DANMARKS TEKNISKE UNIVERSITET

EEG signals discrimination based on deep learning

SPECIAL COURSE

January 28, 2022

Autor: Alvaro Carrera Cardeli - s172254

Instructor: Sadasivan Puthusserypady
Supervisor: Cihan Uyanik

Abstract

Brain computer interface (BCI) provides a direct communication between the brain and external devices by decoding brain signals into commands. The main goal of this study is to implement a deep learning classification algorithm for the task two-class motor imagery. The deep learning algorithm is based on learning from Convolutional Neural Networks (CNN) due to the spatial and temporal resolution of the trials. Some pre-processing is required before performing the modeling of the classifier. Since the model is likely to over-fit due to the lack of trials and high dimensions of each of the trails, some control-complexity parameters are analyzed before training the final model. Four models based on CNN with different architectures are compared: 1D temporal convolutions, 2D temporal and spatial convolutions, shallow CNN with single temporal and spatial convolution, and deep CNN with temporal-spatial convolution combined with temporal convolutions. The best result is obtained with the 1D temporal convolution architecture reaching an accuracy of 75.23%.

Contents

1 Introduction

Brain Computer Interface (BCI) is a computer-based system that provides direct communication between the brain and external devices by acquiring and decoding the brain signals into commands that are relayed to an output device to carry out an action without any muscular activation. Thus, signals generated in the brain do not depend on the brain's normal output pathways of peripheral nerves and muscles to evoke an action. The usual process for a BCI system begins with the acquisition of the user's brain activity (i.e. brain signals). Then, the goal is to process the electroencephalographic (EEG) signal to detect the user's intention. Finally, the signal is applied in any specific framework to execute an action.

In principle, any type of brain signal could be used to control a BCI system. Also, there are several BCI systems depending on the different acquisition modalities that can be either invasive or non-invasive. EEG-based BCIs record brain signals from the scalp and are the most common technique due to their portability, convenience, safety, and low cost. On the other hand, this systems have to overcome problems such as attenuation generated by the skull or overlapping of signals generated in different areas of the brain. Thus, it is important a pre-processing step in order to clean the EEG signals before training the detection algorithm. Usually, the pre-processing in this type of BCI systems deals with filtering the signal to the frequencies of interest (i.e. removing all the frequencies of the signals that cannot be generated with the action that the system is trying to detect). Also, some other techniques are used for a more exhaustive cleaning of the EEG signal, such as Common Average Reference (CAR) removal. This can be useful to remove brain signals that cannot be controlled by the user, such as eye blinking, that are present in all the channels at the same time.

When it comes detecting the user's intention, machine learning and deep learning techniques play an important role in EEG-based research and application areas since they allow extracting information from EEG recordings. Some machine learning algorithms use feature generation and transformation, pattern recognition or linear modelling in order to optimize the information and classify. As will be shown in the following section, these techniques perform quite good in EEG detection. However, due to the nature of the EEG signals, that have temporal and spatial information, it may be interesting to dig in areas such as non-linear transformation, temporal/spatial analysis and feature compression. Deep learning performs good in this type of scenarios.

The main goal in this report is to implement a deep learning algorithm based on Convolutional Neural Network (CNN) that performs with accuracy in the classification of a two-class motor imagery data set. The large variety of tuning parameters that compose a CNN will difficult the task, since the network can vary a lot depending on the data that is trying to model.

A state of the art related to the topic will be presented in Section 2. Then, a brief introduction about the human physiology regarding the pathology this BCI is trying to overcome will be introduced in section 3. Section 4 will describe the data set and the CNN algorithms implemented, where the CNN will vary in depth and dimensions in convolution ways. The final sections will focus in presenting the results, discussing about them and providing some conclusions, section 5, 6, and 7, respectively. Also, in section 7 it will be purposed some future lines.

2 State of art

In the literature there is a lot of research using some machine learning methods for two-class motor imagery classification that perform quite well. Also, lately, there is a deeper research about deep learning performance for the same task. In this section, some results obtained with machine learning algorithms will be presented to make an idea of the accuracy levels that this methods are achieving (the final goal is to improve this ratios). Then, some deep learning algorithms will be presented, mainly based on CNN, to help to define a starting point for this report.

Some recent researches regarding machine learning performance can be found in *Motor-Imagery EEG Signals Classification using SVM, MLP and LDA Classifiers* , where the authors compare the performance of support vector machine (SVM), milt-layer perception (MLP) and linear discriminant analysis (LDA). All the methods are preforming the extraction of eight features before classification, and the results show that SMV perform the best with the given feature vector, reaching and accuracy of 98.8% [1]. However, this sort of models are more specific for the data set used, since there are new features generated that adapt better to the data set used, thus it would require feature extraction for different classifications.

On the other hand, deep learning based algorithms just need to define an architecture of the network, and the features will be learned by the model. In [2] it is compared the performance of CNN and Long Short-Term Memory (LSTM), where LSTM outperforms with 74% of accuracy. On the other hand, in [3] combine Deep Convolutional Neural Network (DCNN) to extract the spatial and frequency features followed by LSTM to extract temporal features, obtaining an average accuracy of 70.64%. In the same report, the results are compared with SVM, and the deep learning algorithm outperforms by 5%.

Regarding the data set that will be used in this report, in [4] a non-linear method named Random Forests (RF) is used to classify the trials. This method was used because certain non-linear methods claim to generalize well when only limited amount of data is available, like in the case of this data set. By using common spatial patterns (CSP) it was achieved performances above 70% [4].

Lastly, in [5] there is a deep discussion about CNN performance for different architectures, such as shallow networks or combined networks (temporal and spatial-temporal convolutions). Also, the trials are tested with different configurations looking for the best training performance. [5] will be taken as a reference in some models and results within this report.

3 Human physiology

In this section, it will be done a short introduction about human physiology, so that the goal of this report will be better understood. The problems of interest for this report are the physiology of the brain and the neural system. To understand it better, the target patients for this system are the ones that have some injuries in the spinal cord that makes communication between the brain and peripheral nerves impossible for the execution of actions. Thus, the problem that the system that is being developed aims to overcome is the connection between the brain of a patient and final nerves where some specific actions are executed.

It is important to notice that in these patients the brain is perfectly working. Thus, the action potentials to execute a specific action are generated, although the action never gets to be executed because there is a problem in the connection with the peripheral nerves.

Once discussed about the problem of interest, the physiology that is important to take into consideration it is the way the brain is structured, which will allow to know where the system should focus to recover motor signals form the brain. Also, a short introduction about the neural system will be done, since it is the behaviour that the device will try to mimic.

3.1 How the brain is structured

The brain can be divided into three regions: hind-brain, mid-brain and fore-brain as shown in figure 1. In the hind-brain is where it is usually allocated the connection problem (i.e. the spinal cord injury). However, where the information of interest (i.e. the motor actions) is generated is in the fore-brain. Therefore, this is the part of the brain that will be discussed next.

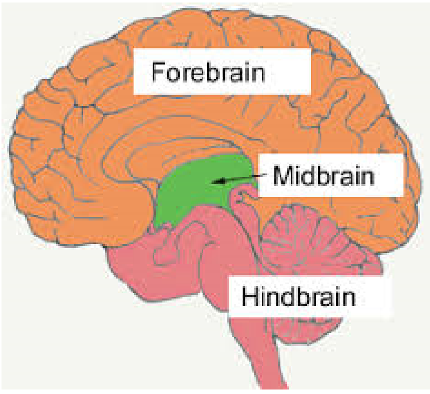

Figure 1: Brain distribution: fore-brain, mid-brain and hind-brain [6].

The fore-brain is also divided in several parts: diencephalon, cerebral hemispheres, basal ganglia, limbic system, frontal lobe, parietal lobe, temporal lobe and occipital lobe. Only the last four will be discussed in this section:

- **Frontal lobe:** extends from the frontal pole to the central and lateral sulci. This region is involved in planning complex learned movement patterns.
- **Parietal lobe:** lies behind the central sulcus and above the lateral sulcus. This region is necessary for somesthetic perception, especially concerning perception of *where* the stimulus is in the space and in relation to body parts.
- **Temporal lobe:** lies below the lateral sulcus. This area is important in discrimination of sounds entering opposite ears.
- **Occipital lobe:** lies posterior to the temporal lobe and parietal lobe. This area is closely connected with the primary visual cortex and thalamus. Integrity of the association cortex is required for visual experience, including experiences of color, motion, depth perception, pattern form and location in space.

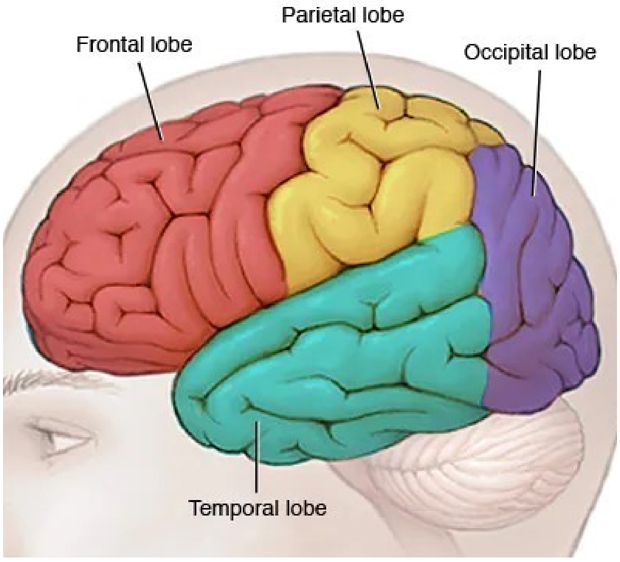

Figure 2: Distribution of brain lobes [7].

Figure 2 shows where each of the lobes are allocated in the brain. Since the problem the device is trying to solve deals with motor actions, the region of interest will be the parietal lobe, where the motor cortex is allocated and all the movement information is processed. Thus, in the following sections it will be seen that the electrodes used to capture the signals will be allocated along the parietal lobe.

3.2 Basics knowledge about Neural system

Neurons are the funcionating cells of the nervous system. Afferent (sensory) neurons transmit information to the Central Nervous System (CNS), whereas efferent (motor) neurons carry information away form the CNS. The system developed will try to mimic somehow the behaviour of the efferent neurons.

Neurons are characterized with the ability to communicate with other neurons through electrical impulses (action potentials). Neuron transfer information from one location to another via the frequency and pattern of action potentials. This is a simple idea of how the information generated in the brain is transmitted to the target nerves in the periphery.

Figure 3 shows a simple scheme of how the neural system is structured. As stated previously, in the spinal cord is where the connection between nerves is lost, thus the information cannot go further down in the neural system. The device aims to bypass this connection by taking the information from the origin (i.e. the brain), and carry it with an external hardware to the destiny (i.e. extremities where the actions should be executed). The goal of this report is to discuss about the software part of the device, thus the way the signals are processed in order to know which actions should be executed. The following sections will discuss about this matter.

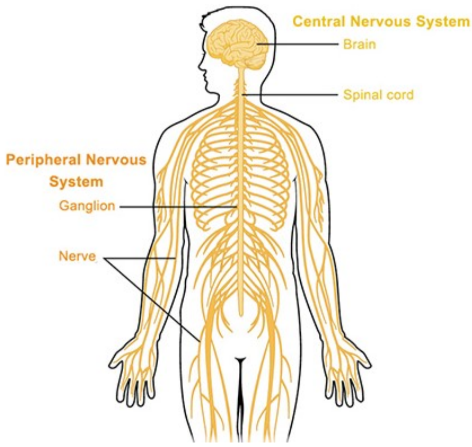

Figure 3: Nervous system [8].

4 Methods and materials

In this section the methods and data used in this project are introduced. Firstly, a short introduction and description of the data set will be done. Then, it will be explained all the pre-processing steps followed in order to prepare the trial for the training. Lastly, the model used with all the hyper-parameters considered will be introduced. All the coding is developed in Python.

4.1 Data set

In this report, it is used a two-class motor imagery data set provided by the organization *BCNI Horizon 2020* [9]. The data set was generated with 14 participants doing a single session each of them. The consisted of eight runs, divided into training (five of them) and evaluation (three of them). Each of the runs is composed of 20 trials, thus each participant would have recorded 100 trials for train and 60 trials for evaluation [4]. This leads to a total of 1400 train trials and 840 evaluation trials in the data set.

The paradigm was based on the cue-guided *GrazBCI training paradigm*. Thus, the trials where generated as follows: participants had the task of performing, during 5 seconds, sustained kinaesthetic motor imagery (MI) of the right hand and of the feet each instructed by the cue. 15 EEG signals were measured with a bio-signal amplifier and active Ag/AgCl electrodes at a sampling rate of 512 Hz. The electrodes placement was around three center electrodes at positions C3, Cz, and C4 (that were also part of the recording). There would be four additional electrodes around each center electrode with a distance of 2.5 cm, thus 15 electrodes in total. The reference electrode was mounted on the left mastoid and the ground electrode on the right mastoid [4]. This distribution should be enough to control the motor cortex signals ans can be seen in figure 4, where the center electrodes would be allocated in the line along the blue area.

