BCI4ALS

Team Omri (2021-2022)

Tomer Roditi, Harel Rom, Nitai Kerem, and Itay Yaron

A High-Level Overview of the Project

The project included fitting a neural activity classifying system to one of three categories using a motor imagery paradigm. Our project's goal is to allow our mentor, Omri Hotam, basic and continuous communication throughout the day without him having to move his body at all (in contrast to his main means of communication today, which utilizes eye movement. For example, see Hamedi et al., 2016.

To overcome major obstacles in the way of creating such a system, namely the need to initially choose specific "neural markers" by which we could execute the classification task (Kotsiantis et al., 2007), we chose to use a deep learning network (LeCun et al., 2015) which receives EEG recorded neural activity as input and learns to extract the optimal features to maximize the classification task accuracy as we defined it.

The specific task we used in order to teach the deep learning network included three states (idle, motor imagery with the left hand, and motor imagery with the right hand), according to neural activity recorded by EEG helmets made by the course team, The helmet is an upgraded version of the 2020-2021 course helmet and is relatively more comfortable for the user, It is also cheaper and quite modular so it could potentially allow for ongoing usage, specifically adapted to Omri's needs (the option of removing specific occipital and parietal electrodes in order to allow for Omri to lean his head back on a pillow, as he normally does when sat down, is crucial. These electrodes are obviously discarded during classification). The main goal for the future is to hand the helmet over to Omri for regular use without having to rely on a team.

Practically, as is the case for standard classifier tasks, a sufficient quantity of examples is necessary for successful classifications not only during the training phase but during daily usage. Ideally, this training phase would include recordings of neural activity in different states during the day, with slightly different helmet positions, so that the network could extract the specific neural features that contribute to the classifier accurately and would be able to generalize correctly.

Throughout the project we used MATLAB to gather data for the training phase, implement the network itself, and present the network performance. The project code was shared between us via GitHub (https://github.com/tomerroditi/bci4als-online), where we also shared the recordings taken from Omri and from ourselves.

After training the network on our own neural activity (with real movement and not motor imagery), we reached relatively good results. For a general overview, when we recorded one person (Tomer) a few times across many recording sessions, we were not able to achieve a good accuracy of the network in a discrete classification task due to an insufficient amount of data. In a continuous classification, the more ambitious goal of this project, we were able to exploit the data better and create models that reached up to 100 percent accuracy in gestures classification and up to 90 percent accuracy in segment classification on the validation set.

Process of Forming the Product

Getting to Know the Mentor

Throughout the project, we got to know Omri and his family. He is 33 years old, married to Gaya, and is a father of a son. His caretaker, Rolando, is also present at all times. They are all pleasant and easy-going people, a fact that enabled us to work together towards promoting the project.

Between meetings, we communicated with Omri via WhatsApp (upon receiving his number from the course coordinators), and we were able to schedule meetings with Omri about a week in advance, preferably during noon/afternoon hours. throughout the meetings, communication was different- while the team (us) talked verbally, Omri did so via his eye-tracking system in which he types down words and sentences using eye movement for a computer to verbally convey them. Major parts of the conversations would usually be guided by our inquiries, as it is demanding of Omri to speak with his eye-tracking system. For this reason or another, he did not generally ask us too much. It is important to note that when he wishes to say something, answer a question, etc., he will look at his keyboard and you will hear him typing. It is recommended to be very patient and aware of this, as conversations with an ALS patient might seem very slow. A robust sign for you to know that he is done talking is if he shifts his gaze back to you. It is also recommended not to interfere with another question while he is typing, and to try to refrain from talking amongst yourselves.

Rolando was present at every meeting (once there was an additional caretaker), and Gaya was there once. In any case, the meetings were always conducted with (sometimes) necessary assistance for us to communicate and interact with Omri optimally.

Meetings usually lasted an hour and a half, depending on the number of recordings you acquired and the goal of the meeting.

<u>Extra note</u>: Due to parking issues, we advise those of you arriving by car to plan on getting there half an hour before the scheduled meeting (inner streets are usually a safe bet).

Understanding the Need

During our first meeting, we asked Omri about his needs for the project, and he said that he is interested in a simple system that allows him to interact via a binary communication system with a visual display, much like the system defined by the group that worked with Omri in the year prior to us. Omri did not want to expand his demands, but it would be a good idea to ask him next year if anything has changed in that realm.

