Member-only story

I Built A Brain Computer Interface To Play Space Invaders Using Thoughts

And I will explain you how, including code examples in Python

Tim de Boer · Follow
Published in Towards Data Science
15 min read · Sep 13, 2022

Listen Share More

Playing Space Invaders with our BCI system and the g.tec Unicorn EEG device. Image by author.

Imagine people moving their arms again after a spinal cord injury left them paralyzed, using their thoughts to control a robotic arm. Imagine people walking again after a stroke, using their thoughts to command an exoskeleton.

With brain-computer interfaces (BCIs), this is possible.

Paralyzed persons are unable to move certain parts of their body, but the brain can still imagine moving these parts through motor imagery (MI). A BCI system collects this brain signal data, and with the help of machine learning (ML), a prediction of the mental task a person is performing can be made.

These BCIs can be used to control external devices such as exoskeletons and robotic arms, giving back functionality to paralyzed persons, and improving their rehabilitation [1].

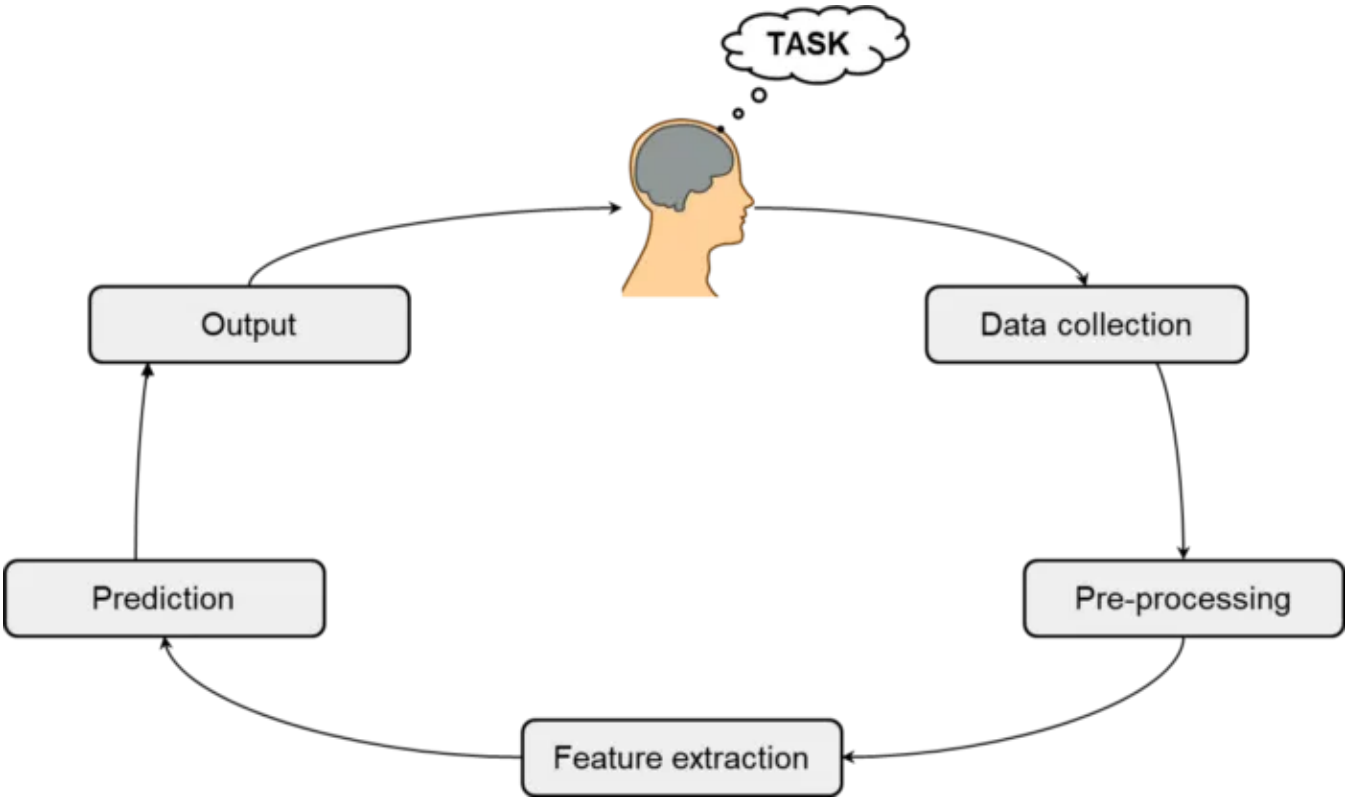
In the last 6 months, I did my master project for the master AI of the VU Amsterdam. I learned to code a program to let someone play Space Invaders using MI with a considerably cheaper device to capture brain signals (the g.tec Unicorn) than often used in research. Being able to play a game of Space Invaders using the Unicorn serves as a first step towards building a cheaper BCI, lowering the threshold to eventually be used during rehabilitation by as much paralyzed persons as possible.