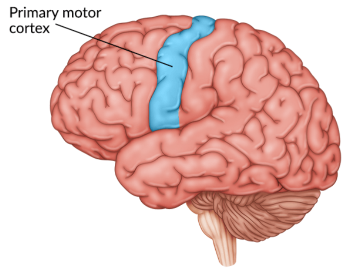

Figure 4: Primary motor cortex location in the brain [10].

As stated in before, there is the same number of trials for both classes (i.e. right hand movement and feet movements), thus the data set is balanced. In figure 5, it can be seen a representation of a trial for each class. On the left subplot, a right hand movement trial is shown while on the right subplot, a feet movement is shown.

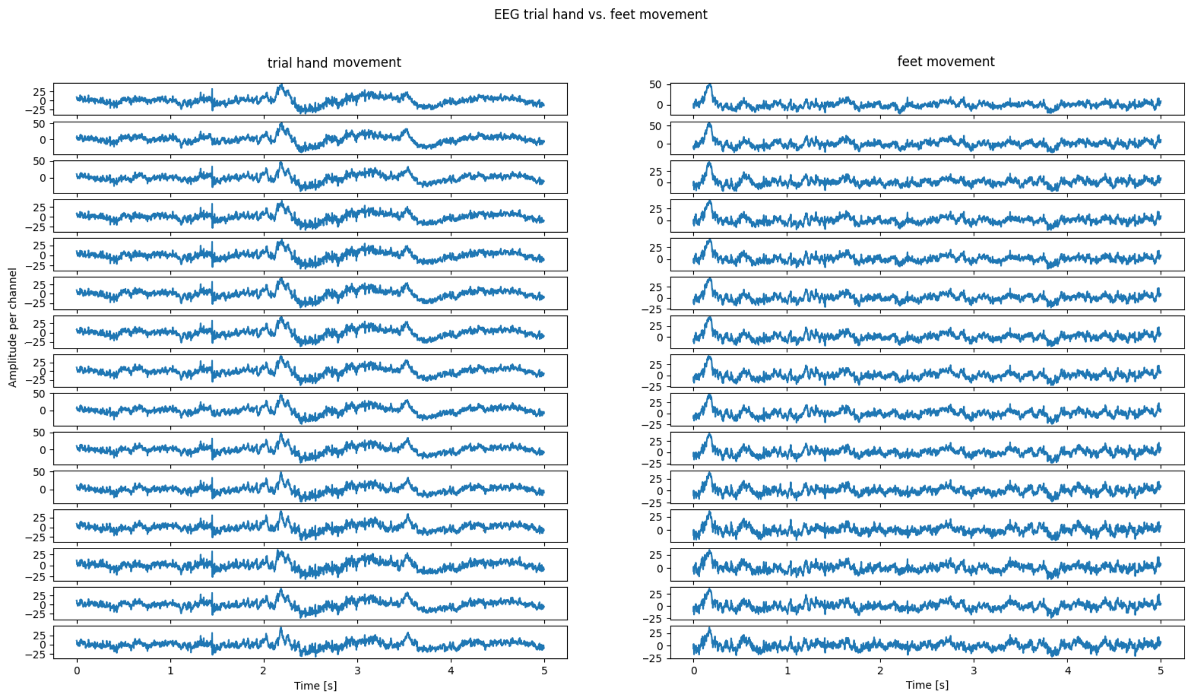

Figure 5: Two class problem. (left) A trial from class right hand movement. (right) A trial from class feet movement.

Lastly, regarding the participants, they were aged between 20 and 30 years old. Also, eight of them were naive to the task, thus some randomness would be introduced in the data set generated. None of the participants had known medical or neurological diseases [4].

4.2 Signal pre-processing

To prepare the trials for training the model it is necessary to do some pre-processing beforehand so that noise is removed from the information of interest. In order to help the classification model and make easier its task, some normalization will be applied to the trials too. Figure 6 shows all the steps followed in order to prepare the signals for the classification algorithm.

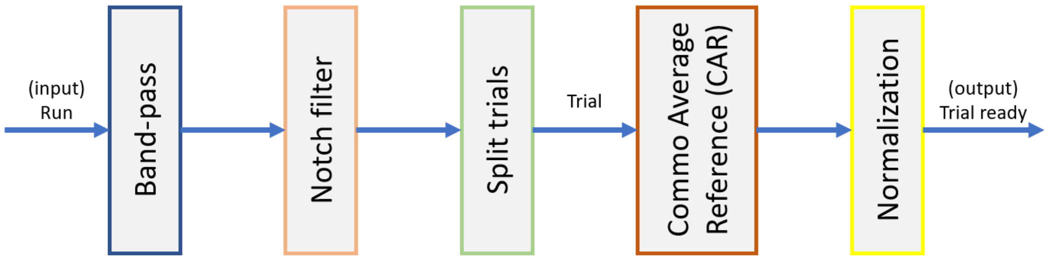

Figure 6: Pre-processing setup.

Regarding figure 6, each of the blocks will be introduced shortly, while doing a comparative between one input and the corresponding output to the specific block:

- **Band-pass:** as stated in the previous section the sampling frequency is of 512 Hz, thus there is up to 256 Hz frequency components in the recorded signals (*Nyquist theorem*). The frequency range of the signals that are about to be classified is from 4 to 40 Hz. Thus, provided signals are passed through a band pass filter with low frequency cut of 4 Hz and high frequency cut of 40 Hz, in order to filter out the frequencies that are not of interest (i.e.: noise in this system).

- **Notch filter:** although frequencies above 40 Hz are filtered out with the band-pass filter, there is a strong interference generated by the measuring devices at 50 Hz. This frequency component my not be attenuated enough after the band-pass filter, thus a notch filter at 50 Hz is also applied to the recorded signals. After applying both filters, the output is shown in figure 7.

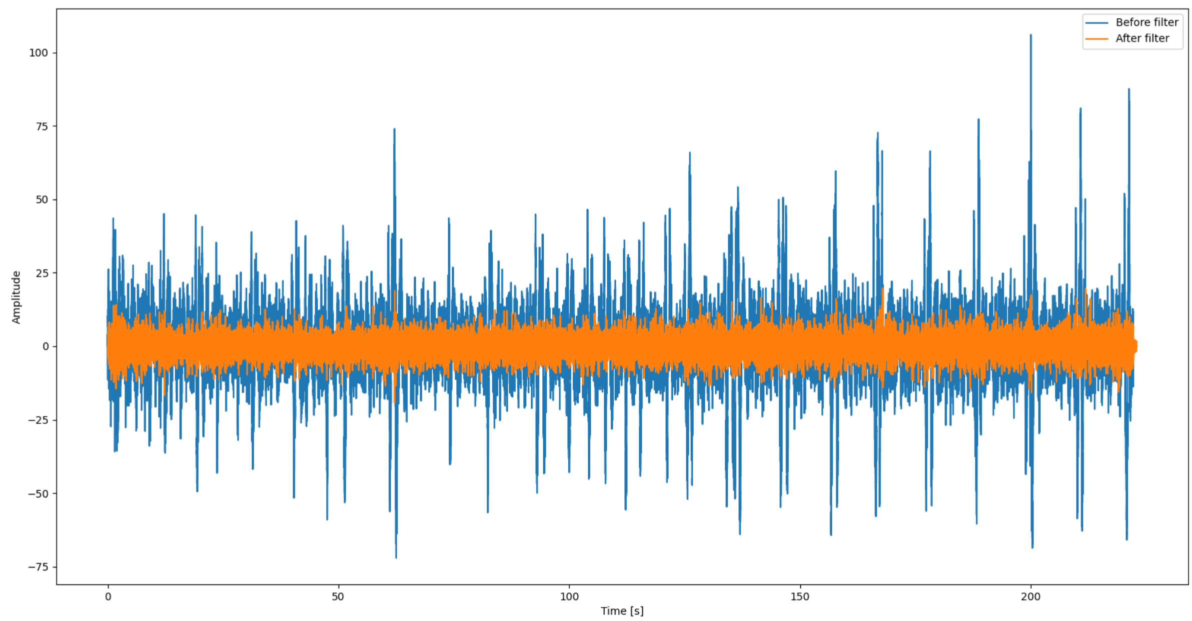

Figure 7: One random run before (blue) and after (red) band-pass and notch filtering.

- **Split trials:** after filtering the signals recorded in each of the runs, all the trials within the runs will be obtained. This is due to the fact that the participants are required to do each of the tasks during 5 consecutive seconds, while it is recorded all the run. Since, each of the tasks is performed during 5 seconds and the sampling frequency is 512 Hz, the trials will be composed of 15 channels with 2560 points each (see figure 5).

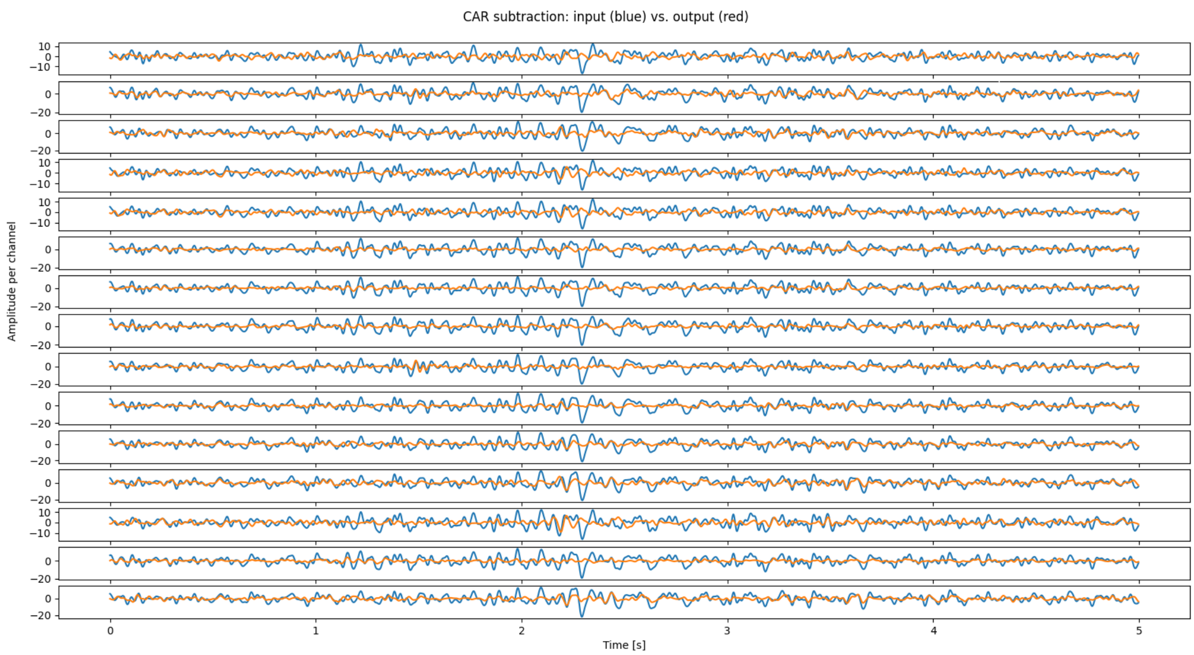

Figure 8: Random trial before (blue) and after (red) CAR subtraction.

- **Common Average Reference (CAR):** CAR is commonly used in EEG, where it is necessary to identify small signal sources in very noisy recordings [11]. The technique consist on taking an average of all the recordings on every electrode site (channel) and using it as a reference. Thus, only signal/noise that is common to all channels (i.e.: it is correlated) remains on the CAR. Then, by subtracting the CAR to all the channels, only the uncorrelated information will remain in the trials. Figure 8 shows the result of subtracting the CAR in one random trial.

- **Normalization:** this is done for training reasons so that it speeds up the convergence of the model. The trials are normalized per channel by subtracting the mean of the channel and dividing by the standard deviation of the channel. Figure 9 shows the outputs of a random trial after normalizing.

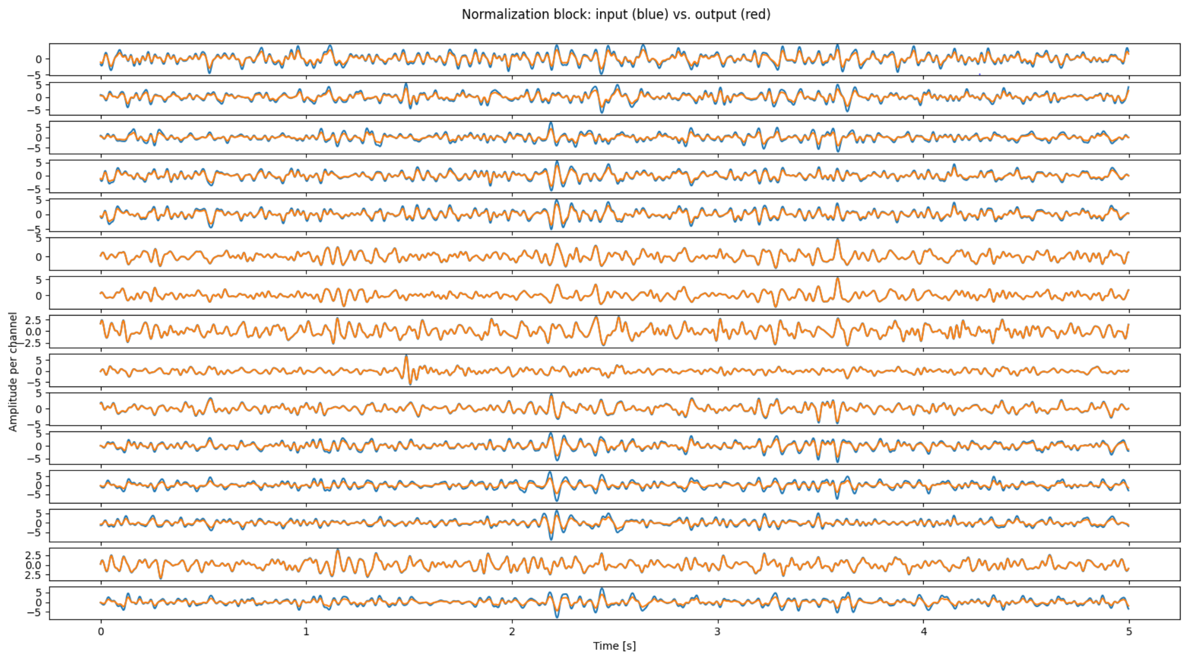

Figure 9: Random trial before (blue) and after (red) normalization.

4.3 Classification algorithm

The two-class motor imagery task that this report is facing has been solved successfully with a bunch of different machine learning algorithms as was introduced in the previous sections. However, given the nature of data for the classification to be done, it has been considered to try to explore algorithms based on deep learning to see if they can give as good results as those of less sophisticated machine learning algorithms.

Specifically, the deep learning algorithms explored will be based on Convolutinoal

Neural Networks (CNN). The reason, as introduced before, is the nature of the data where is contained the information. Each of the trials, that will be feeding the CNN have 15 channels and are measured during 5 seconds. Thus, in each of the trials there is temporal and spatial information, that can be combined in search for new features.

In this section, a brief introduction of the CNN will be done, followed by the description of the CNN architectures that will be modeled in this report.

4.3.1 Convolutional Neural Networks (CNN)

A Convolutional Neural Network is a deep learning algorithm that is able to differentiate some inputs after training some learnable weights and biases. The most common inputs are the ones with high dimensions, such us images, since this networks are good at identifying and highlighting spatial and temporal information. Thus, the role of this networks is to reduce the dimensions of the inputs by keeping just the most important information.

In the case of the trials, as it was stated previously, due to the high dimension per trial (i.e. 15 channels x 2560 time points), and the temporal (per channel) and spatial (between channels allocated spatially close to each other) information, this networks could get to classify properly the trials.

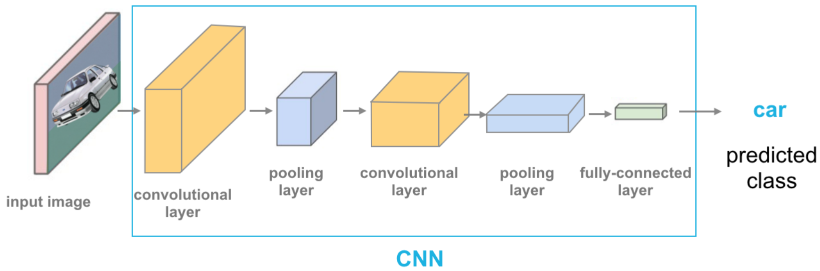

Figure 10: Layout of a simple CNN obtained from [12]

Figure 10 shows the basic structure of the CNN highlighting the main blocks that compose it:

- **Convolutional layer:** it is the core block of the CNN. This layer consist of a set of learnable filters that are temporally and spatially smaller than the trials, but extend through the full depth of the trials volume.
- **Batch normalization:** this block normalizes the batch within the CNN to help the model to learn faster and easier.

Block	Parameter	Values/Description
Convolutional layer	Kernel size	Convolution window size: 1-D (temporal), 2-D (temporal and spatial)
	Dropout	With/Without dropout to control the over-fitting
	Output channel	Number of output channels form the convolutional layer
Activation function	Type	Relu, Elu
Pooling layer	Type	Max, Mean
	Kernel size	High dimension reduction: [1,4], normal dimension reduction: [1,3], [1,2]
Training parameters	Train set size	Number of trials to train, between 600 and 1400. Important for computational cost and time wasting
	Batch size	Small: 4, big: 128
	Epochs	Minimum considered: 150 epochs
Optimizer	Type	Adam, SGD
	Learning rate	Value, and with or without learning rate decay
	Weight decay	To control over-fitting between 1e-4 and 1e0
Criterion	Type	Cross entropy loss

Table 1: Important parameters that affect the performance of the CNN.

- **Activation function:** this block is at the output of the convolutional layers and it is the one that helps to pick the useful information and to fire the rest.
- **Pooling layer:** it is the block that progressively reduces the temporal and spatial size of the input trials to reduce the amount of parameters and computation in the network.
- **Fully-connected layer (classifier):** once the convolutional layers are finished, this layer is implemented connecting all the outputs from the convolutional blocks to classify the input trial.

Lastly, table 1 shows the main tuning parameters of the architecture, with some of the values that will be considered within this report and their explanations.

4.3.2 Design of CNN

In this section four different CNN architectures will be purposed: *Deep CNN for 1D*, *Deep CNN for 2D*, *Shallow CNN for temporal-spatial convolution* and *Deep CNN for temporal-spatial combined with temporal convolutions*.

Deep CNN for 1D:

Block	Parameter	Values/Description
Convolutional layer	Kernel size	40, 11, 7, 3
	Dropout	10%
	Output channel	12, 8, 4, 2
Activation function	Type	Elu
Pooling layer	Type	Mean
	Kernel size, stride	4 for both
Classifier	Input size	64
	Dropout	25%

Table 2: Tuning parameters for deep CNN for 1D convolution.

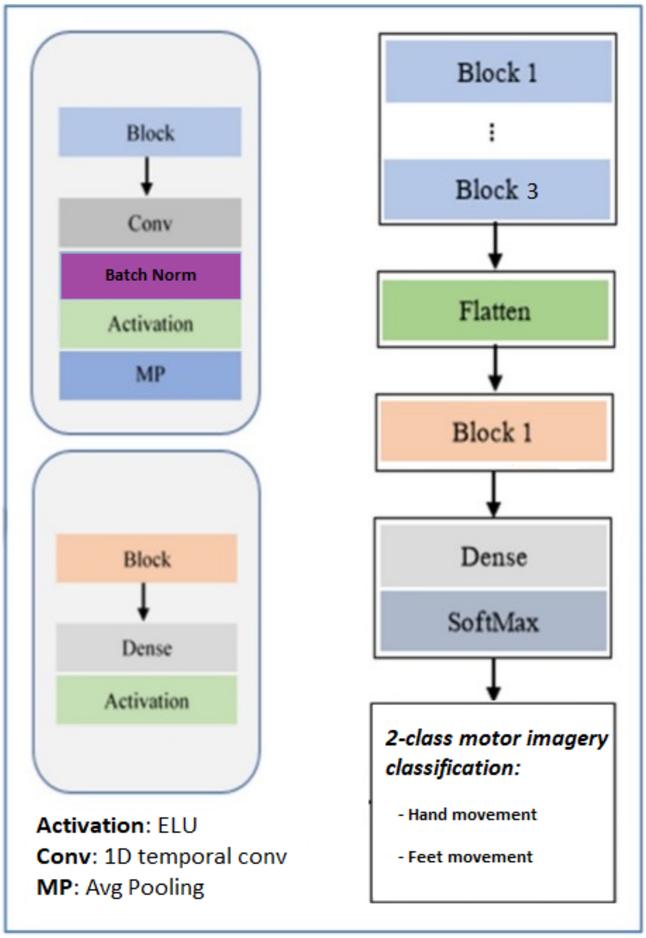

Figure 11: Deep CNN for 1D architecture.

This is the original network generated after repeated trial and error, in search for the architecture that preforms the best. Figure 11 shows the final CNN setup. It is important to notice that only temporal information is convolved (thus the reason of 1D). The tuning parameters for the final configuration are shown in table2.

Deep CNN for 2D:

This architecture is based on the previous model, but in this case is trying to also recover spatial information. Figure 12 shows the architecture, where there are convolution blocks for temporal and spatial dimensions.

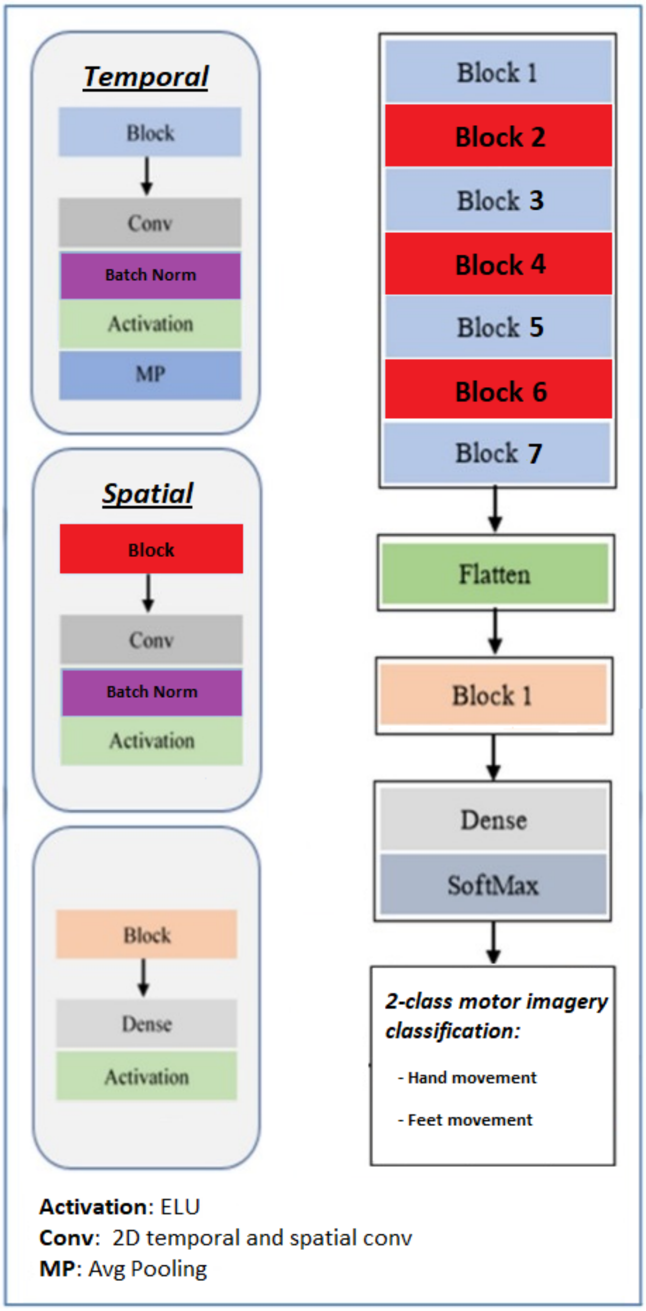

Figure 12: Deep CNN for 2D architecture.

Also, the tuning parameters for the final configuration are shown in table 3.

Block	Parameter	Values/Description
Convolutional layer (temporal)	Kernel size	9, 7, 5, 3
	Dropout	10%
	Output channel	3, 6, 9, 12
Convolutional layer (spatial)	Kernel size	5, 3, 3
	Dropout	10%
	Output channel	4, 7, 10
Activation function	Type	Elu
Pooling layer (only for temporal blocks)	Type	Mean
	Kernel size, stride	3 for both
Classifier	Input size	768
	Dropout	25%

Table 3: Tuning parameters for deep CNN for 2D convolution.

Shallow CNN for temporal-spatial convolution:

This architecture was proved performing well for two-class motor imagery task in [5], thus it was implemented in order to compare the performance with the other architectures generated. Figure 13 shows the setup provided in [5], where there is only one temporal and spatial convolution block.

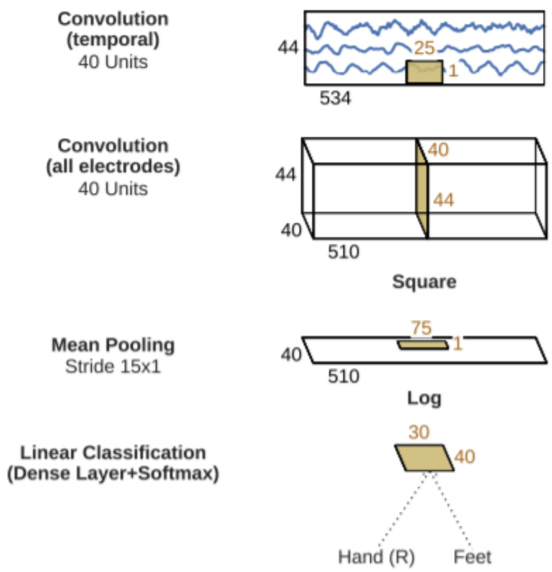

Figure 13: Shallow CNN architecture [5].

Deep CNN for temporal-spatial combined with temporal convolutions:

Like the previous architecture, this architecture was proved performing well for two-class motor imagery task in [5], thus it was implemented in order to compare the performance with the other architectures generated. Figure 14 shows the setup provided in [5], where there is one block of temporal-spatial convolution, followed by three blocks of temporal convolution.

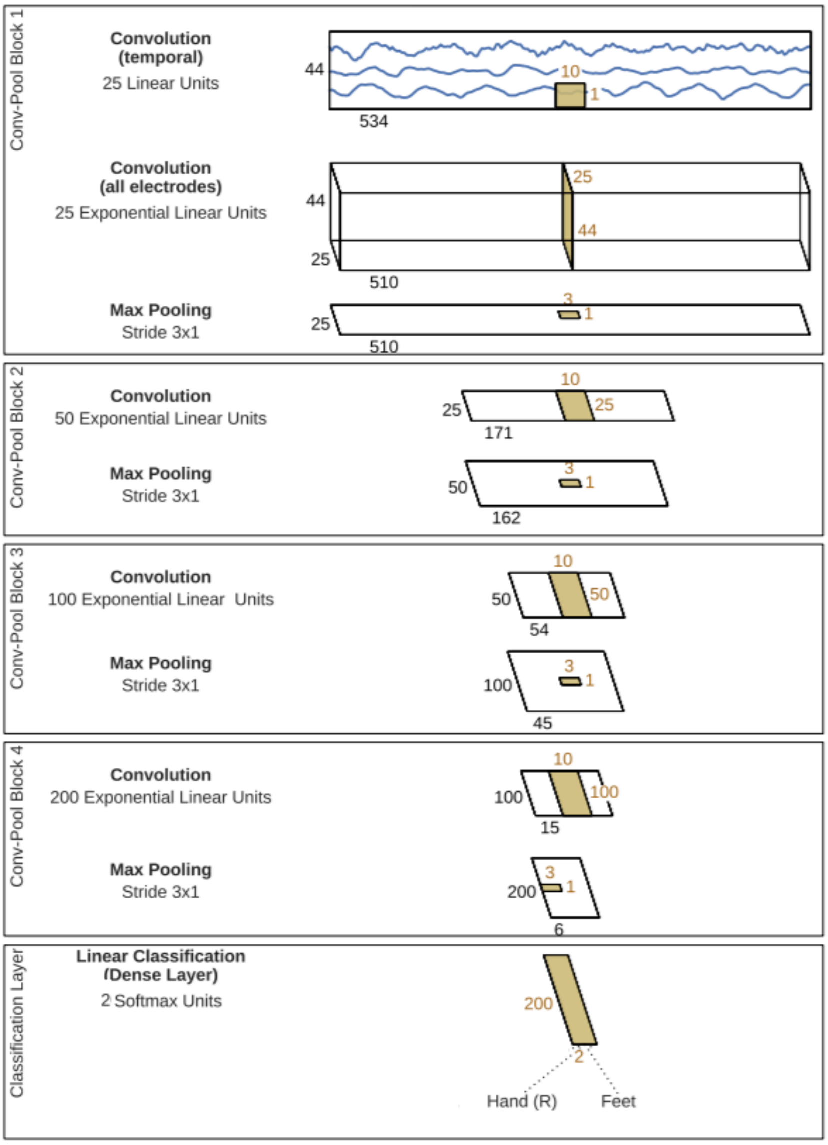

Figure 14: Deep CNN architecture [5].

Out of the architectures introduced in this section, the first one (*Deep CNN for 1D*) will be used in most of the results in the following section to decide some optimal tuning parameters. Then, a last subsection, will compare the performance of this four architectures.

5 Results

In this section, it will be shown the results obtained. First, the effect of some learning parameters for the optimizer, such as learning rate or weight decay, will be studied and the optimal value will be given. Then, some parameters for the training and the CNN model will be analysed in the same way.

Once all the tuning parameters are fixed with the optimal values, all the models introduced in the previous section will be compared. To conclude the section, the performance of the best model with the optimal parameters will be shown.

5.1 Parameters for the optimizer

In this section it will be compared the performance of Stochastic Gradient Descent (SGD) and Adam optimizer. Then, some parameters for Adam optimizer, which turns to perform better will be analysed. In all the experiments of this section the model used will be *Deep CNN for 1D*.

5.1.1 SGD vs. Adam optimizer

Figure 15 compares the training and evaluation curves for Adam (red) and SGD (blue) optimizer for 150 epochs. The train set size is 1000 trials, batch size: 6, and learning rate of 0.001 and 0.00005 for SGD and Adam, respectively.

Figure 15: Train and evaluation curves for Adam (red) and SGD (blue) optimizer.

5.1.2 Weight decay

From now and on, the optimizer used will be Adam optimizer. Figure 16 shows the error (left) and accuracy (right) graphs for different values of weight decay within the range from 1e-5 to 1e2. The following parameters are fixed; train set size: 1000 trials, batch size: 6, epochs 200, learning rate: 0.0001.

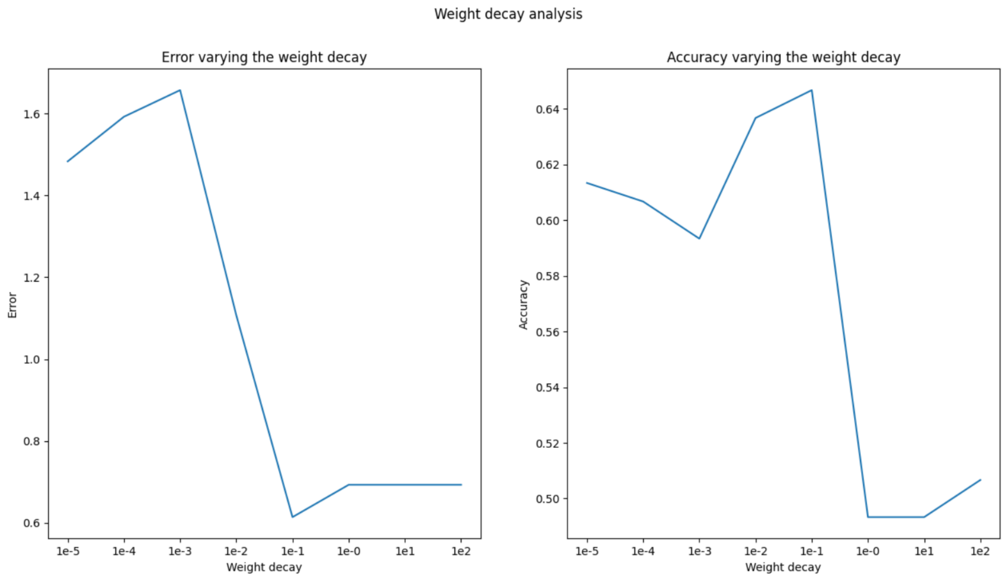

Figure 16: Error (left) and accuracy (right) graphs for different values of weight decay.

5.1.3 Learning rate

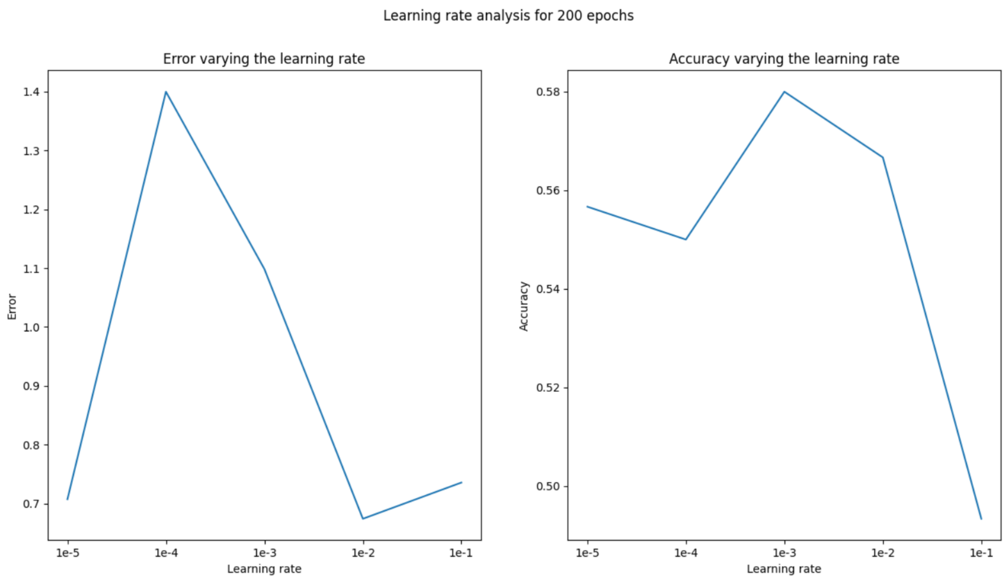

Figure 17: Error (left) and accuracy (right) graphs for different values of learning rate.

Figure 17 shows the error (left) and accuracy (right) graphs for different values of learning rate ranging from 1e-5 to 1e-1. The following parameters are fixed; train set size: 1000 trials, batch size: 6, epochs 200, weight decay: 1e-1.

5.1.4 Learning rate decay

In this section it will be evaluated if a variable learning rate would improve the performance of the model. Thus, a learning rate decay is introduced and compared with some trains with a fix learning rate. Figure 18 compares the train (doted lines) and evaluation (continuous lines) graphs for fixed learning rate of 0.00005 (blue), 0.0001 (green), and variable learning rates of 0.0001 with gamma equals to 0.96 (red) and 0.98 (black). The following parameters are fixed; train set size: 1000 trials, batch size: 6, epochs 200, weight decay: 1e-1.

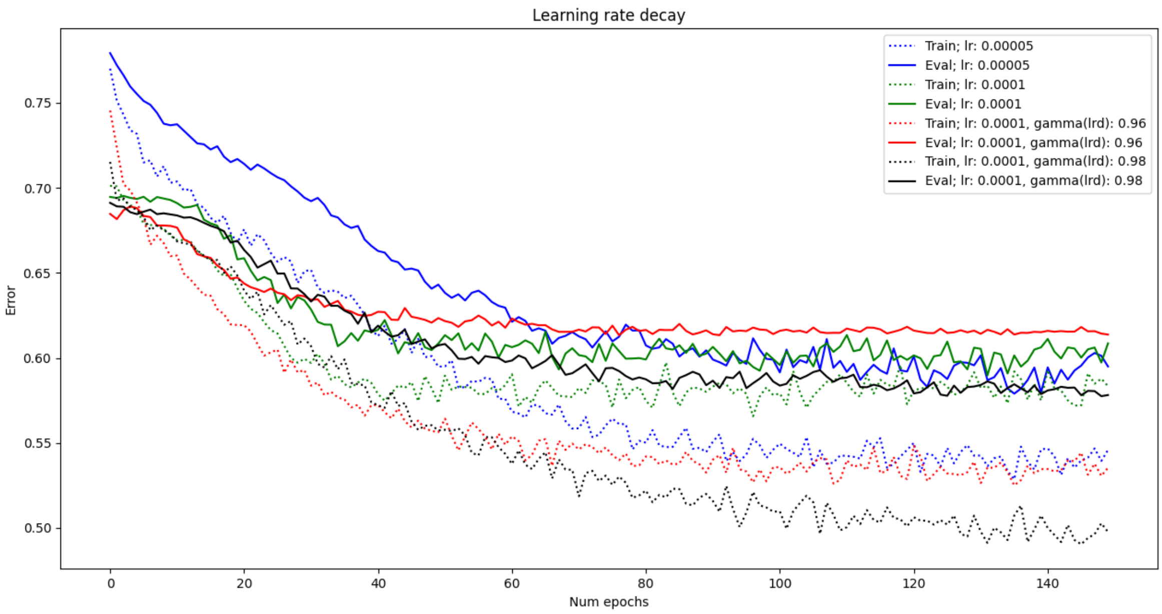

Figure 18: Comparison of fixed learning rate vs. learning rate decay.

Out of figure 18 some further comparisons were done in figure 19, where a fixed learning rate of 0.00005 (blue) is compared with learning rate decay of 0.98 for staring learning rates of 0.00005 (green) and 0.0001 (red). The fixed parameters remain the same.

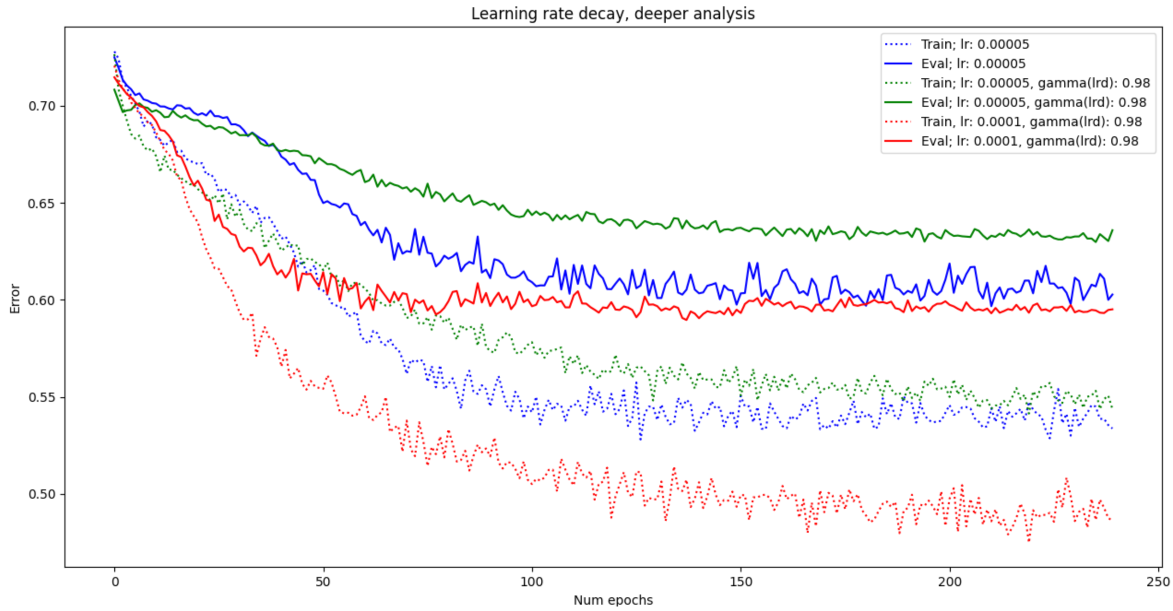

Figure 19: Deeper comparison of fixed learning rate vs. learning rate decay.

5.2 Parameters of the training

In this section, it will be evaluated the optimal values for some parameters for the training task such as batch size and train set size. The model used is the same as in the previous sections.

5.2.1 Batch size

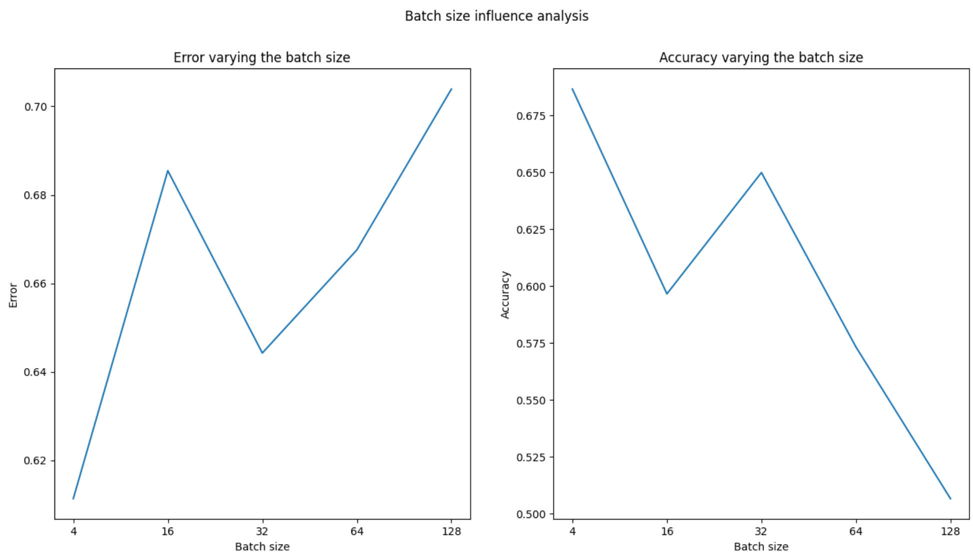

Figure 20: Error (left) and accuracy (right) graphs for different values of batch size.

Figure 20 shows the error (left) and accuracy (right) graphs for different values of batch size ranging from 4 to 128. The following parameters are fixed; train set size: 1000 trials, epochs 200, weight decay: 1e-1, learning rate: 0.00005.

5.2.2 Train set size

Figure 21 shows the error (left) and accuracy (right) graphs for different values of training set size ranging from 600 to 1400 trials. The following parameters are fixed; batch size: 6, epochs 200, weight decay: 1e-1, learning rate: 0.00005.

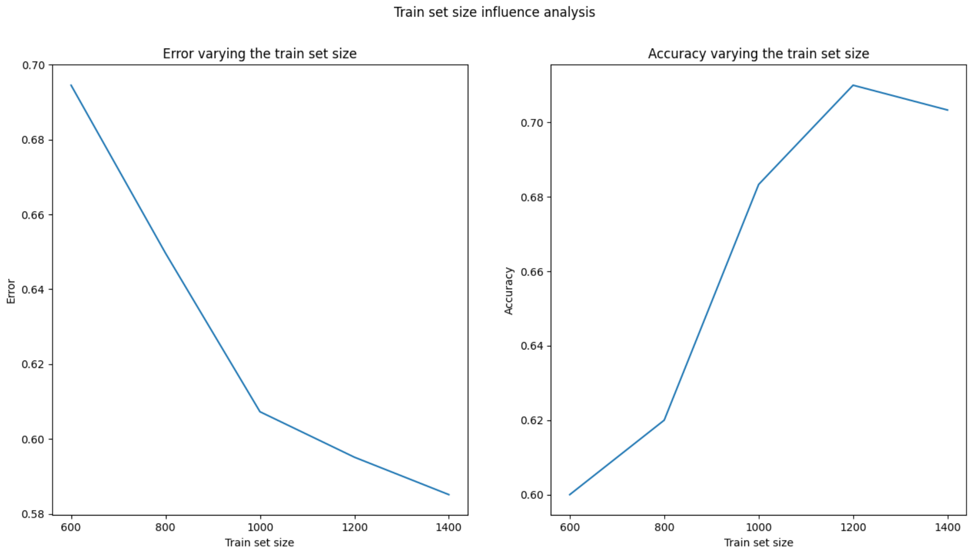

Figure 21: Error (left) and accuracy (right) graphs for different values of train set size.

5.3 Parameters of the CNN

In this section, some blocks that compound the CNN will be compared to use the most optimal one. Firstly, it will be shown the performance of the dropout layers. Then the two activation layers considered will be compared. Lastly, two types of pooling layers will be also compared. In this section it is used the same CNN architecture as in the previous sections.

5.3.1 Dropout layers

Figure 22 compares the train (doted lines) and evaluation (continuous lines) curves when applying dropout layers in the convolution blocks of the CNN. Performance without dropout (blue) is compared with performance with dropout of 10% and 40% in green and red, respectively. The following parameters are fixed; train set size: 1400, batch size: 6, epochs 150, weight decay: 1e-1, learning rate: 0.00005.

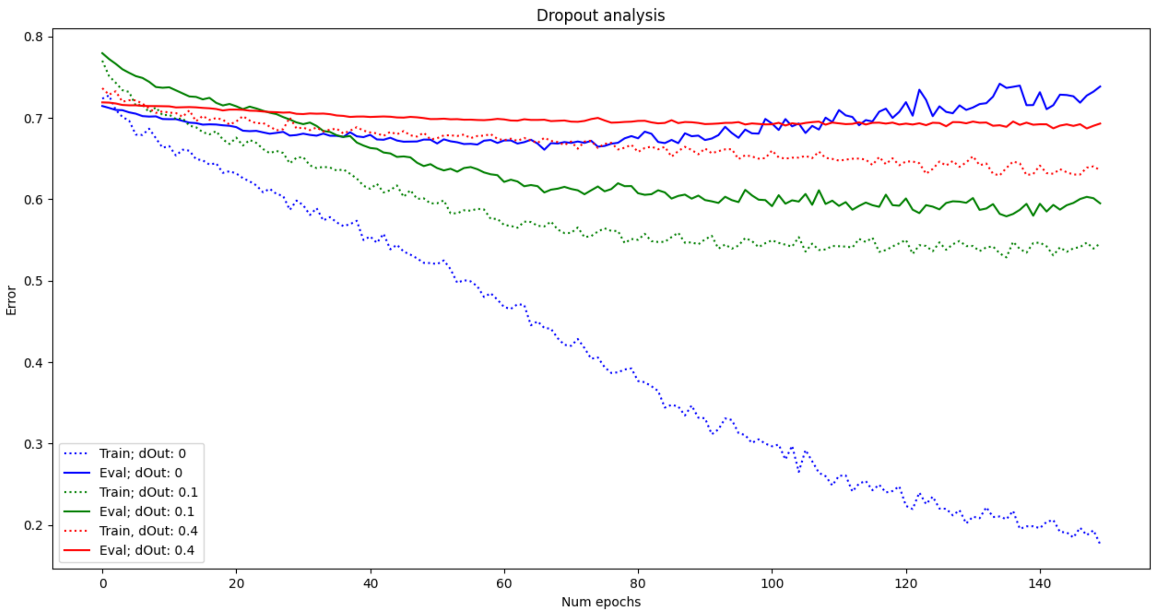

Figure 22: Performance of CNN with dropout layers of 10% (green) and 40% (red), compared with performance without dropout layers (blue).

5.3.2 Activation layers: Relu vs. Elu

Figure 23 shows the train (doted lines) and evaluation (continuous lines) curves when using ReLU activation functions (blue) in comparison to use ELU activation functions (red). The following parameters are fixed; train set size: 1400, batch size: 6, epochs 150, weight decay: 1e-1, learning rate: 0.00005.

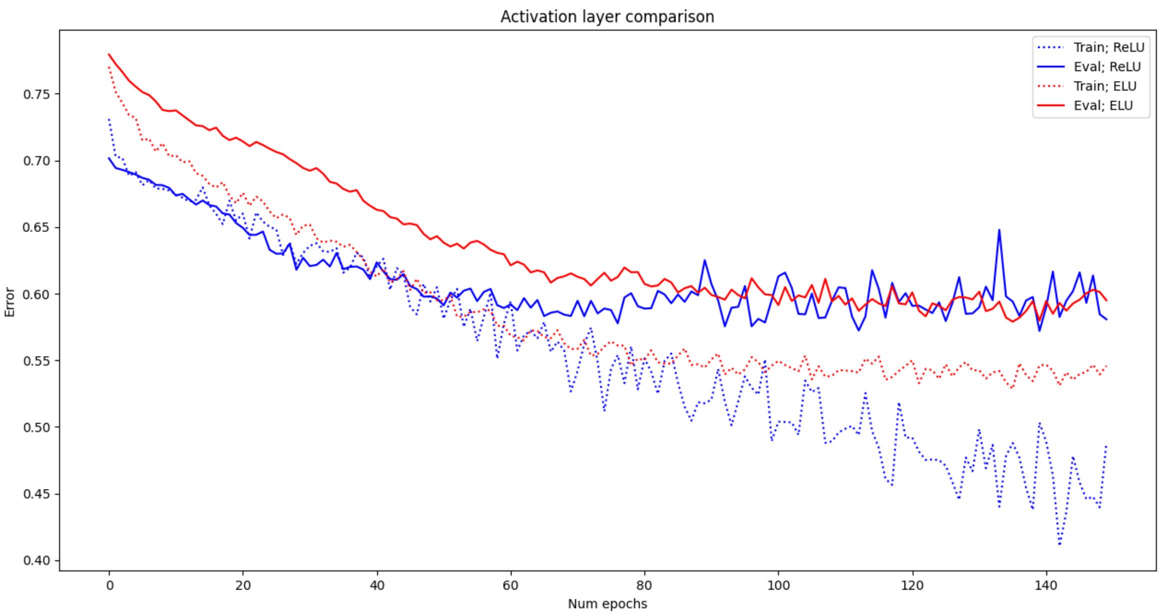

Figure 23: Comparison of CNN performance when using ReLU activation functions (blue) and ELU activation functions (red).

5.3.3 Pooling layers: Max vs. Mean

The last block that will be evaluated within the CNN architecture is the pooling layer. Figure 24 shows the train (doted lines) and evaluation (continuous lines) curves obtained form using max pooling layers (blue) and mean pooling layers (red). The following parameters are fixed; train set size: 1400, batch size: 6, epochs 150, weight decay: 1e-1, learning rate: 0.00005.

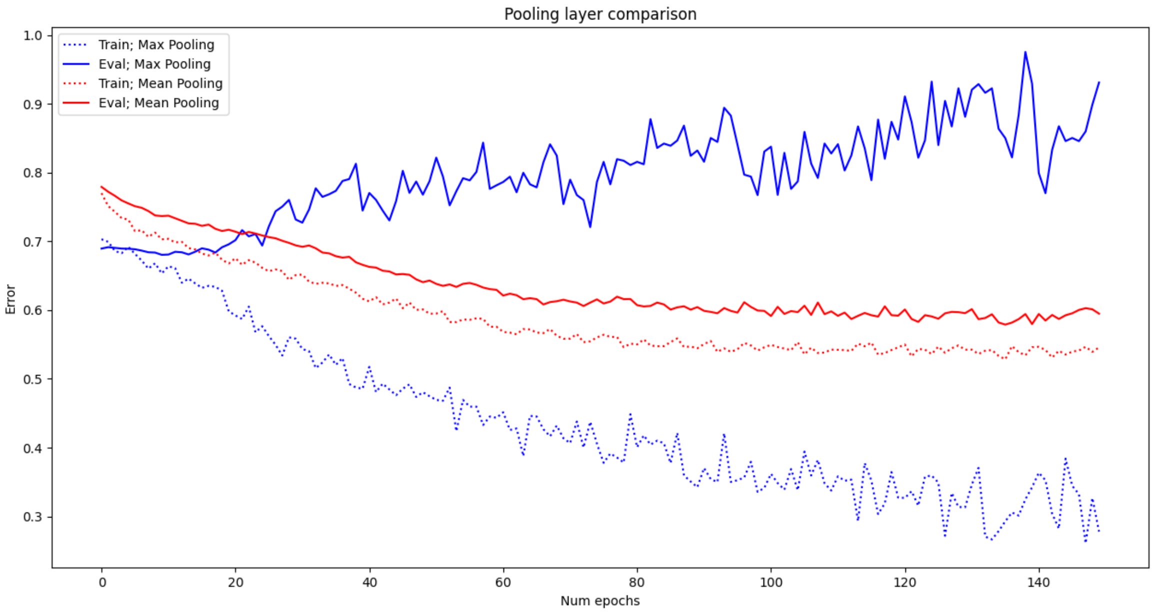

Figure 24: Comparison of CNN performance when using max pooling (blue) and mean pooling (red).

5.4 Model comparison

In this subsection, the architectures introduced in *methods and materials* will be compared to decide which performs the best. Table 4 shows the error and accuracy obtained for each of the models after training them with the following parameters: train set size: 1200, batch size: 6, epochs 200, weight decay: 1e-1, learning rate: 0.00005.

	Error	*Accuracy*
Deep CNN for 1D	0.5517	69.33%
Deep CCN for 2D	0.6341	62.67%
Shallow CNN	0.6939	45%
Deep CNN temporal-spatial and temporal blocks	0.7978	54%

Table 4: Model comparison in error and accuracy.

5.4.1 Test best model performance

By using the optimal parameters, that will be discussed in the following section, the train (blue) and evaluation (orange) curves are shown in figure 25. The highest accuracy achieved is **75.23%**. The following parameters where considered the optimal; train set size: 1200, batch size: 4, epochs 240, weight decay: 1e-1, learning rate: 0.00005, model: *Deep CNN for 1D.*

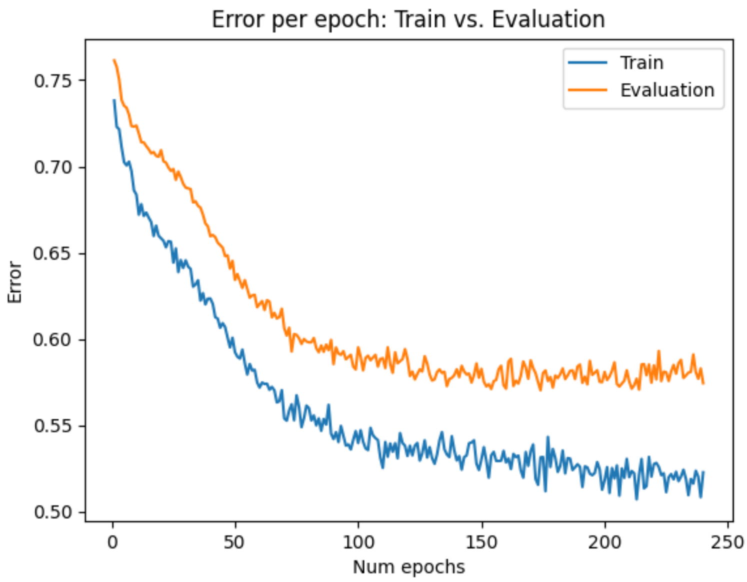

Figure 25: Train (blue) and evaluation (orange) curves for the best model with the optimal parameters.

6 Discussion

In this section it will be discussed the results obtained in the previous section. Thus, parameters for the optimizer, training and CNN will be discussed. To conclude, a discussion about the model comparison and the best model performance will be done in the last subsection.

6.1 Parameters for the optimizer

Depending on the optimizer used, and how this optimizer is tuned, the models will be able to learn faster or slower. In some, cases may not even be able to learn if the optimizer is not tuned correctly.

In figure 15 it is shown a comparison between Adam and SGD optimizer. It can be seen that the learning curves for Adam decrease gradually, while in the case of SGD there is clear over-fitting since the model is just learning for the train set but not for the evaluation set. Therefore, Adam optimizer will be the one used for training the model.

Once the optimizer is selected, it is important to control the over-fitting of the model. In the case of the data set its quite likely to over-fit the model, since the trials are compound with a large number of data points and there is not that many trials to make sure a generalization of the model. Thus, in figure 16 a evaluation of the optimal weight decay for Adam optimizer is done. The results show that the optimal value is in the order of 1e-1 since the lowest error and the highest accuracy are achieved.

Another important parameter for the optimizer is the learning rate. It determines how fast the model learns as well as how depth can be found the minimum for the optimization. In some cases, in order to find an equilibrium between helping the model to learn fast and getting to the minimum, an adaptive learning rate, called *learning rate decay* is used. Figure 17 shows that the optimal fixed learning rate could be around 0.01. However, this learning rate is too high, thus after few epochs the model stops learning.

In figures 18 and 19 is analyzed graphically the behaviour of this parameter. By comparing fixed learning rates with learning rate decay, it can be seen in figure 19 that the optimizer that performs with the least over-fitting is when the learning rate is fixed to 0.00005. Also, it can be seen the problem introduced by the learning rate decay: when the starting learning rate is too low, the model never gets to learn enough (green graphs), while in the case when the starting learning rate is a bit high, the model reaches to a minimum that is far from the optimal minimum too fast (red graphs).

6.2 Parameters of the training

Looking for optimal parameter for training will play an important role too. This is due to the fact that some training can get to be too time consuming or high computational demanding. Thus, training the model with the minimal number of trials or epochs that assure a good performance will speed the learning process. Also, the batch size is important, not only in terms of training speed (the larger the batch size, the faster it is trained), but also in terms of performance of the CNN model.

Figure 20 confirms what is expected with this sort of scenarios that the smaller the batch size, the better it will perform the CNN. The reason is that, each trial has a large amount of information, thus if the model is updated with a large batch size the randomness of the train trials will be lost, since there are few available trails for training. On the other hand, using a small batch size means that the training will take a bit more of time.

On the other hand, figure 21 shows what is expected when facing data set with a limit amount of trials. This is, that the larger the train set size, the better the model will perform. This is due to the fact that if few trials are used for training, the model will learn quite fast of this trials and there will be a clear and fast over-fitting that will prevent the model from learning for a generalized model.

6.3 Parameters of the CNN

The blocks that conform the CNN are important for the performance of the model. In deep learning, the configuration and the architecture of the neural network can vary a lot depending on the purposes of the model. In this case, it will be analyzed the impact of the dropout layers, the activation layers and the pooling layers.

Figure 22 shows the training curves for different degrees of dropout within the convolutional blocks. It can be seen how much the model over-fits when there is not dropout in the training (blue curves). Also, how difficult is for the network to learn if there is a high dropout in the model (red curves). The equilibrium is found when applying a 10% of dropout in the convolutional block for the training (green curves). Lastly, it is important to notice the formation of ripples in the learning curves due to this control layer. The reason is that, this layers are used to control the over-fitting by randomly *turning off* some connections from the network. This, also limits the accuracy of the learning at some point.

Regarding the activation layers, in figure 23 there is a performance comparison between ReLU and ELU activation layer. It can be seen that in the initial stages ReLU learns faster, but after few epochs the model stops generalizing and just learns to classify the train trials. On the other hand, ELU activation layer shows a constant decreasing of the error and a clear control of the over-fitting. Thus, ELU is the optimal activation layers for this scenario.

Lastly, figure 24 compares mean and max pooling. Like in the previous discussion, only the mean pooling shows a continuous decrease of the error and control of the over-fitting.

6.4 Model comparison

To conclude the discussion, the four models proposed in section *methods and materials* are compared with the optimal values discussed in the previous section. It is important to point-out, that this is just a why to evaluate graphically and with numbers how likely each of the models is to perform well for this task. However, the optimal parameters were chosen for the so-called *Deep CNN for 1D* model, thus makes sense that is the one that performs best (see table 4).

Lastly, in figure 25 can be seen the training curves for the best model with the optimal parameters discussed for a larger number of epochs. The highest accuracy achieved is 75.23%, which is around the values that were found is other papers with the same task. Also, it can be seen how the over-fitting is controlled during the training, which is a success since a lot of complexity-control parameters where required to introduce the needed bias to the model.

7 Conclusion and future lines

Throughout this report it was studied an evaluated the effect of some tuning parameters in the performance of a CNN model. During all the experiments, there was one main problem found in common that is the over-fitting of the model while training. This could be expected due to the large dimension of the trials and the few amount of trials available for training. This lead to dig deep in the complexity-control parameters that would introduce bias to the training so that a more generalized model could be achieved.

Regarding the optimizer, it was found out that the best optimizer was Adam with a fixed learning rate of 0.00005. However, some results show that playing with the starting learning rate and the learning rate decay ratio better results may be achieved.

Then, regarding the CNN configuration, it is important that the batch size used is small due to the low amount of trials used for training. Also, it was found out that the best activation layer is ELU, and mean pooling improves the model too. Lastly, dropout layers deem important and effective in order to control the over-fitting.

To conclude, some possible future lines will be proposed. The large amount of points per trial makes the CNN algorithms to over-fit easily. This could be overcome by implementing even deeper architectures that learn slowly from the information of each trails, and reduced drastically the information of each trial. Also, another possible solution could be testing the performance of the already implemented CNN with fragments of the trials (i.e. using 3 seconds instead of the full trial). Lastly, regarding that the trials are too pure, this may be a reason of fast over-fitting for the model. Thus, it could be interesting a data set generation with shifts of the windows of the trials some milliseconds to the left an to the right.

References

[1] Y. Narayan, "Motor-imagery eeg signals classificationusing svm, mlp and lda classifiers," 2021.

[2] M. G. C. M. Amira Echtioui, Wassim Zouch and H. Hamam, "Multi-class motor imagery eeg classification using convolution neural network," 2021.

[3] M. W. Y. I. Ward Fadel, Csaba Kollod and I. Ulbert, "Multi-class classification of motor imagery eeg signals using image-based deep recurrent convolutional neural network," 2020.

[4] O. F. David Steyrl, Reinhold Scherer and G. R. Muller-Putz, "Motor imagery brain-computer interfaces: Random forests vs regularized lda - non-linear beats linear," no. 061-1, 2014.

[5] J. T. S. Robin Tibor Schirrmeister, "Deep learning with convolutional neural networks for brain mapping and decoding of movement-related information from the human eeg," 2018.

[6] C. components. Forebrain midbrain and hindbrain. [Online]. Available: https://cerebralcomponents.weebly.com/forebrain-midbrain-and-hindbrain.html

[7] M. clinic. Brain lobes. [Online]. Available: https://www.mayoclinic.org/brain-lobes/img-20008887

[8] Q. B. Institute. Peripheral nervous system. [Online]. Available: https://qbi.uq.edu.au/brain/brain-anatomy/peripheral-nervous-system

[9] B. H. 2020. Data sets. [Online]. Available: http://bnci-horizon-2020.eu/database/data-sets

[10] F. Rehab. Primary motor cortex. [Online]. Available: https://www.flintrehab.com/primary-motor-cortex-damage/

[11] N. B. L. M. D. J. D. J. A. Kip A. Ludwig, Rachel M. Miriani and D. R. Kipke, "Using a common average reference to improve cortical neuron recordings from microelectrode arrays," 2008.

[12] C. Camacho. Convolutional neural networks. [Online]. Available: https://cezannec.github.io/Convolutional_Neural_Networks/

A Python code

A.1 Main function

```
import numpy as np
from scipy import signal
from mat4py import loadmat
from cnn1D import NetExtended1D
from ShallowConvNet import ShallowNet
from cnnTemp_Spac import DeepConvNet
from cnn2D import Net2D
import matplotlib.pyplot as plt
import torch.optim as optCNN
import torch.nn as nn
from dataset import Dataset
import torch
import timeit
import random

is1D = True
useAllTrial = False
isDatabaseReady = True


def loadControlPatients(fileName):
    X_total = []
    y_total = []
    for patient in fileName:
        data = loadmat('data/Data Control Patients/' + patient)
        duration = 5
        data = data['data']
        datasetX = []
        y = []
        for dt in data:
            aux = np.array(dt['X'])
            fs = dt['fs']
            trial = np.array(dt['trial'])
            y_dt = np.array(dt['y'])
            EEG_signals = aux.transpose()
            sig_Filtered = []
            for sig in EEG_signals:
                sig_bp = bandPass(sig, fs)
                sig_bp_notch = notchFilter(sig_bp, fs)
                sig_Filtered.append(sig_bp_notch)
            X_split = splitTrials(sig_Filtered, trial, fs, duration)
```

```
                datasetX.append(X_split)
                y.append(y_dt)
            datasetX = reshapeDataset(datasetX)
            X_wo_CAR = commonAverageReference(datasetX)
            X = []
            for i in range(len(X_wo_CAR)):
                X.append(normalization(X_wo_CAR[i]))
            y = np.concatenate(y, axis=None)
            X_total = X_total + X
            y_total.append(y)
    y_total = np.concatenate(y_total, axis=None)
    return X_total, y_total


def reshapeDataset(dataset):
    # Reshape the dataset (from 20 x 5 matrices to 100 matrices):
    datasetX_reshaped = []
    for ii in dataset:
        for jj in ii:
            datasetX_reshaped.append(jj)
    return datasetX_reshaped


def bandPass(EEGSignal, fs):
    band = np.array([5, 39])
    stopBand = np.array([3, 41])

    N, Wn = signal.cheb2ord(wp=band, ws=stopBand, gpass=3, gstop=60, fs=fs)
    sos = signal.cheby2(N=N, rs=60, Wn=Wn, btype='bandpass', fs=fs, output="sos")
    filtered = signal.sosfilt(sos, EEGSignal)
    return filtered


def notchFilter(EEGSignal, fs):
    band = np.array([48, 52])
    stopBand = np.array([49, 51])

    N, Wn = signal.cheb2ord(wp=band, ws=stopBand, gpass=3, gstop=60, fs=fs)
    sos = signal.cheby2(N=N, rs=60, Wn=Wn, btype='stop', fs=fs, output="sos")
    filtered = signal.sosfilt(sos, EEGSignal)
    return filtered
```

```python


def splitTrials(EEG_signals, trials, fs, duration):
    # timeShift = 256
    # numPoints = 522
    timeShift = 0
    numPoints = fs * duration
    X = []
    for trial_ind in trials:
        trial = []
        for sig in EEG_signals:
            trial.append(sig[trial_ind - 1 - timeShift:trial_ind +
    numPoints - 1 - timeShift])
        X.append(trial)
    return X


def normalization(EEGSignal):
    EEG_mean = np.mean(EEGSignal, axis=1)
    EEG_std = np.std(EEGSignal, axis=1)
    return (EEGSignal - np.array(EEG_mean, ndmin=2).T) / np.array(
    EEG_std, ndmin=2).T


def commonAverageReference(ds):
    dataset_wo_CAR = []
    for EEGMatrix in ds:
        CAR = np.mean(EEGMatrix, axis=0)
        matrix_wo_CAR = np.subtract(EEGMatrix, CAR)
        dataset_wo_CAR.append(matrix_wo_CAR)
    return dataset_wo_CAR


def main():
    print('Start Training')
    start = timeit.timeit()
    num_trials_train = 1200
    num_trials_test = 300
    if not isDatabaseReady:
        control_patients = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
    14]
        control_patients_train = []
        control_patients_test = []
        for ii in control_patients:
            if ii < 10:
```

```
                control_patients_train.append('S0' + str(ii) + 'T.mat')
                control_patients_test.append('S0' + str(ii) + 'E.mat')
            else:
                control_patients_train.append('S' + str(ii) + 'T.mat')
                control_patients_test.append('S' + str(ii) + 'E.mat')
        X_train, y_train = loadControlPatients(control_patients_train)
        X_test, y_test = loadControlPatients(control_patients_test)
        # saveDataset(X_train, y_train, X_test, y_test)
    else:
        X_train = np.load('X_train.npy')
        y_train = np.load('y_train.npy')
        X_test = np.load('X_test.npy')
        y_test = np.load('y_test.npy')
    # Pick random trials for train and test:
    if not useAllTrial:
        X_train, y_train = generate_random_dataset(X_train, y_train, num_trials_train)
        X_test, y_test = generate_random_dataset(X_test, y_test, num_trials_test)
    train_error_per_epoch, train_correct_per_epoch, eval_error_per_epoch, eval_correct_per_epoch, err_final, corr_final = train_cnn_model(
        X_train, y_train, X_test, y_test)
    print('Finished Training')

    end = timeit.timeit()
    print('Time of the process:')
    print(end - start)
    train_trails_number = len(X_train)
    test_trails_number = len(X_test)
    plot_eval_vs_train(train_error_per_epoch, train_correct_per_epoch, eval_error_per_epoch, eval_correct_per_epoch,
                       train_trails_number, test_trails_number)
    print('Process finished.')


def generate_random_dataset(X_in, y_in, num_trials):
    pair_data = list(zip(X_in, y_in))
    pair_data = random.sample(pair_data, num_trials)
    X_out, y_out = zip(*pair_data)
    return list(X_out), np.asarray(y_out)


def train_cnn_model(X_train, y_train, X_test, y_test):
    save_path = f'./CNN1d-model-50.pth'
```

```
    batch_size = 6
    num_epoch = 240

    params = {'batch_size': batch_size,
              'shuffle': True}

    # Generators
    training_set = Dataset(X_train, y_train)
    training_generator = torch.utils.data.DataLoader(training_set, **
params)

    test_set = Dataset(X_test, y_test)
    test_generator = torch.utils.data.DataLoader(test_set, **params)

    # Model:
    num_classes = 2
    x_axis = 3
    y_axis = 640

    # Create NN:
    model = NetExtended1D(num_classes, x_axis, y_axis)  # best
    # model = ShallowNet(num_classes, x_axis, y_axis)
    # model = DeepConvNet(num_classes, x_axis, y_axis)
    # model = Net2D(num_classes, x_axis, y_axis)

    # Loss Function:
    criterion = nn.CrossEntropyLoss()
    # optimizer = optCNN.SGD(model.parameters(), lr=0.001, momentum
=0.9)
    optimizer = optCNN.Adam(model.parameters(), lr=0.00005, betas=[0.9,
 0.95], weight_decay=1e-1)
    lr_decay_scheduler = torch.optim.lr_scheduler.ExponentialLR(
optimizer=optimizer,
                                                                 gamma
=0.98)

    train_error_per_epoch = []
    train_correct_per_epoch = []
    eval_error_per_epoch = []
    eval_correct_per_epoch = []
    for epoch in range(num_epoch):
        model, train_error_epoch, train_correct_epoch = train_epoch(
training_generator, optimizer, model, criterion)
        eval_error_epoch, eval_correct_epoch = eval_epoch(
test_generator, criterion, model)
```

```
        print("Epoch:{}/{}; AVG Training Loss:{:.3f}, AVG Evaluation Loss:{:.3f}".format(epoch + 1, num_epoch,
                      np.mean(train_error_epoch),
                      np.mean(eval_error_epoch)))
        train_error_per_epoch.append(np.mean(train_error_epoch))
        train_correct_per_epoch.append(train_correct_epoch)
        eval_error_per_epoch.append(np.mean(eval_error_epoch))
        eval_correct_per_epoch.append(eval_correct_epoch)
        # lr_decay_scheduler.step()
    # Saving the model
    torch.save(model.state_dict(), save_path)

    error_final, correct_final = eval_epoch(test_generator, criterion, model)

    return train_error_per_epoch, train_correct_per_epoch, eval_error_per_epoch, eval_correct_per_epoch, np.mean(
        error_final), correct_final


def train_epoch(training_generator, optimizer, model, criterion):
    model.train()
    error_train = []
    correct = 0
    for inputs, labels in training_generator:
        inputs, labels = inputs.type(torch.FloatTensor), labels.type(torch.LongTensor)
        if not is1D:
            inputs = inputs[:, None, :, :]
        optimizer.zero_grad()
        output = model(inputs)
        labels = torch.subtract(labels, 1)
        batch_loss = criterion(output, labels)
        batch_loss.backward()
        optimizer.step()
        # total correct
        error_train.append(batch_loss.data.detach().numpy())
        preds = np.argmax(output.data.numpy(), axis=-1)
        correct += np.sum(labels.data.numpy() == preds)
    return model, error_train, correct


def eval_epoch(test_generator, criterion, model):
```

```
    model.eval()
    error_eval = []
    correct_eval = 0
    for inputs, labels in test_generator:
        inputs, labels = inputs.type(torch.FloatTensor), labels.type(torch.LongTensor)
        if not is1D:
            inputs = inputs[:, None, :, :]
        output = model(inputs)  # forward + backward + optimize
        labels = torch.subtract(labels, 1)
        batch_loss = criterion(output, labels)
        error_eval.append(batch_loss.data.detach().numpy())  # test error
        preds = np.argmax(output.data.detach().numpy(), axis=-1)
        correct_eval += np.sum(labels.data.numpy() == preds)
    return error_eval, correct_eval


def plot_results(train_error_per_batch, correct_per_epoch, test_error_per_batch, correct_test, output_predicted,
                 output_expected):
    # Plot error per batch:
    x = np.arange(1, len(train_error_per_batch) + 1)
    plt.figure(1)
    plt.plot(x, train_error_per_batch)
    plt.xlabel('Num batches')
    plt.ylabel('Error')
    plt.title('Train error per batch')

    # Plot correct predictions per epoch:
    x = np.arange(1, len(correct_per_epoch) + 1)
    plt.figure(2)
    plt.plot(x, correct_per_epoch)
    plt.xlabel('Num epochs')
    plt.ylabel('Correct')
    plt.title('Correct predictions per epoch (max of 100)')

    # Plot test error per batch:
    x = np.arange(1, len(test_error_per_batch) + 1)
    plt.figure(3)
    plt.plot(x, test_error_per_batch)
    plt.xlabel('Num batches')
    plt.ylabel('Error')
    plt.title('Test error per batch')

```

```
    # Plot test predicted vs. real:
    output_predicted_array = []
    for sublist in output_predicted:
        for item in sublist:
            output_predicted_array.append(item)
    output_expected_array = []
    for sublist in output_expected:
        for item in sublist:
            output_expected_array.append(item)

    x = np.arange(1, len(output_expected_array) + 1)
    plt.figure(4)
    plt.plot(x, output_predicted_array, 'o')
    plt.plot(x, output_expected_array, 'o')
    plt.xlabel('Num samples')
    plt.ylabel('Classification')
    plt.title('Test classification predicted vs. expected')
    plt.legend(['Predicted', 'Expected'])

    plt.show()


def plot_eval_vs_train(train_error_per_epoch, train_correct_per_epoch, eval_error_per_epoch, eval_correct_per_epoch,
                       train_trails_number, test_trails_number):
    # Print statistics:
    print('Train error, accuracy of the training:')
    print(np.mean(train_error_per_epoch),
          np.sum(train_correct_per_epoch) / (len(train_correct_per_epoch) * train_trails_number))
    print('Eval error, accuracy of the evaluation:')
    print(np.mean(eval_error_per_epoch),
          np.sum(eval_correct_per_epoch) / (len(eval_correct_per_epoch) * test_trails_number))
    print('BEST accuracy:')
    best = np.max(np.array(eval_correct_per_epoch) / test_trails_number)
    print(best)

    # Plot test error per epoch:
    x = np.arange(1, len(train_error_per_epoch) + 1)
    plt.figure(1)
    plt.plot(x, train_error_per_epoch)
    plt.plot(x, eval_error_per_epoch)
    plt.xlabel('Num epochs')
```

```
    plt.ylabel('Error')
    plt.title('Error per epoch: Train vs. Evaluation')
    plt.legend(['Train', 'Evaluation'])

    # Plot test error per epoch:
    x = np.arange(1, len(train_correct_per_epoch) + 1)
    plt.figure(2)
    plt.plot(x, np.array(train_correct_per_epoch) / train_trails_number
    )
    plt.plot(x, np.array(eval_correct_per_epoch) / test_trails_number)
    plt.xlabel('Num epochs')
    plt.ylabel('Correct')
    plt.title('Correct per epoch: Train vs. Evaluation')
    plt.legend(['Train', 'Evaluation'])

    plt.show()


def saveDataset(X_train, y_train, X_test, y_test):
    np.save('X_train', X_train)
    np.save('y_train', y_train)
    np.save('X_test', X_test)
    np.save('y_test', y_test)


main()
```

Listing 1: main.py

A.2 Deep CNN for 1D class

```
import torch.nn as nn

# CNN parameters:
input_channels = 15
num_filters_conv1 = 12
num_filters_conv2 = 8
num_filters_conv3 = 4
num_filters_conv4 = 2
kernel_size_conv1 = 40
kernel_size_conv2 = 11
kernel_size_conv3 = 7
kernel_size_conv4 = 3
kernel_size_pooling = 2
```

```
padding_conv1 = 19
padding_conv2 = 5
padding_conv3 = 3
padding_conv4 = 2
pool_stride1 = 10
pool_stride2 = 4
pool_stride3 = 4
num_l1 = 4
dOut = 0.1


class NetExtended1D(nn.Module):
    def __init__(self, num_classes, x_dim, y_dim):
        super(NetExtended1D, self).__init__()
        self.num_classes = num_classes
        self.x_dim = x_dim
        self.y_dim = y_dim

        self.features_block1 = nn.Sequential(
            nn.Conv1d(in_channels=input_channels,
                      out_channels=num_filters_conv1,
                      kernel_size=kernel_size_conv1,
                      padding=padding_conv1),
            nn.BatchNorm1d(num_filters_conv1),
            nn.ELU(),
            nn.Dropout(dOut),
            nn.AvgPool1d(kernel_size=kernel_size_pooling,
                         stride=pool_stride1)
        )

        self.features_block2 = nn.Sequential(
            nn.Conv1d(in_channels=num_filters_conv1,
                      out_channels=num_filters_conv2,
                      kernel_size=kernel_size_conv2,
                      padding=padding_conv2),
            nn.BatchNorm1d(num_filters_conv2),
            nn.ELU(),
            nn.Dropout(dOut),
            nn.AvgPool1d(kernel_size=kernel_size_pooling,
                         stride=pool_stride2)
        )

        self.features_block3 = nn.Sequential(
            nn.Conv1d(in_channels=num_filters_conv2,
                      out_channels=num_filters_conv3,
```

```
                        kernel_size=kernel_size_conv3,
                        padding=padding_conv3),
            nn.BatchNorm1d(num_filters_conv3),
            nn.ELU(),
            nn.Dropout(dOut),
            nn.AvgPool1d(kernel_size=kernel_size_pooling,
                         stride=pool_stride3)
        )

        self.classifier = nn.Sequential(
            nn.Linear(64, num_l1),
            nn.ELU(),
            nn.Dropout(0.5),
            nn.Linear(num_l1, num_classes)
        )

    def forward(self, x):
        x = self.features_block1(x)
        x = self.features_block2(x)
        x = self.features_block3(x)
        x = x.view(x.size(0), -1)  # flatten
        x = self.classifier(x)
        return x
```

Listing 2: cnn1D.py

A.3 Deep CNN for 2D class

```
import torch.nn as nn

# ---------- CNN parameters:
input_channels = 1

# -------- 7 convolutional blocks:
num_filters = [3, 4, 6, 7, 9, 10, 12]

# ____TEMPORAL: 4 blocks_____
# Convolution parameters:
kernel_temporal = [9, 7, 5, 3]
padding_temporal = [4, 3, 2, 1]  # (kernel_temporal-1)/2
kernel_size_pooling_temp = [1, 2]
pool_stride3_temp = [1, 3]

# SPATIAL: 3 blocks
```

```
kernel_spatial = [5, 3, 3]
stride1_spatial = [1, 1]
stride2_spatial = [2, 1]

num_l1 = 46


class Net2D(nn.Module):
    def __init__(self, num_classes, x_dim, y_dim):
        super(Net2D, self).__init__()
        self.num_classes = num_classes
        self.x_dim = x_dim
        self.y_dim = y_dim

        self.temporal_block1 = nn.Sequential(
            nn.Conv2d(in_channels=input_channels,
                      out_channels=num_filters[0],
                      kernel_size=[1, kernel_temporal[0]],
                      padding=[0, padding_temporal[0]]),
            nn.BatchNorm2d(num_filters[0]),
            nn.ELU(),
            nn.Dropout(0.1),
            nn.AvgPool2d(kernel_size=kernel_size_pooling_temp,
                         stride=pool_stride3_temp)
        )

        self.spatial_block1 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters[0],
                      out_channels=num_filters[1],
                      kernel_size=[kernel_spatial[0], 1],
                      stride=stride1_spatial),
            nn.BatchNorm2d(num_filters[1]),
            nn.ELU(),
            nn.Dropout(0.1)
        )

        self.temporal_block2 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters[1],
                      out_channels=num_filters[2],
                      kernel_size=[1, kernel_temporal[1]],
                      padding=[0, padding_temporal[1]]),
            nn.BatchNorm2d(num_filters[2]),
            nn.ELU(),
            nn.Dropout(0.1),
            nn.AvgPool2d(kernel_size=kernel_size_pooling_temp,
```

```
                          stride=pool_stride3_temp)
        )

        self.spatial_block2 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters[2],
                      out_channels=num_filters[3],
                      kernel_size=[kernel_spatial[1], 1],
                      stride=stride2_spatial),
            nn.BatchNorm2d(num_filters[3]),
            nn.ELU(),
            nn.Dropout(0.1)
        )

        self.temporal_block3 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters[3],
                      out_channels=num_filters[4],
                      kernel_size=[1, kernel_temporal[2]],
                      padding=[0, padding_temporal[2]]),
            nn.BatchNorm2d(num_filters[4]),
            nn.ELU(),
            nn.Dropout(0.1),
            nn.AvgPool2d(kernel_size=kernel_size_pooling_temp,
                         stride=pool_stride3_temp)
        )

        self.spatial_block3 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters[4],
                      out_channels=num_filters[5],
                      kernel_size=[kernel_spatial[2], 1],
                      stride=stride2_spatial),
            nn.BatchNorm2d(num_filters[5]),
            nn.ELU(),
            nn.Dropout(0.1)
        )

        self.temporal_block4 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters[5],
                      out_channels=num_filters[6],
                      kernel_size=[1, kernel_temporal[3]],
                      padding=[0, padding_temporal[3]]),
            nn.BatchNorm2d(num_filters[6]),
            nn.ELU(),
            nn.Dropout(0.1),
            nn.AvgPool2d(kernel_size=kernel_size_pooling_temp,
                         stride=pool_stride3_temp)
```

```
        )

        self.classifier = nn.Sequential(
            nn.Linear(768, num_l1),
            nn.ELU(),
            nn.Dropout(0.25),
            nn.Linear(num_l1, num_classes)
        )

    def forward(self, x):
        # input: [, 1, 15, 2560]
        x = self.temporal_block1(x)  # output: [, 3, 15, 854]
        x = self.spatial_block1(x)  # output: [, 4, 11, 854]
        x = self.temporal_block2(x)  # output: [, 6, 11, 284]
        x = self.spatial_block2(x)  # output: [, 7, 5, 284]
        x = self.temporal_block3(x)  # output: [,9, 5, 94]
        x = self.spatial_block3(x)  # output: [, 10, 2, 94]
        x = self.temporal_block4(x)  # output: [, 12, 2, 32]
        x = x.view(x.size(0), -1)  # flatten
        x = self.classifier(x)
        return x
```

Listing 3: cnn2D.py

A.4 Shallow CNN for temporal-spatial convolution

```
import torch.nn as nn

# CNN parameters:
input_channels = 1
channel_temp = 40
num_l1 = 12


class ShallowNet(nn.Module):
    def __init__(self, num_classes, x_dim, y_dim):
        super(ShallowNet, self).__init__()
        self.num_classes = num_classes
        self.x_dim = x_dim
        self.y_dim = y_dim

        self.temp_conv = nn.Sequential(
            nn.Conv2d(in_channels=input_channels,
                      out_channels=channel_temp,
```

```
                        kernel_size=[1, 61])
        )

        self.spac_conv = nn.Sequential(
            nn.Conv2d(in_channels=channel_temp,
                      out_channels=channel_temp,
                      kernel_size=[15, 1])
        )

        self.pool_block = nn.Sequential(
            nn.AvgPool2d(kernel_size=[1, 100],
                         stride=[1, 100])
        )

        self.classifier = nn.Sequential(
            nn.Linear(1000, num_l1),
            nn.ELU(),
            nn.Dropout(0.25),
            nn.Linear(num_l1, num_classes)
        )

    def forward(self, x):
        x = self.temp_conv(x)  # (,40,15,2500)
        x = self.spac_conv(x)  # (,40,1,2500)
        x = self.pool_block(x)  # (,40,1,25)
        x = x.view(x.size(0), -1)  # flatten
        x = self.classifier(x)
        return x
```

Listing 4: ShallowConvNet.py

A.5 Deep CNN for temporal-spatial convolution with temporal convolutions

```
import torch.nn as nn

# CNN parameters:
input_channels = 1

num_filters = [25, 50, 100, 200]
kernel_temporal = [22, 10, 10, 10]

num_l1 = 12

```

```
class DeepConvNet(nn.Module):
    def __init__(self, num_classes, x_dim, y_dim):
        super(DeepConvNet, self).__init__()
        self.num_classes = num_classes
        self.x_dim = x_dim
        self.y_dim = y_dim

        self.temporal_block1 = nn.Sequential(
            nn.Conv2d(in_channels=input_channels,
                      out_channels=num_filters[0],
                      kernel_size=[1, kernel_temporal[0]])
        )

        self.spatial_block1 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters[0],
                      out_channels=num_filters[0],
                      kernel_size=[15, 1]),
            nn.MaxPool2d(kernel_size=[1, 4],
                         stride=[1, 4])
        )

        self.temporal_block2 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters[0],
                      out_channels=num_filters[1],
                      kernel_size=[1, kernel_temporal[1]]),
            nn.BatchNorm2d(num_filters[1]),
            nn.ELU(),
            nn.MaxPool2d(kernel_size=[1, 4],
                         stride=[1, 4])
        )

        self.temporal_block3 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters[1],
                      out_channels=num_filters[2],
                      kernel_size=[1, kernel_temporal[2]]),
            nn.BatchNorm2d(num_filters[2]),
            nn.ELU(),
            nn.MaxPool2d(kernel_size=[1, 3],
                         stride=[1, 3])
        )

        self.temporal_block4 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters[2],
                      out_channels=num_filters[3],
```

```
                        kernel_size=[1, kernel_temporal[3]]),
            nn.BatchNorm2d(num_filters[3]),
            nn.ELU(),
            nn.MaxPool2d(kernel_size=[1, 3],
                         stride=[1, 3])
        )

        self.temporal_block5 = nn.Sequential(
            nn.Conv2d(in_channels=num_filters[3],
                      out_channels=num_filters[3],
                      kernel_size=[1, 12]),
            nn.BatchNorm2d(num_filters[3]),
            nn.ELU()
        )

        self.classifier = nn.Sequential(
            nn.Linear(400, num_l1),
            nn.ELU(),
            nn.Dropout(0.25),
            nn.Linear(num_l1, num_classes)
        )

    def forward(self, x):
        x = self.temporal_block1(x)  # [, 25, 15, 513]
        x = self.spatial_block1(x)  # [, 25, 1, 171]
        x = self.temporal_block2(x)  # [, 50, 1, 54]
        x = self.temporal_block3(x)  # [, 100, 1, 15]
        x = self.temporal_block4(x)  # [, 200, 1, 2]
        x = self.temporal_block5(x)  # [, 200, 1, 2]
        x = x.view(x.size(0), -1)  # flatten
        x = self.classifier(x)
        return x
```

Listing 5: cnnTemp_Spac.py