It is worth mentioning that in our last meeting, Omri was interested if the system could work even when he will not be able to move his body at all. While acquiring new data (recordings) we explained to Omri that apart from imagining movement, he should also move his hand/leg to enhance the network's ability to extract relevant brain signals and to help us reduce potentials problems with the data, since imagining a movement is not so simple, especially if doing so repetitively. After achieving a reliable model for Omri while he is moving his hands (as much as he can) you should start working on a motor imagery model for him. We suggest imagining a simple movement like grabbing an object for the motor imagery task.

Characterizing the Product

The basic functions Omri asked for were:

1. An SOS button- the option of communicating during a critical situation. As few false alarms as possible.
2. An everyday "yes" or "no" response- allowing Omri to answer questions. In this state, the threshold for false alarms is less crucial, hence it is currently more doable.

As mentioned, Omri did not want to change the product, so we tried working with the code written by the team that worked with Omri the year before us (a link to last year's team's repository on GitHub: https://github.com/AssafUni/bci4als-online).

Because of the insufficient performance of last year's project (around 40% accuracy in discrete classification of movements), and because it had lots of bugs, undocumented code and was hard to modify we decided to drop all of last year's team's code, which relied on a simple SVM classifier, and a definition of the neural features inserted to the classifier manually and subjectively. Notwithstanding, the training task remained very similar, with changes regarding trial length and technical alterations relating to categorizing each event saved during the run.

In our project, we chose to use a deep learning neural network (LeCun et al., 2015). Specifically, the first method we chose was using an EEGNet network which showed high-performance accuracies, classifying motor imagery tasks among others, in different studies in which researchers used an EEG signal to classify brain activity (Lawhern et al., 2018; Wang et al., 2020).

Moreover, to successfully meet the project's main goal of classifying MI signals in real-time and continuously throughout the day, we decided to add layers into the EEGNet basic architecture such as "LSTM", "GRU" (recurrent layers), and more for optimizing the system for continuous classification. We also tried different DL architecture that utilizes the frequency domain of the signals.

Working with the mentor

Beyond what was already described under the "Getting to Know the Mentor" section, it is worth noting that the recordings took place in Omri's living room. There were quite a lot of electronic devices in the room, and accordingly line noise is an issue worth considering when recording. When we asked Omri and his caretaker to disconnect some of the devices, Omri immediately asked his caretaker to disconnect all of the devices (after a while we had to reconnect them as some of those devices are crucial for him and cannot stay disconnected for a long period).

Generally, Omri was very pragmatic during the meetings, and after a very short small talk, we immediately started working with him on the recordings. During the recordings, one or two of us sat on the sofa next to Omri (this allows us to see Omri's eye-tracking based keyboard, to see if he wants to communicate with us, and to show Omri our computer in which the stimuli of the motor imagery task were presented), while the other three students sat in from of Omri. Because he cannot turn to his sides and look at the student on the sofa, the students who sat in front of Omri kept observing him checking if he made gestures hinting at trying to communicate with us.

In the living room, there is a large TV screen that can be used potentially to present the task/presentations to Omri, but we haven't used it (we used our personal laptops for these purposes).

It's very important to check that Omri is correctly executing the gestures on the screen if you are recording him with actual movements, one wrong movement probably won't cause any harm to the model but if it's repeating itself, developing reliable models might be impossible. In MI tasks it is impossible to verify that he is executing the right movements, our advice is to check that he is focused and not sleepy during the recording stage.

Hardware

We used the dedicated EEG headset that was given to us by the BCI4ALS team (an upgraded version of the original headset that was used by the previous team who worked with Omri), without the two most posterior electrodes (O1 and O2) that were detached from the headset. The detachment of the occipital electrodes was done to fit Omri's needs, as these electrodes are located exactly where Omri places a cushion to rest his head (since we used a motor imagery paradigm this was not problematic as we are mostly interested in activity around the motor cortex).

We placed the headset on Omri's head according to the standard instructions given to us by the course instructors. This was done most efficiently when two of us approached Omri, and placed the helmet while making sure he feels comfortable during its placement. On some occasions Omri asked his caretaker to adjust the helmet/his head cushion, and this of course has consequences with regard to the level of control we have for the exact location of the electrodes. Accordingly, we did not use a very consistent protocol for the placement of the helmet with Omri, and we advise you to take into consideration this aspect of uncertainty regarding electrode locations (at least by examining your classifier's sensitivity to helmet location jitters or finding a way to assure that the location is identical every time.).

Notice that the headset has lots of hardware related problems, from unstable sample rate to extremely high noise, how we handle those cases will be elaborated later.

We used a dedicated GPU card in Tomer's laptop to train our models, for some reason (MATLAB inner bugs) the EEGNet cannot be trained on a CPU.

Software & Experimental Design

The code is distributed into ten different folders. Here we will examine their content and provide short explanations to help you navigate and use this code successfully, reading this alongside the code itself will help you get a deeper understanding of its architecture.
Notice that the code is well documented so I will spare explanations about certain functions and how they work.

The OOP architecture

The code is written in an object-oriented fashion to create readable, intuitive, and reproducible code. The classes are stored in the "classes" folder as you might guess.
We have five classes – my_pipeline, recording, multi_recording, bci_model, and bci_model_cv.
I strongly recommend going over the object's methods after reading this brief overview of the classes so you will be familiar with the full capabilities of each class.

My pipeline:
This class stores all kinds of parameters that are used through the preprocessing of the EEG recordings and the model training, you may change them to experiment with different pipelines. It is also used to store the paths of specific packages such as 'EEG lab', 'liblsl' etc.
When constructing a new pipeline object it will ask you to supply some important paths if they are not in MATLAB's search path already, it will also add all the folders, 1-9, to MATLAB's path.
It is recommended to go over the object properties (which will be elaborated on later) and to be familiar with all the different capabilities we can modify there since this is the place controlling the pipeline customizations.

Recording:
This class stores a single EEG recording file. When constructing the object the constructor will read the provided file and my_pipeline object and apply the desired preprocessing such as segmentation, filtering, normalization, and feature extraction. For further preprocessing operations such as

oversampling (for imbalanced data), creating a data store, and augmentations we will use the object methods under the “preprocessing methods” code block.
There are methods for different visualizations of data and segment predictions, they are useful for inspecting our signals and model performances.
Notice in the constructor function that there is also an option to read edf files (we use xdf files to store our data), this is useful if you want to use public data sets, specifically the data set I downloaded which you can find in the recordings folder (numbered folders), you can modify the function edf2data to match different public data sets.
This class supports empty object construction.

<u>Multi recording:</u> *subclass of recording*

This class is used to aggregate recording objects, to create one object that holds the data of multiple recordings. This object can manage big data sets that will not fit in the working memory so don’t be afraid to create lots of data! This should be automatic but currently, it’s done manually, just change the big data property value (In the class script) to true before creating the object.
The object constructor inputs are the desired recorders’ names, file numbers, and an optional my_pipeline object. Each recording file will be converted into a recording object and then aggregated one by one. Big data sets (the object’s segments and features properties) are stored in the “data” folder and used as a database for the object’s data store.
This class supports empty object construction.

<u>BCI model:</u>

This class is used to train classification models. Its inputs are recording, or multi-recording objects referred to as train and val (validation). The object holds his training and validation sets as well as the pipeline object that the data it was trained on used. Notice the gesture detection block in this object, this is the final method we are using to evaluate our models. It tells us how many movements we managed to detect in the recording and how much we missed. To fully understand how this is calculated you should go over this part in detail, it is well documented and has explanations inside the code.
To calculate gesture detection the object has these 3 properties:

1. ‘conf_level’ - which stands for confidence level, it’s the number of predictions in a row of some class, except idle, required to classify a gesture.

2. 'cool_time' – number of seconds to not classify after a gesture was detected. The idea behind this property is that when we have a conversation, we are not rapidly answering question after question but rather have small pauses, it also helps us prevent repetitive gesture recognition.

3. 'max_delay' – the maximum time delay between the start of the gesture (the moment we start thinking of the movement) and the gesture detection time. We don't want the system to lag in a conversation hence we set a maximum delay time. This is purely for when we calculate the gesture detection confusion matrix so we would consider late responses as false positives rather than true positives.

Notice the threshold property and how we calculate it in the construction function, it's possible to set new values for it using the 'set_threshold' method. We use this threshold for our custom classification function, "evaluation", which helps us reduce false positives (in general reducing the number of false positives is very important so the system will remain reliable and accurate, we rather miss gestures than falsely classify ones!). Another thing to notice is how we save and load the model. To save memory we do not save the object with its recording objects but only with the names that were used to create each one of them, then when we load the object we can recreate the recording objects – check the "save and load models" code block.

<u>BCI model cv:</u>

This object is used to do cross-validation training. The object's constructor inputs are a multi-recording and an optional number for K-folds (default is the number of recordings in the multi-recording). The object trains k-fold models and stores them in the model property. Use this class to estimate a model's stability and to get its mean performances.

Constructing custom pipelines

We will now go over how to create custom pipelines using these classes. In general, the pipeline includes data preprocessing (segmentation, segments rejection, filters, normalization, feature extraction, oversampling, creating data store, and augmentations), training a model, and evaluating it.

<u>The "my_pipeline" object:</u>

<u>Classes labels and markers properties:</u>

- class_names – a char cell array with the classes names you would like to load from the files

- Class_marker – a char cell array, the marker sign of each class (needs to be aligned with class_names, meaning the first name corresponds to the first marker and so on), you may use more than one marker for a single class, just use a char cell array instead of char. Default – {'Idle', 'Left hand', 'Right hand'}
- Class_label – numeric array, the label of each class in the class_names (also needs to be aligned with it). You may set more than one class to the same label, this will automatically merge the classes into one class, check the 'fix_class' method. To reject classes use the label -1, do not use 0 as a label! Default – {'1.000000000000000', '2.000000000000000', '3.000000000000000'}
- end_trial (char) – the marker for ending a cue (each cue is a "trail").
- Default – '9.000000000000000'
- start_recording (char) – recording start marker. Default – '111.0000000000000'
- end_recording (char) – recording end marker, default '99.00000000000000'

<u>Model-related and feature extraction fields:</u>

- model_algo (char) – The DL model to train, set this to any file name (excluding the ".mat") that is in the "DL pipelines" folder. Default – 'EEGNet'
- feat_or_data (char) – set this to either "data" if you want to train a model on the segments themselves or "feat" if you want to train the model on extracted features. Default – 'data'.
- feat_algo (char) – the feature extraction algorithm, set this to any file name (excluding the ".mat") that is in the "feature extraction methods" folder or to 'none' if you don't want to extract features. Default – 'none'.

<u>Segmentation-related fields:</u>

- cont_or_disc (char) – specify if you want to segment the signal continuously or discretely. Setting to "continuous" will apply a moving window on the raw signal while rejecting unphysiological segments, contrary to the "discrete" mode which will only segment the specific times where a cue was given while recording. Default – 'continuous'.

<u>Discrete mode-related fields:</u>

- Pre_start (double) – time is seconds to segment before the cue was given. Default – 0.75.
- Post_start (double) – time in seconds to segment after a cue was given. Default – 2.

Continuous mode-related fields:

- seg_dur (double) – time in seconds of each segment. Default – 2.
- overlap (double) – time in seconds of overlapping between consecutive segments. Default – overlap – 0.5.
- sequence_len (int) – the number of segments in a sequence, must be equal to or greater than one, this is used in sequence models only. Default – 1.
- sequence_overlap (double) – time in seconds of overlapping between each segment in a sequence. Default – 1.
- threshold (double) – a threshold for segment labeling in the 0-1 range. If more than the threshold percentage of the segment is in a cue period, it will be labeled as that cue label otherwise it will be labeled as idle. If sequence length is more than 1 the label is determined by the last segment in the sequence. Default – 0.7.

Filters properties:

- buffer_start (int) – number of samples to include in the segment before the real samples of the segment. Buffers are used to avoid getting the unstable part of the signal (due to the filters) in the segment; we remove those samples after filtering the signal.
- buffer_end (int) – number of samples to include in the segments after the real samples of the segment, since we are using IIR filters we can set this to 0.
- high_freq (double) – IIR bandpass high cut-off frequency in HZ.
- high_width (double) - the width of the transition band for the high cut-off frequency.
- low_freq (double) - IIR bandpass low cut-off frequency in HZ.
- low_width (double) - the width of the transition band for the low cut-off frequency.
- notch (double array) – frequencies (in HZ) to apply IIR notch filter on.
- notch_width (double) – the width of the notch filters.
- eog_artifact (bool) – true, removes EOG\EMG artifacts using BSS (blind source separation) algorithm, false do not remove EOG\EMG artifacts (same algo used in the eeglab package)
- remove_chan (double array) – electrodes to remove from the data (unused electrodes for example)
- avg_reference (bool) – true, re-reference electrodes to CZ. False, do not re-reference.

Normalization properties:

- quantiles (double array) – quantiles of each electrode to normalize by

augmentation properties:

- x_flip_p (double) – the probability to apply x-flip augmentation on each trial when reading data from data stores
- wgn_p (double) – the probability to apply white gaussian noise on each trial when reading data from the data store.

DL model training properties:

- verbose_freq (int) – number of iterations to print model checkpoints
- max_epochs (int) – maximum number of epochs for training a model
- mini_batch_size (int) – the size of a mini-batch, number of trials to feed the model in each call
- validation_freq (int) – the frequency, in iterations, of validation checkpointing
- learn_rate_drop_period (int) – number of epochs to pass before dropping the learning rate

Hardware properties:

- sample_rate (double) – the headset sample rate, should remain constant.
- electrode_num (int array) – array containing the electrode numbers
- electrode_loc (char cell) – the location of each electrode

Paths properties:

- root_path (char) – the local path of the folder containing the code folders
- eeglab_path (char) – the local eeglab path
- lab_recorder_path (char) – the local path for the lab recorder package
- liblsl_path (char) – the local path to the liblsl package
- channel_loc_path (char) – the local path to the channel location file (currently not used)

to create a custom pipeline, simply create a “my_pipeline” object and use it as an input to construct recording or multi-recording objects which will be used to construct a “bci_model” object as well. Check the “train_model” script in the scripts folder for a basic demonstration of creating a custom pipeline.

Constructing new feature extraction methods:

The folder “feature extraction methods” hold functions that extract features from segments (or sequence of segments). To create a new feature extraction method that can integrate with our classes you will need to create a function with this signature:

function features = my_feature_extraction_method (recording)

Save the function in the “feature extraction methods” folder. The function input is a recording object from which you can access all its data (raw data, segments, etc.) and parameters, and the output is the feature matrix. The feature matrix must have sequences concatenated in the 4th dimension (even if sequence length is one you need to create this dimension and set it to one, it will be removed later) and trials concatenated in its 5th dimension. You can treat dimensions 1,2,3 however you like but notice that the best practice is to set them to be Spatial, Spatial, and Channels (image channels not electrodes!) in that order. In case you are constructing a one-dimension array feature just set the rest of the dimensions to one.

Constructing new DL models:

The folder “DL pipelines” holds functions that train a DL model. To create a new DL model that can integrate with our classes you will need to create a function with this signature:

function model = my_dl_pipeline(train_ds, val_ds, my_pipeline)

save the function in the “DL pipelines” folder. The output of the function is a trained ‘DAGNetwork’ or ‘SeriesNetwork’ object. The first two inputs are data stores for training and validation of the model, reading from those data stores will output a cell array with two columns, in the first column there is the data and in the second the label (categorical objects). The data dimensions are aligned with MATLAB’s demands for image or sequence input layers, meaning the dimensions are – [Spatial, Spatial, Channels, Sequence, Batch]. The channel (image channel) dimension is set to one if we are using the segments as the training data. If we are using a feature matrix the dimensions might be different according to the feature extraction method you applied. The sequence dimension is removed if there is no sequence, meaning the dimensions are – [Spatial, Spatial, Channel, Batch]. The third input is a ‘my_pipeline object- you may use it to set some of the training options parameters such as ‘max epochs’. Check the available DL functions for examples.

Constructing new classic ml models:

In the “train_my_model” function there is a dedicated place for classic ML algorithm, I decided to leave this unused for now since it’s clear that DL is superior to classic ML in our task, if you want to use classic ML then you should place your model here and set in the pipeline object the property "model_algo" with any name that doesn't exist in the DL pipelines folder. It's very likely that errors will occur while using classic ML since I haven't tested and modified the code to work properly with those models.

Recording new data files

- make sure that psychtoolbox package is properly installed – run the 'SetupPsychtoolbox.m' function (located in the package folder)

<u>To record new data files follow the instructions below:</u>

1. connect the "dongle" to your computer
2. open the "open BCI" GUI (the .exe file)
3. select "cyton" and then "serial (from dongle)"
4. select the "16 channels" button in "channel count" and press "AUTO-CONNECT"
5. open the "networking" window in the GUI and select "LSL", press start streaming
6. open the "lab recorder" GUI (.exe file) and select a path to save the file into in "study root"
7. change the file name to "EEG.xdf"
8. open the MATLAB script "new_rec" in the scripts folder and run it
9. follow the instructions in the command line

recording data files are stored in the "recordings" folder, each recorder has its own folder. To add new recordings files you must create a new folder with a unique number in the following templates – '###', where '#' stands for any digit, 0-9. Inside each folder save an EEG.xdf file. Check the "tomer" folder for an example of how to organize your recording files.

To change new recording settings just change the parameters in the beginning of the "new_rec" script. For further modifications, you will need to edit the "record_me" function in the "functions" folder. When loading the data files we discard any recording time prior to the recording start marker and after the recording end marker so it is ok if it takes time to start the simulation after starting the stream recording or ending the stream recording after finishing the simulation. Notice that before any cues are given there is a period of idle cue (just a white cross image), currently 40 seconds, this is used to prevent unused cues recording times due to the buffers we are using, it might also be useful as a reference signal to use for denoising in the preprocessing part. I have not implemented this kind of denoising, but you may consider using it to improve the SNR.

Adding new classes is very simple, add its marker signature (an int) in the “markers” variable and add its cue image in the “images” folder inside the functions folder. It’s important to make sure that the image format is “.jpeg” and that the image name is the same as its class marker signature or an error will prompt. After adding new classes it’s recommended to add them in the “my_pipeline” class-default values for “class_names” and “class_markers” properties so you could keep track of the markers you used and their motor imagery task.

Keeping the markers for recording\trial start\end and for each MI task constant is very recommended to keep all recording files in a uniform format, it will help you prevent bugs and mistakes. The psychtoolbox package might prompt warnings about inaccuracy timing- do not let it bother you, it is ok!

Recordings Signals quality

Due to the fact we are using low-cost equipment, we experienced lots of troubles while recording with the headset. Sometimes we had to throw complete recording files due to hardware problems such as extreme noise in specific electrodes, damaged sample rate, and unphysiological amplitudes (above 100 microvolt) of the signal. Use the “signal_visualization” script to visualize the recordings in different ways, for examples of extreme noise and too harmonic signals you may refer to recording 6 from Tomer’s recordings, and for unphysiological amplitudes refer to recordings 16\17 of Tomer as well.

Discarding damaged recordings:

The easy option to deal with recordings that have extreme noise is to not use them, this leaves us with the problem that lots of our recordings have brief times of extreme noise hence rejecting them completely minimizes our data set. There are cases where we don’t have many options but to not use the whole recording such as when the sample rate is damaged. The true sample rate is supposed to be 125 HZ but, in some recordings, it deviates too far from it. For example, I encountered recordings with an effective sample rate of 96 HZ, probably due to hardware problems. In the recording object constructor, we automatically reject any recording that has a lower sample rate than 124.5 or higher than 125.5, leaving room for errors since the effective sample rate is never perfect, and since this is low-cost equipment, it deviates from the desired sample rate with relatively large error.

Discarding damaged segments:

As mentioned before, rejecting complete recording files due to brief times of extreme noise is not recommended, hence in the segmentation process, we discard only the segments that are not physiological and extremely noisy. This method allows us to extract good quality signal segments from damaged recordings giving us the opportunity to increase our data set. To truly understand these problems, how they visualize and how we reject segments you should go to the “segment_continuous” function and uncomment the commented code (follow the instructions above the commented code lines), and then run some script that reads recordings files – “signal_visualization” for example. Rejecting these segments is crucial to the success of our models since this data is not a good representation of EEG signals and it might lead the model to learn false features.

Optimizations

After finding a good model to classify our data we would like to maximize its capabilities. We should check different pipeline schemes and how they affect the model performances. For this purpose, I created the “pipeline_optimization” script.

The “pipeline_optimization” script is an example of how you may search for the optimal pipeline. The script creates “my_pipeline” objects for different pipelines and trains a model for each one of them. It then presents the results in a table for you to pick the best model. How to determine who is the best model is a great question, our final goal is to maximize the gesture detection accuracy while minimizing the gesture miss rate and the time delay. You should create a metric that considers those parameters to help you choose the best model.

Notice the method “find_optimal_values” in the “bci_model” class, it is used to find the best parameters of the confidence level and cool time of the model to get optimal results for the gesture metrics and sets them as the object values of those fields. You may change the function I chose to calculate the metric we base our choice on.

Evaluating models

After achieving a good model it needs to be evaluated. We would like to check the model stability, find its optimal working point, design better classification functions, try to explain how it classifies, what features are extracted from the data, and more.

Cross validations:

To cross-validate a custom pipeline, create a single multi-recording object which will be partitioned in the “bci_model_cv” object. The “bci_model_cv” constructor inputs are a multi-recording object and an optional input of the number of folds (K-Folds, default is the number of recording in the multi-recording). The object trains K models according to the K-Fold partition. The models are stored in the object as well as the data that is used to train each one of them. The mean and std of the models’ accuracy, gesture accuracy, and more are computed and stored in the object properties. The goal is to find models that have high mean accuracy and low mean miss rate while maintaining small variance, this means that the model is stable – getting the same results for different training inputs. Check the “train_model_cv” script.

Classification function and working point:

Probabilistic models (what the pipeline is suited for) outputs are probabilities of different classes, the default classification function is to classify the input data to the class with the highest probability. To improve our model reliability we construct a custom classification function. Since false positives are what we try to prevent as much as possible we will find the working point (probability threshold) that classifies the idle class with the highest accuracy. This is the threshold of the model, if the idle class probability is above the threshold, then it is classified as idle, otherwise, it will be classified as the class with the highest probability (excluding idle class). You may change the model threshold with the method “set_threshold” of “bci_model” objects. To change the classification function edit the “evaluation” function.

Model explainability:

One of the important fields when working with DL networks is the ability to explain the model decisions. We would like to know how the model classifies our data, what parts of the signal influenced that decision, and more. Using the EEGNet architecture allows us to explain our model in terms of temporal and spatial filters, where the first convolution layer is referred to as the temporal filter and the second convolution layer is the spatial filter. Use the “plot_weights” function or the “EEGNet_explain” method to visualize those filters. You should add more explainable AI methods to help you improve your models, check the “model_explainability” script.

Model visualization:

Another way to inspect our model performance is to check its outputs of different layers, this can help us determine if the model projects similar data points to close outputs. We can also visualize our predictions to see where our models fails and where it succeeds. Check the "model_visualization" script for examples.

Saved models:

The models are saved in a compact way, check the "save and load models" code block in the "bci_model" class script to learn how to load and save models. Compact model saving allows us to create and save lots of models so we could compare them later and choose the best ones. You should save your models in the folder "figures and models".

Cross subject models:

EEG data is often very different from one user to the other, thus making robust BCI systems is challenging. A possible solution is to just train a completely new model for each user, this will require us to gather lots of data from every user. Another possible solution is to use transfer learning on a model that we trained with lots of data from a single user, with a relatively small amount of data from a new user. Consider exploring the second solution as the method for adjusting the BCI system to new users, check the "transfer_learning" script for an example of how to do it.

BCI interface

Our final goal is a reliable real-time brain-computer interface!

The implementation of our real-time classifier is the "BCI" script. After connecting the headset and starting an LSL stream using the open BCI GUI you may run this script. Choose a model for classification and choose either to perform a quick fine-tuning to the model before starting using the BCI or start using the BCI immediately.

Before starting the fine-tuning or the real-time classifier we need to verify that the signal quality is good, we start withdrawing data from the LSL stream to validate that the signal is similar to the signals we trained our model on, meaning no loose electrodes, unphysiological data or high noise.

The fine-tuning is done by recording new data and then training the model on this recording for a few epochs with a low learning rate. The lab recorder GUI will open automatically and then you need to follow the same instructions as in recording new data. In future work, this part should be converted to be completely automatic. Fine-tuning might help us improve the classifier results since every time we wear the headset its location is a bit different, letting the model tune to the current location might be useful, but this hasn't been tested to help yet.

Finally, we are ready to start our real-time classification. We define a timer that calls the "my_bci" function every X seconds, where X is the time between overlapping segments we trained the model with. The first few calls might be longer since we are initializing the preprocessing filters, and we are also waiting for enough data to be available in the stream to be able to process it. You can choose how many times to call the function, you may create an infinite loop.

Final General notes

While training a new DL model that has batch normalization layers it's possible that the validation data will have a higher loss and lower accuracy than the training data and at the end of the training, it will suddenly be improved to be roughly the same as the training set. This is caused due to the way Matlab implemented their validation classification while training, the batch normalization layers are not assigned with the normalization values and they are calculated per batch, hence if we have a significant difference between recordings, we might get very different normalization values between the train and validation sets. When the model has finished training the batch normalization layers are set with their normalization values (calculated from the train set during training or after training has finished) and then we get better results for the validation set. So just consider that at training time it might look like the model is overfitting while it's not!

At the beginning of every script there is a call to the function "script_setup" this function ensures that all the paths related to the code are in MATLAB's search path.

Mac users might experience bugs related to paths.

The following packages are necessary for the code to run:

1. DSP toolbox
2. Communications toolbox
3. Statistics and Machine Learning toolbox
4. Deep Learning toolbox
5. Parallel Computing toolbox
6. Computer Vision toolbox
7. psychtoolbox 3

Make sure they are installed before starting to experiment with the code

The script "notes" is used for writing notes, feel free to use it as well.

The folder "documents" stores all kinds of documents including this document and some useful articles, you should check its content, you might find some useful things there.

This code is a great head start for MI classification research, I tried making it as adaptable and convenient as possible. If you encounter any bugs or have issues using the code, you may contact me at tomerroditi1@gmail.com and I will try to help or guide you.

Currently best models

Several DL architectures gave us high performances – EEGNet, EEG-stft, EEGNet-lstm (check the code folder DL pipelines). Achieving roughly 85-90 percent accuracy on the training and validation segments (using Tomer's recordings, recording 11 used as validation), and 80-100 percent accuracy on their gestures with a 0-30 percent miss rate. You may load them from the figures and models folder to check their pipelines and performances.

When trying to cross validate using these models we had high accuracies in some validation sets and low accuracies on others, the recordings with the low accuracies were the same every time and with every model, giving us the feeling that those recordings that the model "failed" on might be not very good. This might be due to inaccurate headset placement creating high variance between recordings data. We suggest creating 10 or more recordings, with 20-30 trials per class (left and right) each, in one headset placement. This will eliminate the electrode placement problems and might confirm if this was the problem or if we are facing different issues.

Future work and improvements

We summarized a few aspects that can improve our work and pave the way for an improved classifier for everyday use:

1. If possible, try to communicate with other groups to have shareable data. Ideally, if data were shareable across groups, each team would benefit from rich training which could be used by different online interfaces. This is especially important when using a deep learning network as we did. This would require using similar paradigms (motor-imagery in our case) and keeping similar recording variables (for example classes recorded, markers, etc.). Comparing our

classifier to the classifiers of other teams which preferred using simpler classifiers requiring less data, we found our classifier superior in terms of cross-validated accuracy although our data was limited to data recording from members in our group and from Omri. We think that this advantage over a simpler approach would be even greater with large data sets. **Creating more high-quality data is currently the top priority mission, it will most likely boost the model performances the most!**

2. Monitoring data quality during the recording was found to be tricky, and sometimes led us to 'throw away' full-day recordings with Omri. Bad electrodes or high levels of noise that were absent or not detected at the beginning of the recording is a serious problem with the current hardware, and accordingly, we suggest a few ways to overcome this issue:
    a. Automatic monitoring of recording issues, using a dedicated graphical user interface (GUI). This would require adding an automatic noise/malfunctioning electrodes/no variability in the signal of an electrode/other simple features indicating high SNR, and a small GUI during online co-learning (e.g., a small red light when detecting an issue), and writing the problem to a log file.
    b. Using an autoencoder to detect high SNR efficiently and automatically.
    c. Opening the OpenBCI GUI on another monitor (for example the TV screen of Omri or a dedicated tablet). This would enable online monitoring of signal quality, similarly to the validation process we usually do at the beginning of the recording session (looking for power low distribution in the frequency domain, monitoring each electrode across time searching for biological level amplitudes, etc.). This is important because when using a single monitor the whole screen is occupied, and the OpenBCI GUI cannot be observed.
3. Implementing and checking the online interface of the project. We had a few ideas for additions that can be helpful in that regard:
    a. Training the classifier during the Co-Learning session. In addition to 'teaching' the user (e.g., Omri) how to tune himself and the way he imagines motor actions to the classifier, adding an option to select trials in which the classifier predicted the user's imagined action not so well to the training set of the classifier could be helpful.
4. Creating more complex classification functions, for example 2 models, one for classifying idle vs not idle and the other to classify different MI tasks. This might help you increase the number of classes you can use while maintaining high accuracy.

Appendix

Our GitHub repository: https://github.com/tomerroditi/bci4als-online

Last year GitHub repository: https://github.com/AssafUni/bci4als-online

EEGNet paper, with python code: https://paperswithcode.com/paper/eegnet-a-compact-convolutional-network-for

References:

Hamedi, M., Salleh, S. H., & Noor, A. M. (2016). Electroencephalographic motor imagery brain connectivity analysis for BCI: a review. *Neural computation*, *28*(6), 999-1041.

Kotsiantis, S. B., Zaharakis, I., & Pintelas, P. (2007). Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, *160*(1), 3-24.

Chaudhary, S., Taran, S., Bajaj, V., & Sengur, A. (2019). Convolutional neural network based approach towards motor imagery tasks EEG signals classification. IEEE Sensors Journal, 19(12), 4494-4500.

Lawhern, V. J., Solon, A. J., Waytowich, N. R., Gordon, S. M., Hung, C. P., & Lance, B. J. (2018). EEGNet: a compact convolutional neural network for EEG-based brain–computer interfaces. *Journal of neural engineering*, *15*(5), 056013.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436-444.

Wang, X., Hersche, M., Tömekce, B., Kaya, B., Magno, M., & Benini, L. (2020, June). An accurate eegnet-based motor-imagery brain–computer interface for low-power edge computing. In *2020 IEEE international symposium on medical measurements and applications (MeMeA)* (pp. 1-6). IEEE.