In this post, I'll tell you the main concepts I learned including code examples in Python!

This post is organized as follows:

1. The 6 Building Blocks Of A Brain-Computer Interface In Short: the person, data collection, pre-processing, feature extraction, prediction and output.
2. The Person: motor imagery & how to set up your experiment
3. Data Collection: which device to use & how to stream data to Python
4. Pre-Processing: notch filter, common average referencing & artifact detection
5. Feature Extraction: frequency filtering & spatial filtering
6. Prediction: applying machine learning to get your prediction
7. Output: using the prediction output as input to play Space Invaders

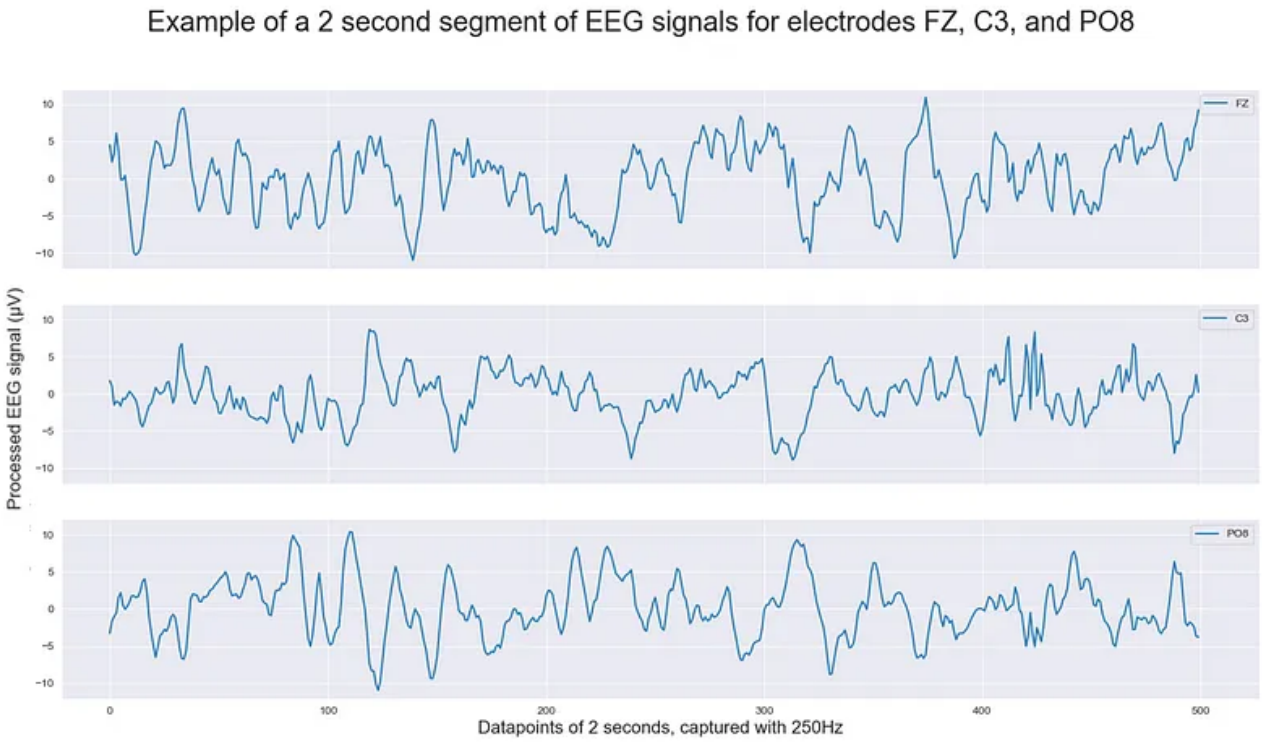
The 6 Building Blocks Of A Brain-Computer Interface



The BCI cycle. Image by author, inspired by [2].

The Person: Of course, a BCI cannot exist without a person using the system. This person performs a certain mental task like motor imagery (MI), resulting in brain activity in certain areas. MI is often used to perform BCI experiments with healthy persons, trying to simulate how paralyzed persons would use the system. Classes to be distinguished could be *left arm MI versus right arm MI* to control a robotic arm, *legs MI versus relax* to start and stop an exoskeleton, or in my case, *left arm MI versus right arm MI versus relax* to play Space Invaders.

Data Collection: When a person is performing a mental task, we need to collect their brain activity data. In this post, we focus on electroencephalography (EEG). The electrodes of the EEG device are placed on top of your scalp, measuring electrical activity originating from brain cells in the area beneath the electrode.



An example of EEG signals from 3 electrodes. Image by author.

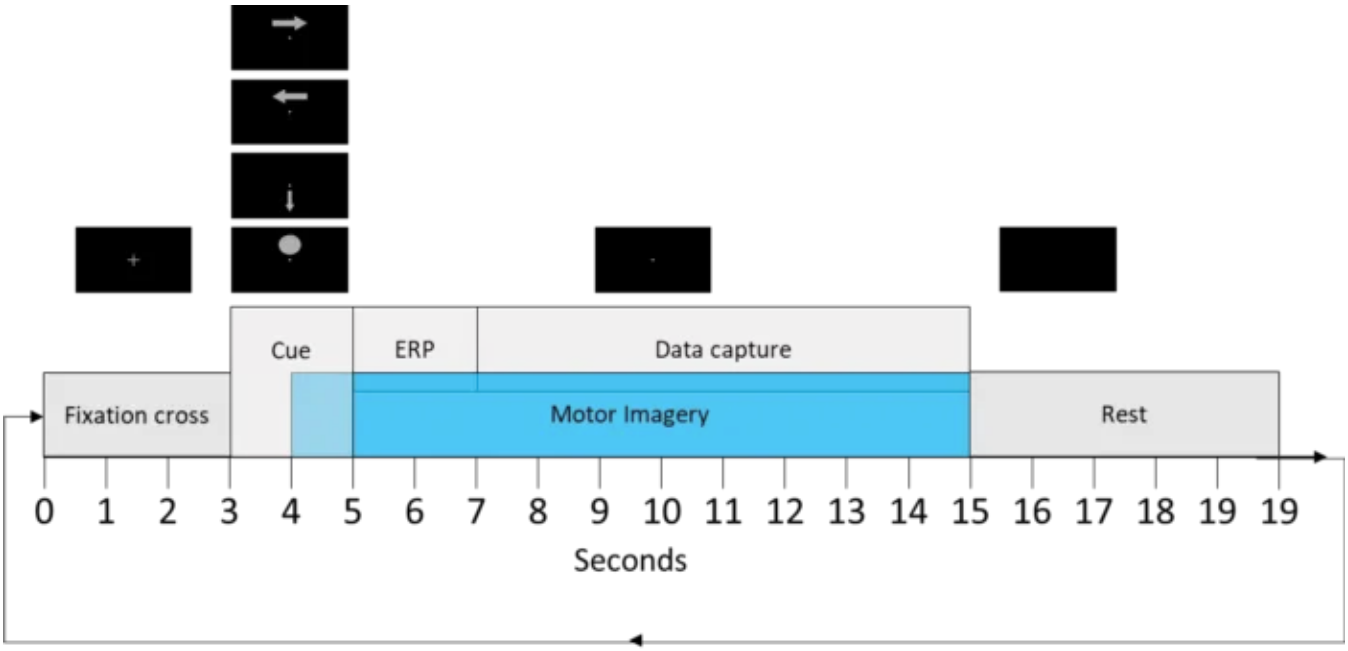
Pre-Processing: After data is collected, pre-processing is needed. Loads of factors influence the EEG signal, and add a lot of noise to the signal. General steps in pre-processing consist of power line noise removal (noise from other electronic devices), artifact detection, and re-referencing of the signal.

Feature Extraction: Brain activity occurs in differing amplitude (the maximum displacement of the signal) and frequency (the number of waves passing by per second). Different frequency bands correspond with different mental states, as brain activity in specific frequency intervals were found to be predominant over the other frequency intervals. To extract features in frequency bands relevant for your tasks, frequency filtering needs to be applied. Next, spatial filtering is applied. As an EEG device is placed on your scalp and thus far away from your brain, and measures your brain activity with a limited amount of electrodes, the relevant brain signals can be spread around several EEG channels. A spatial filter combines information of EEG signals from several channels, trying to recover the original signal by gathering the relevant information from the different channels.

Prediction: After feature extraction, we arrive at predictions. This is were machine learning models come into play, which try to distinguish the data into the classes (the mental states of MI).

Output: For a real-time BCI system, prediction is not the last step! In an online BCI system, the prediction is used to control some external device. It is important to note that this output in turn influences thought and goals of the person, completing the cycle.

The Person: motor imagery & how to set up your experiment



A typical setup for collecting EEG data for motor imagery. Image by author.

Although the future goal is to use BCIs in everyday life, right now they just do not work good enough, and factors which can influence the performance should be limited. Think of distraction. Fatigue. Noise.

A good experiment design tries to avoid this. So, how does a good BCI experiment look like?

Above figure gives an overview of 1 loop of 1 trial in an experiment. We captured data of right arm motor imagery (the right arrow), left arm motor imagery (left arrow), legs motor imagery (bottom arrow), and just relaxing (the circle). Per trial, we had 8 loops. Per experiment, we had 10 trials.

We implemented multiple things to avoid distraction. One thing not visible in the overview above, but very important, is to have an empty table in front of the person, with only the monitor showing the experiment. At the start of each loop, a cross is presented, meant to signal to the person that a new task is coming up, and letting them focus on the screen with distraction.

Performing MI can be hard. Especially when doing it for a prolonged time. To avoid fatigue, we limited the trials to 8 loops (3 minutes) and between each loop we had a 2 minute break.

Noise in EEG data can be caused by numerous factors. In the overview figure, you can see a little block called 'ERP'. This is an abbreviation of 'event related potential', a characteristic number of peaks in your brain activity as a direct result of a sensory, cognitive, or motor event [3]. In our experiments, an ERP could be initiated due to the start of the task (cognitive and motor), and the disappearance of the cue (sensory). An ERP lasts around 1 second. To be sure that we avoid having ERPs in our data, we skipped the first 2 seconds of the motor imagery task.

Another cause of noise is due to muscle movement. Think of blinking your eyes, moving your eyes, clenching your teeth, or swallowing. Therefore, the subject was instructed to blink as little as possible from the moment the cue appeared until the end of the task, and to avoid jaw clenching, sudden eye movements, and swallowing during the task.

Data Collection: which device to use & how to stream data to Python

Various EEG devices are available on the market. To directly come to the point, the single most important feature to look for, is location of the electrodes!

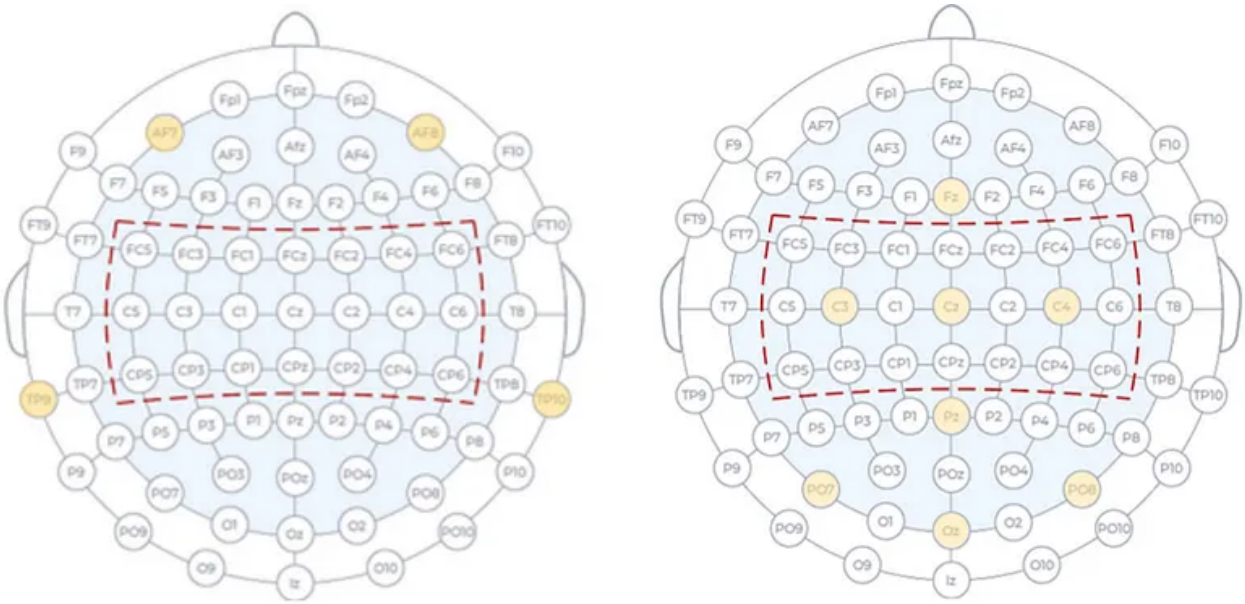
To illustrate this, let me talk about my 2 motor imagery projects: a failed one from last year, and the more successful one I did during my master project.

Last summer, I tried to build my first BCI project for motor imagery prediction. For this project, I used the Muse 2 headset. Little did I know, that by choosing the Muse 2, I made it nearly impossible to succeed in this task, even before I wrote any code!

The area for motor control lays around the middle of the center part of the brain, in the area of the frontal lobe called the motor cortex. To capture brain activity around this area, electrodes need to be placed above this area. The 10–20 international system is a system used for electrode placement to ensure comparable placement across studies, but also to identify which areas of the brain lay are captured by each electrode.

In the figure below, we see the electrode placement of the Muse 2 versus the device I used this year, the g.tec Unicorn. It is clearly visible that the electrodes of the Muse

2 do not cover the motor cortex, while the Unicorn has 3 electrodes in the area! That seems more promising.



The electrode placement of Muse 2 (left) and g.tec Unicorn (right). The red highlighted area contains electrodes placed above the motor cortex. Image by author, inspired by [4].

Streaming the data to Python is actually quite easy using PyLSL, a Python interface

Connect your Unicorn to the Unicorn Suite: We first need to connect the Unicorn device to the Unicorn Suite software. g.tec actually provided some very handy YouTube tutorials regarding this. When successful, let's move on to step 2!

Connect the Unicorn to the LSL interface: In the Unicorn Suite software, go to *DevTools* > *LSL Interface* > *Open*. Then, launch the UnicornLSL executable, and your Unicorn device should appear as available device. Then, simply open the stream, and press start. Now, your Unicorn is streaming data to your laptop. Now we need to write some Python code to collect this data!

Write a Python script: First, we only need to install the pylsl and pandas packages. Then, we write the following small Python script, and we're all set!

```python
from pylsl import StreamInlet, resolve_stream
import pandas as pd

# initialize the streaming layer
finished = False
streams = resolve_stream()
inlet = StreamInlet(streams[0])

# initialize the colomns of your data and your dictionary to capture the data.
columns=['Time','FZ', 'C3', 'CZ', 'C4', 'PZ', 'PO7', 'OZ', 'PO8','AccX','AccY','AccZ',
'Gyro1','Gyro2','Gyro3', 'Battery','Counter','Validation']
data_dict = dict((k, []) for k in columns)

while not finished:
    # get the streamed data. Columns of sample are equal to the columns variable, only the first
    # concatenate timestamp and data in 1 list
    data, timestamp = inlet.pull_sample()
    all_data = [timestamp] + data

    # updating data dictionary with newly transmitted samples
    i = 0
    for key in list(data.keys()):
        data_dict[key].append(all_data[i])
        i = i + 1

    # data is collected at 250 Hz. Let's stop data collection after 60 seconds. Meaning we stop
    if len(data_dict['Time']) >= 250*60:
        finished = True

# lastly, we can save our data to a CSV format.
data_df = pd.DataFrame.from_dict(data_dict)
data_df.to_csv('EEGdata.csv', index = False)
```

unicorn_pylsl.py hosted with ❤ by GitHub view raw

As we are building a real-time BCI, we also need to segment our data into smaller parts; these segments are then our datapoint used as input for our pipelines. Later, when using the system for real-time predictions, we feed the system a 2 second segment.

The observant reader would now notice that our system is not actually real-time, but actually has a delay of 2 seconds. However, real-time sounds way cooler than almost-real-time. So please, forgive me ;)

To make a short side note, if you are unable to collect data yourself, the best online resource to achieve open source EEG data is **MOABB: The Mother of all BCI Benchmarks**. MOABB is an open science project, aiming to build a benchmark of popular BCI algorithms applied on freely available EEG datasets. The code is available on Github, and a documentation is available as well.

Pre-Processing: notch filter, common average referencing & artifact detection

Now we have our data, we can apply pre-processing to clean our data for further steps. This consists of 3 steps: power line noise filtering using a notch filter, common average referencing (CAR), and artifact detection.

EEG picks up noise from all kinds of electronic devices in the room. Therefore, we need to remove the frequency of the power line with a notch filter. In Europe, this is at 50 Hertz, while in USA this is at 60 Hertz. Most EEG devices have a built-in analog power line noise filter, but if not, this is how one can apply a power line noise filter digitally in Python.

Next, artifact detection can be applied. Automatic artifact removal has been implemented based on a statistical analysis of the segments [5]. A segment was considered to contain an artifact when the power, standard deviation, or amplitude of a trial was 2.5 times higher than the mean signal value of all segments, and such segments were rejected. This statistical approach has been adapted in a later study with slight variations to work with real-time experiments by only analyzing the current segment, and not all segments [6]. Segments will be discarded when the absolute amplitude of the segment would exceed 125 μV, or when the kurtosis of a segment would exceed the standard deviation of the segment by four times. The authors have claimed the method to be successful, although no comparison was provided between this method and other artifact removal methods. Due to the simplicity of the method, another benefit when compared to more advanced algorithms is the fact that processing time is short, making it suitable for real-time processing. Processing time of other methods can take more time, making it a possible bottleneck for real-time processing.

Last, CAR could be applied [7]. CAR can be described as an simple spatial noise filter. CAR works by calculating the mean signal value from all EEG channels for each point in time, and then this value will be subtracted from the value of each EEG channel. Here, the goal is to reduce the general noise present in all channels, in

order to improve the often weak specific signal component of the brain region under each individual EEG electrode.

Coding up the pre-processing steps is done as follows. Having our EEG data in a Pandas DataFrame as (timepoints, channels), the code will be:

```
from scipy import signal, stats
import pandas as pd
import numpy as np

for curr_segment in segments:
    # 1. notch filter
    b_notch, a_notch = signal.iirnotch(50, 30, sampling_frequency)
    for column in curr_segment.columns:
        curr_segment.loc[:,column] = signal.filtfilt(b_notch, a_notch, curr_segment.loc[:,colum

    # for next steps data should be in (channels, timepoints format)
    curr_segment = curr_segment.T

    # 2 OUTLIER DETECTION
    for i, j in curr_segment.iterrows():
        if stats.kurtosis(j) > 4*np.std(j) or (abs(j - np.mean(j)) > 125).any():
            if stats.kurtosis(j) > 4*np.std(j):
                print('due to kurtosis')
            outlier +=1

    # 3 APPLY COMMON AVERAGE REFERENCE (CAR)
    if  'deep' in pipeline_type:
        curr_segment -= curr_segment.mean()
```

preprocess_BCI.py hosted with ❤ by GitHub view raw

Feature Extraction: frequency filtering & spatial filtering

To explain why we apply frequency and spatial filtering for feature extraction, let's first explain what happens in the brain during motor imagery.

Motor imagery causes changes in brain activity by so called event-related synchronization (ERS) and event-related desynchronization (ERD). An increase in the power of an EEG signal in a certain frequency band is called an ERS, and a decrease of EEG signal power an ERD [8]. As an example, right arm MI wil lead to a contralateral ERD in the motor cortex (thus, the left motor cortex), and to an ERS just after finishing the MI. For left arm MI, this applies to the right motor cortex.

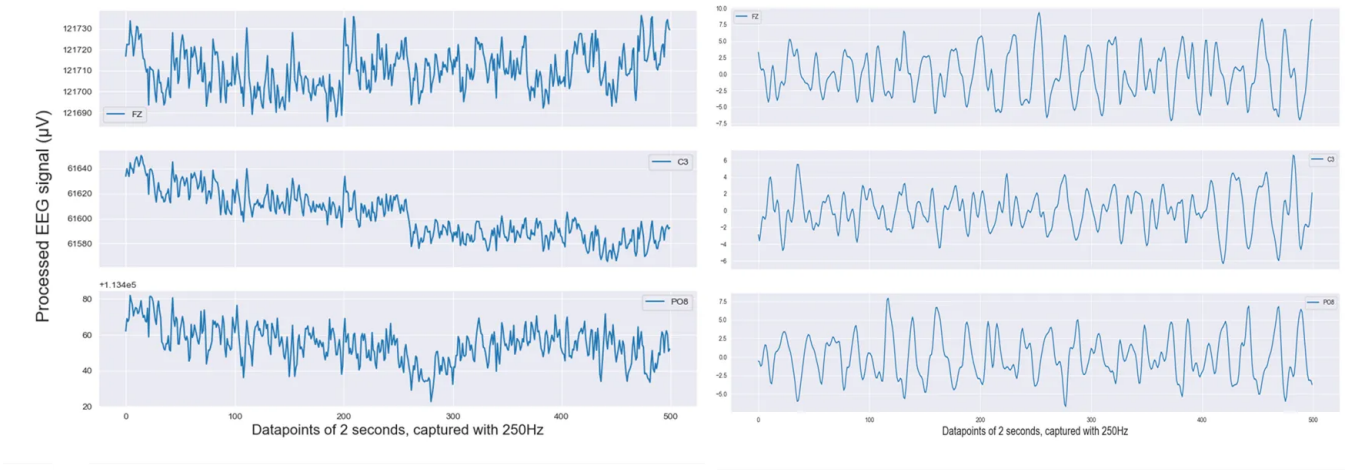
Frequency band filtering gives features that represent the power (energy) of the EEG data for a given frequency band, over a certain time window. Changes in the signal power during MI is most notable around the 8 to 24 Hz frequency band. However, the frequencies with the most notable MI signal differs per subject. In order to broadly capture the relevant EEG signals from MI of all subjects, researchers often design spectral filters for MI-BCIs between 4 and 40 Hz.

Frequency filtering can be done in Python as follows:

```
from scipy import signal
import pandas as pd

# We want to apply a bandpass between 4 and 40 Hz
cutoffs = [4, 40]
# the sample frequency of our device is 250Hz
fs = 250

# we have to device the order of the filter.
# the order determines the sharpness of the filter, i.e., around the # cutoff, do we allow some
# order) or do we want to have a sharp cutoff (high order)? A higher
# order introduces a bigger effect on the signal when initializing,
# thereby destroying the original signal at that part. Note that
# later we apply forward and backward filtering using
# signal.filtfilt, thereby essentially doubling the order we
# determine below.
order = 2

# initialize the filter coefficients
b, a = signal.butter(order, [cuttoff[0]/(fs/2), cuttoff[1]/(fs/2)], "bandpass")

# then apply the filter for each electrode channel:
for column in data.columns:
   data.loc[:,column] = signal.filtfilt(b, a, data.loc[:,column])
```

filtering_BCI.py hosted with ❤ by GitHub view raw

Looking at a segment of 2 seconds, we can clearly see the differences in the signal:



An EEG segment of 2 seconds before (left) and after (right) frequency filtering. Image by author.

To uncover the location of the signal, we use spatial filtering. EEG has poor spatial resolution, and underlying brain signals can be spread around several EEG channels. A spatial filter combines information of EEG signals from several channels, often using weighted linear combinations. Thereby, spatial filtering can recover the original signal by gathering the relevant information which is spread over the different channels. Two commonly used spatial filtering methods are the Common Spatial Patterns filter (CSP) and Riemannian Geometry (RG) [9, 10]. We won't go into depth in these methods in this post, but code will be provided so you can apply them.

Code for spatial filtering will be provided together with the code of the machine learning models, as scikit-learn and additional packages make it very easy to combine both!

Prediction: applying machine learning to get your prediction

After spatial filtering, the next step in the MI-BCI pipeline is classification of the MI state using machine learning (ML).

As data is already heavily processed after applying CSP or RG, often traditional linear ML algorithms are applied, like linear discriminant analysis (LDA), Support Vector Machine (SVM), or a somewhat more complex algorithm like Random Forest (RF).

Before coding it up, let's talk about the data format. Our data is essentially a 3D array, with multiple segments in a (channels, timepoints) format. Scikit-learn likes to work with Numpy arrays. For this purpose, the Numpy function *stack* can come in handy.

> *np.stack: Join a sequence of arrays along a new axis.*

So, by putting our segments (each an np.array of (channels, timepoints) into a list or array, we can use np.stack in order to get our 3D array.

To apply CSP, we need to install the MNE library. Then, we import CSP from mne.decoding.

> *MNE-Python: Open-source Python package for exploring, visualizing, and analyzing human neurophysiological data: MEG, EEG, sEEG, ECoG, NIRS, and more.*

Regarding Riemannian Geometry, a library called PyRiemann has been developed.

> *PyRiemann: a Python machine learning package based on scikit-learn API. It provides a high-level interface for processing and classification of multivariate time series through the Riemannian geometry of symmetric positive definite (SPD) matrices.*

Then, we can use the scikit-learn function *Pipeline* together with the ML models provided by scikit-learn to stitch everything neatly together as follows:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.pipeline import Pipeline
from mne.decoding import CSP
from pyriemann.estimation import Covariances
from pyriemann.tangentspace import TangentSpace

csp = Pipeline(steps=[('csp', CSP()),('lda', LDA())])
csp.fit(X_train, y_train)
preds = csp.predict(X_val)

rg = Pipeline(steps=[('cov', Covariances("oas")),('tg', TangentSpace(metric="riemann")),('rf',
rg.fit(X_train, y_train)
preds = rg.predict(X_val)
```

CSP_and_RG.py hosted with ❤ by GitHub view raw

As you can see, the code essentially boils down to only 3 lines: initialization of the pipeline, fit, and predict. That's quite easy!

Output: using the prediction output as input to play Space Invaders

We did it! We now have a prediction about what kind of motor imagery is performed. The only thing remaining is translating this prediction to a move in the game of Space Invaders. The most easy route I found for this, is to write a Python script which presses certain keys on your keyboard based on the prediction.

For this, we run two Python scripts: one running the game of Space Invaders. I used PyGame for this. The other for capturing data, and for each 2 second segment, apply pre-processing, feature extraction, getting our prediction with ML, and finally pressing the keyboard key corresponding to the prediction. Left arm motor imagery would let the player move to the left, right arm motor imagery to the right, and relaxing would shoot bullets!

```
from pynput.keyboard import Key, Controller
keyboard_game = Controller()

while playing_game:
  # collect data in segments of 2 seconds (using PyLSL)
  # pre_process (notch, artifact detection, CAR)
  # feature_extraction (frequency and spatial filtering),
  # get_prediction (from trained model)
  # 0 is relax, 1 is right arm motor imagery, 2 is left arm motor imagery.

if prediction[0] == 0:
    key = Key.up
    keyboard_game.press(key)
if prediction[0] == 1:
    key = Key.right
    keyboard_game.press(key)
elif prediction[0] == 2:
    key = Key.left
    keyboard_game.press(key)
```

outputBCI.py hosted with ❤ by GitHub view raw

Conclusion

In this post, we went over all the steps needed to build a well performing BCI to play Space Invaders. We went over how one can set up an BCI experiment, how to collect data using the g.tec Unicorn, how to pre-process this data, extract features, and make predictions. Last, we looked at how all this could be combined with the game of Space Invaders.

To conclude this post, I would like to address the fact that the discussed BCI systems need to be developed with great care and safety protocols. While BCIs can have great

positive implications for motor-impaired persons to regain autonomy through exoskeleton control, system failure and errors could endanger the life of its users of these systems in specific contexts. For example, an EEG-based exoskeleton failing as a person is crossing a street could cause an accident, and unintended movements of an exoskeleton could cause injuries. Implementing such BCI systems in daily life also raises data collection concerns, as data would be collected during a full spectrum of daily activities, raising privacy issues.

Playing Space Invaders with our BCI system and the g.tec Unicorn EEG device. Image by author.

For more details about each step discussed in this post, the reader is referred to this publication containing separate posts for each step.

Various snippets of code came by, with the intention of letting you understand the theory and code step by step. If you want to dive deeper into the code, the full repository of my project is available on GitHub here. In the repository, one can also see that deep learning with transfer learning has been applied. To keep this post limited in size, I won't dive into that topic. However, if you are interested in how I applied deep transfer learning, I kindly refer you to my other post explaining convolutional neural networks for BCIs.

Thank you for reading, and good luck with your journey into the field of brain-computer interfaces!

References

[1] Jingyi Liu, Muhammad Abd-El-Barr, and John H Chi. Long-term training with a brain-machine interface-based gait protocol induces partial neurological recovery in paraplegic patients. