard to draw conclusions from the weights of the other layers, as there is no apparent pattern in them, so they wont be shown. Aditionally, the weights visualized and conclusions were very similar for the second architecture option.

5 Conclusion

The goal of this project was to develop a classification model, either by traditional feature extraction, classification, or deep learning, to improve the accuracy of a SSVEP based BCI interface, on a public dataset [3].

In this project, the results relative to signal processing and feature extraction revealed the importance of these techniques, such as band-pass filtering and wavelet transforms, in improving signal quality, and how fundamental these steps are in traditional classifications.

Two CNN architectures were developed and thoroughly tested, providing remarkably good results. We learned a lot about hyperparameter finetuning, common practices, CNN architectures used in EEG signal processing, namely for SSVEP, and further developed out skills with Pytorch. The finetuning process consists mostly of trial and error with educated guesses, but we found it relevant to see in real time how a change in the network's configuration affected its performance.

Provided we had more time, we would first train the networks with more subjects, as due to a time and disk space shortage on the HPC system, only 10 of the 35 subjects were used to train all the networks. We would expect the accuracy to decrease a little when we train with all subjects. However, the learned models still provided good test accuracy on a 11th subject, so we could assume it didn't overfit completely to the 10 training subjects.

Finally, a suggested further development could be using a mix of end-to-end and classic feature extraction as a basis for the classification algorithm. This could incorporate the power of deep learning techniques with the model knowledge given by traditional features. Applying dimensionality reduction, via PCA for example, or with a learned autoencoder, could also produce good results and better understanding of the signal.

Furthermore, the integration of adaptive learning mechanisms, that allow the personalization of the CNN's parameters and predictions based on individual user data, ensuring that the system is robust to different subjects and remains effective across a wide range of users

References

[1] N. Siribunyaphat, Y. Punsawad, Steady-state visual evoked potential-based brain–computer interface using a novel visual stimulus with quick response (qr) code pattern, Sensors 22 (4) (2022) 1439.

[2] L. F. Nicolas-Alonso, J. Gomez-Gil, Brain computer interfaces, a review, sensors 12 (2) (2012) 1211–1279.

[3] Y. Wang, X. Chen, X. Gao, S. Gao, A benchmark dataset for ssvep-based brainâcomputer interfaces, IEEE Transactions on Neural Systems and Rehabilitation Engineering 25 (10) (2017) 1746–1752.

[4] A. Vilic, T. W.Kjaer, C. E. Thomsen, S. Puthusserypady, H. B.D. Sorensen, Dtu bci speller: An ssvep-based spelling system with dictionary support, Conference proceedings: ... Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Conference 2013 (4) (2013) 2212–2215.

[5] P. Israsena, S. Pan-Ngum, A cnn-based deep learning approach for ssvep detection targeting binaural ear-eeg.
URL `https://www.frontiersin.org/journals/computational-neuroscience/articles/10.3389/fncom.2022.868642/full`

[6] M. X. Cohen, Analyzing Neural Time Series Data: Theory and Practice, MIT Press, 2014.

[7] P. J. Durka, Time-Frequency Analysis of EEG Signals, CRC Press, 2007.

[8] D. Xu, F. Tang, Y. Li, Q. Zhang, X. Feng, An analysis of deep learning models in ssvep-based bci: a survey, Brain Sciences 13 (3) (2023) 483.

[9] I. Rakhmatulin, M.-S. Dao, A. Nassibi, D. Mandic, Exploring convolutional neural network architectures for eeg feature extraction, Sensors 24 (3) (2024) 877.

[10] M. Liu, W. Wu, Z. Gu, Z. Yu, F. Qi, Y. Li, Deep learning based on batch normalization for p300 signal detection, Neurocomputing 275 (2018) 288–297.

[11] E. Lashgari, D. Liang, U. Maoz, Data augmentation for deep-learning-based electroencephalography, Journal of Neuroscience Methods 346 (2020) 108885.

[12] R. T. Schirrmeister, J. T. Springenberg, L. D. J. Fiederer, M. Glasstetter, K. Eggensperger, M. Tangermann, F. Hutter, W. Burgard, T. Ball, Deep learning with convolutional neural networks for eeg decoding and visualization, Human brain mapping 38 (11) (2017) 5391–5420.

[13] H. J. Khok, V. T. C. Koh, C. Guan, Deep multi-task learning for ssvep detection and visual response mapping, in: 2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC), IEEE, 2020, pp. 1280–1285.

A Appendix A - CNN definition, training and evaluation script

Listing 1: CNN definition

```
## VISUALIZATION FILE


import torch
from torch import nn

import scipy.io
from scipy import signal
import matplotlib.pyplot as plt
import numpy as np

import sys
print(sys.path)


import torch
from torch import nn
import scipy.io
from scipy import signal
import matplotlib.pyplot as plt
import torch
import numpy as np


SOURCEFOLDER = "/tmp/"
SOURCEFOLDER = ""

print(torch.version.cuda)
print(torch.__version__)
print(torch.cuda.is_available())


import os
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader, TensorDataset
from scipy.io import loadmat
from torchsummary import summary

## TEST TRAIN SPLIT
from torch.utils.data import random_split

#import torch.utils.data as data_utils

import time
start_time = time.time()
```

```python




### ============================================================================
### DATA LOADER

class EEGDatasetAugmented(Dataset):
    def __init__(self, data_dir, n_subjects, augment=False):
        """
        Initialize the dataset by loading and preprocessing EEG data.

        Args:
        - data_dir: Path to the directory containing .mat files.
        """
        self.data = []
        self.labels = []
        #self.n_subjects = n_subjects
        self.augment = augment
        self._load_data(data_dir, n_subjects)

    def _load_data(self, data_dir, n_subjects):
        """
        Load and preprocess EEG data from .mat files.

        Args:
        - data_dir: Directory containing .mat files.
        """
        #n_subjects = 2 ##NUMBER OF SUBJECTS TO USE <40
        i_subj = 0
        for file in os.listdir(data_dir):
            if file.endswith(".mat"):
                mat_data = loadmat(os.path.join(data_dir, file))
                eeg_data = mat_data['data']  # Adjust key based on your .mat file
                # eeg_data shape: [64, 1500, 40, 6]
                ## CROP
                eeg_data = eeg_data[:,124:1500-125-1,:,:]

                num_electrodes, num_timepoints, num_classes, num_blocks = eeg_data.shape

                # Reshape into trials: [num_classes * num_blocks, 64, 1500]
                for block in range(num_blocks):
                    for target in range(num_classes):
                        trial = eeg_data[:, :, target, block]
                        self.data.append(trial)
```

```
                        self.labels.append(target)
                i_subj += 1
                if(i_subj >= n_subjects):
                    break

        # Convert lists to numpy arrays for PyTorch compatibility
        self.data = np.array(self.data)  # Shape: [num_trials, 64, 1500]
        self.labels = np.array(self.labels)  # Shape: [num_trials]

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        trial = self.data[idx]

        if self.augment:
            trial = self.apply_augmentation(trial)

        trial = torch.tensor(trial, dtype=torch.float32)  # Shape: [64, 1500]
        label = torch.tensor(self.labels[idx], dtype=torch.long)
        return trial, label


    def apply_augmentation(self, signal):
        """
        Apply random augmentations to the EEG signal.
        Args:
            signal (np.ndarray): EEG signal of shape (64, 1500).
        Returns:
            np.ndarray: Augmented EEG signal.
        """
        if np.random.rand() < 0.5:  # 50% chance to add Gaussian noise
            signal = self.add_gaussian_noise(signal)
        #if np.random.rand() < 0.5:  # 50% chance to apply time shift
        #    signal = self.time_shift(signal)
        if np.random.rand() < 0.5:  # 50% chance to scale amplitude
            signal = self.amplitude_scaling(signal)
        #if np.random.rand() < 0.5:  # 50% chance to apply random masking
        #    signal = self.random_masking(signal)
        return signal

    def add_gaussian_noise(self, signal, std=0.01):
        """Add Gaussian noise to the EEG signal."""
        noise = np.random.normal(0, std, signal.shape)
        return signal + noise

    def time_shift(self, signal, max_shift=50):
        """Apply a circular time shift to the EEG signal."""
```

```
        shift = np.random.randint(-max_shift, max_shift)
        return np.roll(signal, shift, axis=1)

    def amplitude_scaling(self, signal, scale_range=(0.9, 1.1)):
        """Randomly scale the amplitude of the EEG signal."""
        scale = np.random.uniform(*scale_range)
        return signal * scale

    def random_masking(self, signal, mask_prob=0.1):
        """Randomly mask parts of the EEG signal."""
        mask = np.random.rand(*signal.shape) > mask_prob
        return signal * mask


# Example usage:
data_dir = SOURCEFOLDER + "data"
NSUBJECTS = 10
#eeg_dataset = EEGDataset(data_dir, 15)
eeg_dataset = EEGDatasetAugmented(data_dir, NSUBJECTS, augment=True)
##eeg_dataloader = DataLoader(eeg_dataset, batch_size=32, shuffle=True) ## IGINORAR



total_samples = len(eeg_dataset)
print(f"Total samples in the dataset: {total_samples}")

# Define split ratios
train_ratio = 0.8
test_ratio = 0.2

# Compute sizes
train_size = int(train_ratio * total_samples)
test_size = total_samples - train_size

# Randomly split the dataset
train_dataset, test_dataset = random_split(eeg_dataset, [train_size, test_size])

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)
print(f"Total samples in the train dataset: {len(train_dataset)}")
print(f"Total samples in the test dataset:  {len(test_dataset)}")


### 
    =========================================================================================

### EVALUATE FUNCTION

```

```
def evaluate(model, test_loader):
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad():  # No gradients needed for evaluation
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)  # Move to GPU

            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)  # Get predicted class
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    return accuracy




from torch import nn



print1Q = False
#print1Q = True


### ========================================================================================

### DEFINE MODEL #7 - 3 layers
### FROM: Deep learning with convolutional neural networks for brain mapping and decoding of movement-related information from the human EEG
### with one less layer and very custumizable
### groups == in_channels and out_channels == K * in_channels


class CNN3layers(nn.Module):
    def __init__(self, input_length=1250, num_classes=40, num_channels=64, F1=16, F2=32, F3=64, F4=128,
                            W1=11, W2=11, W3=11, pool=3, dropout_prob=0.5, do_batch_norm=True, padding="same", k_hidden_layer=1):

        super(CNN3layers, self).__init__()

        #padding_ = "same"
        #padding_ = "valid"

        # Temporal Convolution to focus on the band of interest
```

```
        layers1 = []
        layers1.append(nn.Conv2d(1,  F1, (1, W1), stride=(1, 1), padding=padding, bias=False))
        if(do_batch_norm): layers1.append(nn.BatchNorm2d(F1))
        layers1.append(nn.ELU())
        layers1.append(nn.Conv2d(F1, F2, (num_channels, 1), groups=F1, bias=False))
        if(do_batch_norm): layers1.append(nn.BatchNorm2d(F2))
        layers1.append(nn.ELU())
        layers1.append(nn.MaxPool2d((1, pool), stride=(1, pool)))
        layers1.append(nn.Dropout(p=dropout_prob))
        self.block1 = nn.Sequential(*layers1)


        # Depthwise Convolution for spatial filtering
        layers2 = []
        layers2.append(nn.Conv2d(F2,  F3, (1, W2), stride=(1, 1), groups=F2, padding=padding, bias=False))
        if(do_batch_norm): layers2.append(nn.BatchNorm2d(F3))
        layers2.append(nn.ELU())
        layers2.append(nn.MaxPool2d((1, pool), stride=(1, pool)))
        layers2.append(nn.Dropout(p=dropout_prob))
        self.block2 = nn.Sequential(*layers2)


        # ANOTHER DEPTHWISE
        layers3 = []
        layers3.append(nn.Conv2d(F3,  F4, (1, W3), stride=(1, 1), groups=F3, padding=padding, bias=False))
        if(do_batch_norm): layers3.append(nn.BatchNorm2d(F4))
        layers3.append(nn.ELU())
        layers3.append(nn.MaxPool2d((1, pool), stride=(1, pool)))
        layers3.append(nn.Dropout(p=dropout_prob))
        self.block3 = nn.Sequential(*layers3)


        # LINEAR LAYER
        #k_hidden_layer = 3
        layersFC = [nn.Flatten()]
        if(padding=="same"): layersFC.append(nn.Linear(F4 * (input_length // (pool**3)), num_classes*k_hidden_layer))
        else:   layersFC.append(nn.Linear(int( F4*(((((input_length - W1+1)//pool) - W2+1)//pool  - W3+1)//pool) ) , num_classes*k_hidden_layer))

        if k_hidden_layer > 1:
            layersFC.append(nn.ELU())
            layersFC.append(nn.Linear(num_classes*k_hidden_layer, num_classes))  # Adapt to temporal dimension

```

```
        self.fc = nn.Sequential(*layersFC)


    def forward(self, x):
        if(print1Q): print("input, ", x.shape)
        x = x.unsqueeze(1) # Add channel dimension: [batch, 1, 64, 1500]
        if(print1Q): print("unsqueeze, ", x.shape)
        x = self.block1(x)
        if(print1Q): print(x.shape)
        x = self.block2(x)
        if(print1Q): print(x.shape)
        x = self.block3(x)
        if(print1Q): print(x.shape)
        x = self.fc(x)
        if(print1Q): print(x.shape)
        return x



### ========================================================================================
### DEFINE MODEL #8 - 3 layers
### FROM GONCALO. ONLY CONVOLUTIONAL LAYERS


class CNNOnlyConv(nn.Module):
    def __init__(self, input_length=1250, num_classes=40, num_channels=64, F1=16, F2=32, F3=64, F4=128,
                            W1=11, W2=11, W3=11, pool=3, dropout_prob=0.5, do_batch_norm=True, padding="same", do_final_linear=-1):

        super(CNNOnlyConv, self).__init__()

        # Temporal Convolution to focus on the band of interest
        layers1 = []
        layers1.append(nn.Conv2d(1,  F1, (1, W1), stride=(1, 1), padding=padding, bias=False))
        if(do_batch_norm): layers1.append(nn.BatchNorm2d(F1))
        layers1.append(nn.ELU())
        layers1.append(nn.Conv2d(F1, F2, (num_channels, 1), groups=F1, bias=False))
        if(do_batch_norm): layers1.append(nn.BatchNorm2d(F2))
        layers1.append(nn.ELU())
        layers1.append(nn.MaxPool2d((1, pool), stride=(1, pool)))
        layers1.append(nn.Dropout(p=dropout_prob))
        self.block1 = nn.Sequential(*layers1)

```

```python

        # Depthwise Convolution for spatial filtering
        layers2 = []
        layers2.append(nn.Conv2d(F2,  F3, (1, W2), stride=(1, 1), groups=F2, padding=padding, bias=False))
        if(do_batch_norm): layers2.append(nn.BatchNorm2d(F3))
        layers2.append(nn.ELU())
        layers2.append(nn.MaxPool2d((1, pool), stride=(1, pool)))
        layers2.append(nn.Dropout(p=dropout_prob))
        self.block2 = nn.Sequential(*layers2)


        # ANOTHER DEPTHWISE
        layers3 = []
        layers3.append(nn.Conv2d(F3,  F4, (1, W3), stride=(1, 1), groups=F3, padding=padding, bias=False))
        if(do_batch_norm): layers3.append(nn.BatchNorm2d(F4))
        layers3.append(nn.ELU())
        layers3.append(nn.MaxPool2d((1, pool), stride=(1, pool)))
        layers3.append(nn.Dropout(p=dropout_prob))
        self.block3 = nn.Sequential(*layers3)


        # LINEAR LAYER
        k_hidden_layer = 3
        layersFC = []
        if(padding=="same" and do_final_linear==-1):
            layersFC.append(nn.Conv2d(F4,  num_classes, (1, (input_length // (pool**3))), groups=num_classes, padding=padding, bias=False))
            layersFC.append(nn.Flatten())
        elif (padding=="valid" and do_final_linear==-1):
            layersFC.append(nn.Conv2d(F4, num_classes, (1, int( ((((input_length - W1+1)//pool) - W2+1)//pool  - W3+1)//pool))), groups=num_classes, padding=padding, bias=False))
            layersFC.append(nn.Flatten())
        elif(padding=="same" and do_final_linear > 0):
            layersFC.append(nn.Conv2d(F4,  do_final_linear, (1, (input_length // (pool**3))), groups=F4, padding=padding, bias=False))
            layersFC.append(nn.Flatten())
            layersFC.append(nn.Linear(do_final_linear, num_classes))
        elif (padding=="valid" and do_final_linear > 0):
            layersFC.append(nn.Conv2d(F4, do_final_linear, (1, int( ((((input_length - W1+1)//pool) - W2+1)//pool  - W3+1)//pool) ), groups=F4, padding=padding, bias=False))
            layersFC.append(nn.Flatten())
            layersFC.append(nn.Linear(do_final_linear, num_classes))

        self.fc = nn.Sequential(*layersFC)
```

```python

        #self.nnn = nn.Linear(do_final_linear, num_classes)


    def forward(self, x):
        if(print1Q): print("input, ", x.shape)
        x = x.unsqueeze(1) # Add channel dimension: [batch, 1, 64, 1500]
        if(print1Q): print("unsqueeze, ", x.shape)
        x = self.block1(x)
        if(print1Q): print(x.shape)
        x = self.block2(x)
        if(print1Q): print(x.shape)
        x = self.block3(x)
        if(print1Q): print(x.shape)
        x = self.fc(x)
        #if(print1Q): print(x.shape)
        #x = self.nnn(x)
        if(print1Q): print(x.shape)
        return x



###
    =========================================================================================
### TRAINING LOOP


import torch.optim as optim
from torchsummary import summary

# Device will determine whether to run the training on GPU or CPU.
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Device: {device}")

train_time = time.time()

# Initialize model, loss, and optimizer
model = CNN3layers(num_classes=40, F1=5, F2=10, F3=20, F4=40, W1=124, W2=64, W3=31,
    dropout_prob=0.5, pool=2, padding="same", do_batch_norm=True, k_hidden_layer=1).to
    (device)
model = CNN3layers(num_classes=40, F1=10, F2=20, F3=40, F4=80, W1=121, W2=59, W3=11,
    dropout_prob=0.5, pool=2, padding="same", do_batch_norm=True, k_hidden_layer=1).to
    (device)
#model = CNN3layers(num_classes=40, F1=4, F2=4, F3=8, F4=80, W1=59, W2=19, W3=19,
    dropout_prob=0.5, pool=1, padding="valid", do_batch_norm=True).to(device)

```

```python
#model = CNNOnlyConv(num_classes=40, F1=20, F2=20, F3=20, F4=40, W1=59, W2=19, W3=19, dropout_prob=0.5, pool=1, padding="valid", do_batch_norm=True, do_final_linear=40).to(device)
## GOLD
model = CNNOnlyConv(num_classes=40, F1=10, F2=20, F3=40, F4=80, W1=59, W2=19, W3=19, dropout_prob=0.5, pool=1, padding="valid", do_batch_norm=True, do_final_linear=-1).to(device)
model = CNNOnlyConv(num_classes=40, F1=5, F2=5, F3=10, F4=40, W1=101, W2=19, W3=19, dropout_prob=0.5, pool=1, padding="valid", do_batch_norm=True, do_final_linear=-1).to(device)

#model = CNNOnlyConv(num_classes=40, F1=5, F2=10, F3=20, F4=40, W1=59, W2=19, W3=19, dropout_prob=0.5, pool=2, padding="valid", do_batch_norm=True, do_final_linear=-1).to(device)



summary(model, (64, 1250), batch_size=16)
criterion = nn.CrossEntropyLoss()
#optimizer = optim.AdamW(model.parameters(), lr=1e-3, weight_decay=0.05)
optimizer = optim.AdamW(model.parameters(), lr=1e-3, weight_decay=0.05)

# Training loop
num_epochs = 20 #10 #30
loss_list  = []
accu_list  = []
train_accu_list = []
train_loss_list = []
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    #for inputs, labels in eeg_dataloader:
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)  # Move to GPU
        optimizer.zero_grad()
        if(print1Q): print(inputs.shape)
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    loss_list += [running_loss / len(train_loader)]
    accu_list += [evaluate(model, test_loader)]
    train_accu_list += [evaluate(model, train_loader)]
    print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {running_loss / len(train_loader):.2f} | Training Accuracy: {train_accu_list[-1]:.2f}% | Validation Accuracy: {accu_list[-1]:.3f}%")
```

```
## SAVE MODEL
pytorch_total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
torch.save(model, SOURCEFOLDER + f"model1_3L_{pytorch_total_params}.pt")





### ==========================================================================================
### VISUALIZATION

fig = plt.figure()
plt.plot(accu_list, label="Validation accuracy")
plt.plot(train_accu_list, label="Training accuracy")
plt.legend()
plt.show()
plt.savefig("Train-Valudation accuracies.png")


print(f"Time elapsed: {(time.time() - start_time)/60:.2f}min")
print(f"training time: {(time.time() - train_time)/60:.2f}min")
```

B Appendix B - model reader and weight visualization script

Listing 2: model reader and weight visualization script

```
import torch
from torch import nn

import numpy as np
from matplotlib.transforms import Bbox
from torch.utils.data import Dataset, DataLoader, TensorDataset
from scipy.io import loadmat
import os

torch.cuda.empty_cache()


SOURCEFOLDER = ""




### ========================================================================================

class EEGDatasetAugmented(Dataset):
    def __init__(self, data_dir, n_subject, augment=False):
        """
        Initialize the dataset by loading and preprocessing EEG data.

        Args:
        - data_dir: Path to the directory containing .mat files.
        """
        self.data = []
        self.labels = []
        #self.n_subjects = n_subjects
        self.augment = augment
        self._load_data(data_dir, n_subject)

    def _load_data(self, data_dir, n_subject):
        """
        Load and preprocess EEG data from .mat files.

        Args:
        - data_dir: Directory containing .mat files.
        """
        mat_data = loadmat(os.path.join(data_dir, f"S{n_subject}.mat"))
        eeg_data = mat_data['data']  # Adjust key based on your .mat file
        # eeg_data shape: [64, 1500, 40, 6]
        ## CROP
        eeg_data = eeg_data[:,124:1500-125-1,:,:]
```

```

        num_electrodes, num_timepoints, num_classes, num_blocks = eeg_data.shape

        # Reshape into trials: [num_classes * num_blocks, 64, 1500]
        for block in range(num_blocks):
            for target in range(num_classes):
                trial = eeg_data[:, :, target, block]
                self.data.append(trial)
                self.labels.append(target)


        # Convert lists to numpy arrays for PyTorch compatibility
        self.data = np.array(self.data)  # Shape: [num_trials, 64, 1500]
        self.labels = np.array(self.labels)  # Shape: [num_trials]

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        trial = self.data[idx]

        if self.augment:
            trial = self.apply_augmentation(trial)

        trial = torch.tensor(trial, dtype=torch.float32)  # Shape: [64, 1500]
        label = torch.tensor(self.labels[idx], dtype=torch.long)
        return trial, label



test_dataset = EEGDatasetAugmented("data", 11, augment=False)

test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)



### ========================================================================================
### DEFINE EVALUATE FUNCTIONS

def evaluate(model, test_loader):
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad():  # No gradients needed for evaluation
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)  # Move to GPU
```

```

            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)  # Get predicted class
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    return accuracy





### ========================================================================================
### DEFINE MODEL #3

print1Q = False
#print1Q = True

# Creating a CNN class
class ShallowConvNet(nn.Module):
#  Determine what layers and their order in CNN object
    def __init__(self, num_classes=40, C=64, T=1250, F1=8, D=2, F2=16, dropout_prob=0.5, Wt=64, Ws=16, pool1=4, pool2=8):
        """
        Args:
            num_classes: Number of output classes (default=40).
            C: Number of EEG channels (default=64).
            T: Number of time points (default=1250).
            F1: Number of temporal filters (default=8).
            D: Depth multiplier for depthwise conv (default=2).
            F2: Number of separable spatial filters (default=16).
            dropout_prob: Dropout probability (default=0.5).
            Wt: width of time kernel
            Ws: width of space kernel
            pool1: width of block 1 pool kernel
            pool2: width of block 2 pool kernel
        """
        super(ShallowConvNet, self).__init__()

        # Block 1: Temporal Convolution
        self.block1 = nn.Sequential(
            nn.Conv2d(1, F1, kernel_size=(1, Wt), padding='same', bias=False),  # Temporal Conv
            nn.BatchNorm2d(F1),
            nn.Conv2d(F1, F1 * D, kernel_size=(C, 1), bias=False, groups=D),
            nn.BatchNorm2d(F1 * D),
```

```
            nn.ELU(),
            nn.AvgPool2d(kernel_size=(1, pool1)),
            nn.Dropout(p=dropout_prob)
        )

        # Block 2: Separable Convolution
        self.block2 = nn.Sequential(
            nn.Conv2d(F1 * D, F2, kernel_size=(1, Ws), padding='same', bias=False), # Depthwise Separable Conv
            nn.BatchNorm2d(F2),
            nn.ELU(),
            nn.AvgPool2d(kernel_size=(1, pool2)),
            nn.Dropout(p=dropout_prob)
        )

        # Fully Connected Layer
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(F2 * (T // (pool1*pool2)), num_classes),  # Adjust based on the pooling
            nn.Softmax(dim=1)
        )


    # Progresses data across layers
    def forward(self, x):
        if(print1Q): print("input, ", x.shape)
        x = x.unsqueeze(1) # Add channel dimension: [batch, 1, 64, 1500]
        if(print1Q): print("input2, ", x.shape)
        x = self.block1(x)
        if(print1Q): print(x.shape)
        x = self.block2(x)
        if(print1Q): print(x.shape)
        x = self.fc(x)
        if(print1Q): print(x.shape)
        return x



### ========================================================================================
### DEFINE MODEL #4


print1Q = False
#print1Q = True

```

```
class Simple3conv(nn.Module):
    def __init__(self, input_length=1250, num_classes=40, num_channels=64, F1=8,
        groups=8, F2=16, F3=32, dropout_prob=0.5, Wt=125, Ws=16, pool1=4):
        """
        Args:
            num_classes: Number of output classes (default=40).
            C: Number of EEG channels (default=64).
            T: Number of time points (default=1250).
            F1: Number of temporal filters (default=8).
            D: Depth multiplier for depthwise conv (default=2).
            F2: Number of separable spatial filters (default=16).
            dropout_prob: Dropout probability (default=0.5).
            Wt: width of time kernel
            Ws: width of space kernel
            pool1: width of block 1 pool kernel
            pool2: width of block 2 pool kernel
        """
        super(Simple3conv, self).__init__()

        # Temporal Convolution to focus on the band of interest
        self.temporal_conv = nn.Sequential(
            nn.Conv2d(1, F1, (1, Wt), stride=(1, 2), padding=(0, (Wt-1)//2), bias=
                False),  # Filter ~2-20Hz
            nn.BatchNorm2d(F1),
            nn.ELU(),
            nn.Dropout(p=dropout_prob)
        )

        # Depthwise Convolution for spatial filtering
        self.spatial_conv = nn.Sequential(
            nn.Conv2d(F1, F2, (num_channels, 1), groups=groups, bias=False),  #
                Depthwise
            nn.BatchNorm2d(F2),
            nn.ELU(),
            nn.Dropout(p=dropout_prob)
        )

        # Separable Convolution for extracting temporal features
        self.temporal_separable = nn.Sequential(
            nn.Conv2d(F2, F3, (1, F2), stride=(1, 2), padding=(0, F2//2), bias=False)
                ,  # Temporal separable
            nn.BatchNorm2d(F3),
            nn.ELU(),
            nn.AvgPool2d((1, 4)),  # Reduce temporal dimension,
            nn.Dropout(p=dropout_prob)
        )

        # Classification Head
```

```
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(F3 * (input_length // (pool1*2*2)), num_classes*2),  # Adapt to
                 temporal dimension
            nn.ReLU(),
            nn.Linear(num_classes*2, num_classes),  # Adapt to temporal dimension
            nn.Softmax(dim=1)
        )


    def forward(self, x):
        if(print1Q): print("input, ", x.shape)
        x = x.unsqueeze(1) # Add channel dimension: [batch, 1, 64, 1500]
        if(print1Q): print("input2, ", x.shape)
        x = self.temporal_conv(x)
        if(print1Q): print(x.shape)
        x = self.spatial_conv(x)
        if(print1Q): print(x.shape)
        x = self.temporal_separable(x)
        if(print1Q): print(x.shape)
        x = self.fc(x)
        if(print1Q): print(x.shape)
        return x


###
    ==========================================================================================

### DEFINE MODEL #7

class CNN3layers(nn.Module):
    def __init__(self, input_length=1250, num_classes=40, num_channels=64, F1=16, F2
        =32, F3=64, F4=128,
                                W1=11, W2=11, W3=11, pool=3, dropout_prob=0.5,
                                    do_batch_norm=True, padding="same"):

        super(CNN3layers, self).__init__()

        #padding_ = "same"
        #padding_ = "valid"

        # Temporal Convolution to focus on the band of interest
        layers1 = []
        layers1.append(nn.Conv2d(1,  F1, (1, W1), stride=(1, 1), padding=padding,
            bias=False))
        if(do_batch_norm): layers1.append(nn.BatchNorm2d(F1))
        layers1.append(nn.ELU())
        layers1.append(nn.Conv2d(F1, F2, (num_channels, 1), groups=F1, bias=False))
```

```
        if(do_batch_norm): layers1.append(nn.BatchNorm2d(F2))
        layers1.append(nn.ELU())
        layers1.append(nn.MaxPool2d((1, pool), stride=(1, pool)))
        layers1.append(nn.Dropout(p=dropout_prob))
        self.block1 = nn.Sequential(*layers1)


        # Depthwise Convolution for spatial filtering
        layers2 = []
        layers2.append(nn.Conv2d(F2,  F3, (1, W2), stride=(1, 1), groups=F2, padding=
            padding, bias=False))
        if(do_batch_norm): layers2.append(nn.BatchNorm2d(F3))
        layers2.append(nn.ELU())
        layers2.append(nn.MaxPool2d((1, pool), stride=(1, pool)))
        layers2.append(nn.Dropout(p=dropout_prob))
        self.block2 = nn.Sequential(*layers2)


        # ANOTHER DEPTHWISE
        layers3 = []
        layers3.append(nn.Conv2d(F3,  F4, (1, W3), stride=(1, 1), groups=F3, padding=
            padding, bias=False))
        if(do_batch_norm): layers3.append(nn.BatchNorm2d(F4))
        layers3.append(nn.ELU())
        layers3.append(nn.MaxPool2d((1, pool), stride=(1, pool)))
        layers3.append(nn.Dropout(p=dropout_prob))
        self.block3 = nn.Sequential(*layers3)


        # LINEAR LAYER
        k_hidden_layer = 3
        layersFC = [nn.Flatten()]
        if(padding=="same"): layersFC.append(nn.Linear(F4 * (input_length // (pool
            **3)), num_classes*k_hidden_layer))
        else:   layersFC.append(nn.Linear(F4 * (input_length - (W1+1) - pool*(W2+1) -
             pool**2*(W3+1)) // (pool**3), num_classes*k_hidden_layer))
        layersFC.append(nn.ELU())
        layersFC.append(nn.Linear(num_classes*k_hidden_layer, num_classes))  # Adapt
            to temporal dimension
        #layersFC.append(nn.Softmax(dim=1))

        self.fc = nn.Sequential(*layersFC)


    def forward(self, x):
        if(print1Q): print("input, ", x.shape)
        x = x.unsqueeze(1) # Add channel dimension: [batch, 1, 64, 1500]
        if(print1Q): print("unsqueeze, ", x.shape)
```

```
        x = self.block1(x)
        if(print1Q): print(x.shape)
        x = self.block2(x)
        if(print1Q): print(x.shape)
        x = self.block3(x)
        if(print1Q): print(x.shape)
        x = self.fc(x)
        if(print1Q): print(x.shape)
        return x





###
    ===========================================================================================

### DEFINE MODEL #8

class CNNOnlyConv(nn.Module):
    def __init__(self, input_length=1250, num_classes=40, num_channels=64, F1=16, F2
        =32, F3=64, F4=128,
                            W1=11, W2=11, W3=11, pool=3, dropout_prob=0.5,
                                do_batch_norm=True, padding="same"):

        super(CNNOnlyConv, self).__init__()

        # Temporal Convolution to focus on the band of interest
        layers1 = []
        layers1.append(nn.Conv2d(1,  F1, (1, W1), stride=(1, 1), padding=padding,
            bias=False))
        if(do_batch_norm): layers1.append(nn.BatchNorm2d(F1))
        layers1.append(nn.ELU())
        layers1.append(nn.Conv2d(F1, F2, (num_channels, 1), groups=F1, bias=False))
        if(do_batch_norm): layers1.append(nn.BatchNorm2d(F2))
        layers1.append(nn.ELU())
        layers1.append(nn.MaxPool2d((1, pool), stride=(1, pool)))
        layers1.append(nn.Dropout(p=dropout_prob))
        self.block1 = nn.Sequential(*layers1)


        # Depthwise Convolution for spatial filtering
        layers2 = []
        layers2.append(nn.Conv2d(F2,  F3, (1, W2), stride=(1, 1), groups=F2, padding=
            padding, bias=False))
        if(do_batch_norm): layers2.append(nn.BatchNorm2d(F3))
        layers2.append(nn.ELU())
        layers2.append(nn.MaxPool2d((1, pool), stride=(1, pool)))
```

```python
        layers2.append(nn.Dropout(p=dropout_prob))
        self.block2 = nn.Sequential(*layers2)


        # ANOTHER DEPTHWISE
        layers3 = []
        layers3.append(nn.Conv2d(F3,  F4, (1, W3), stride=(1, 1), groups=F3, padding=padding, bias=False))
        if(do_batch_norm): layers3.append(nn.BatchNorm2d(F4))
        layers3.append(nn.ELU())
        layers3.append(nn.MaxPool2d((1, pool), stride=(1, pool)))
        layers3.append(nn.Dropout(p=dropout_prob))
        self.block3 = nn.Sequential(*layers3)


        # LINEAR LAYER
        k_hidden_layer = 3
        layersFC = []
        if(padding=="same"): layersFC.append(nn.Conv2d(F4,  num_classes, (1, (input_length // (pool**3))), groups=num_classes, padding=padding, bias=False))
        else:   layersFC.append(nn.Conv2d(F4, num_classes, (1, int( (((((input_length - W1+1)//pool) - W2+1)//pool  - W3+1)//pool))), groups=num_classes, padding=padding, bias=False))
        #layersFC.append(nn.ELU())
        #layersFC.append(nn.Linear(num_classes*k_hidden_layer, num_classes))  # Adapt to temporal dimension
        layersFC.append(nn.Flatten())
        #layersFC.append(nn.Softmax(dim=1))

        self.fc = nn.Sequential(*layersFC)


    def forward(self, x):
        if(print1Q): print("input, ", x.shape)
        x = x.unsqueeze(1) # Add channel dimension: [batch, 1, 64, 1500]
        if(print1Q): print("unsqueeze, ", x.shape)
        x = self.block1(x)
        if(print1Q): print(x.shape)
        x = self.block2(x)
        if(print1Q): print(x.shape)
        x = self.block3(x)
        if(print1Q): print(x.shape)
        x = self.fc(x)
        if(print1Q): print(x.shape)
        return x
```

```
### ==================================================================================

### LOAD MODULE

from matplotlib import pyplot as plt

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
#model = Simple3conv(num_classes=40, F1=8, F2=16, F3=32, dropout_prob=0.5).to(device)
     # 40
#model = CNN3layers(num_classes=40, F1=25, F2=50, F3=100, F4=200, W1=121, W2=63, W3
    =25, dropout_prob=0.3, pool=3, padding="valid", do_batch_norm=True).to(device)
model = CNNOnlyConv(num_classes=40, F1=25, F2=50, F3=100, F4=200, W1=121, W2=63, W3
    =25, dropout_prob=0.3, pool=3, padding="valid", do_batch_norm=True).to(device)
model = torch.load(SOURCEFOLDER + "model1_3L_96930.pt", weights_only=False)
model = torch.load(SOURCEFOLDER + "model1_3L_505270.pt", weights_only=False)
#print(model)

#for p in model.parameters():
    #print(p.name, " | ", p.shape)


### EVALUATE MODEL
print("Subject 11 classification:", evaluate(model, test_loader))


weights_mean = model.block1[0].weight.cpu().detach().numpy()[:,0,0,:].mean(axis=0).
    reshape(1,-1)
print(weights_mean.shape[1])
fig = plt.figure(figsize=(weights_mean.shape[1] * 4 / 100+4, 2+1), dpi=100)
plt.imshow(np.abs(weights_mean), interpolation='nearest', cmap='gray', aspect='auto')
plt.show()
plt.colorbar(fraction=0.046, pad=0.04) #cax=cax
plt.savefig("Weights1_mean.png")


weights_mean2 = np.abs(model.block1[3].weight.cpu().detach().numpy()[:,0,:,0]).mean(
    axis=0).reshape(1,-1)
print(weights_mean2.shape[1])
fig = plt.figure(figsize=(weights_mean2.shape[1] * 4 / 100+4, 2+1), dpi=100)
plt.imshow((np.abs(weights_mean2)), interpolation='nearest', cmap='viridis', aspect='
    auto') #viridis Wistia
#plt.imshow(np.log(np.abs(weights_mean2)), interpolation='nearest', cmap='viridis',
    aspect='auto') #viridis Wistia
plt.show()
```

```
plt.colorbar(fraction=0.046, pad=0.04) #cax=cax
plt.xlabel("Channel number")
plt.ylabel("Log of mean absolute weights")
plt.savefig("Weights2_mean.png", bbox_inches=Bbox([[0,-0.5],fig.get_size_inches()]))


fig = plt.figure()
plt.imshow(model.block1[0].weight.cpu().detach().numpy()[:,0,0,:], interpolation='
    nearest')
plt.show()
plt.colorbar(fraction=0.046, pad=0.04, location="top")
plt.xlabel("Kernel number")
plt.ylabel("Index in kernel")
plt.savefig("Weights1.png")

fig = plt.figure()
plt.imshow(model.block1[3].weight.cpu().detach().numpy()[:,0,:,0], interpolation='
    nearest')
plt.show()
plt.colorbar(fraction=0.046, pad=0.04, location="top")
plt.xlabel("Channel number")
plt.ylabel("Index in kernel")
plt.savefig("Weights2.png", bbox_inches=Bbox([[0,-0.5],fig.get_size_inches()]))

fig = plt.figure()
plt.imshow(model.block2[0].weight.cpu().detach().numpy()[:,0,0,:], interpolation='
    nearest')
plt.show()
plt.colorbar(fraction=0.046, pad=0.04)
plt.savefig("Weights3.png")

fig = plt.figure()
plt.imshow(model.block3[0].weight.cpu().detach().numpy()[:,0,0,:], interpolation='
    nearest')
plt.show()
plt.colorbar(fraction=0.046, pad=0.04)
plt.savefig("Weights4.png")



for i, w in enumerate(weights_mean[0]):
    if np.abs(w) > 0.03:
        print(f"Electrode {i+1}: {w}")

print("\nWeights")
for i, w in enumerate(weights_mean2[0]):
    #if np.abs(w) > 0.02:
    if np.abs(w) > 0.07:
```

```
        print(f"Electrode {i+1}: {w}")

""" #FOR MODEL1
plt.imshow(model.temporal_conv[0].weight.cpu().detach().numpy()[:,0,0,:],
    interpolation='nearest')
plt.show()
plt.savefig("Weights1.png")


plt.imshow(model.spatial_conv[0].weight.cpu().detach().numpy()[:,0,:,0],
    interpolation='nearest')
plt.show()
plt.savefig("Weights2.png")

plt.imshow(model.fc[3].weight.cpu().detach().numpy(), interpolation='nearest')
plt.show()
plt.savefig("Weights3.png")
"""
```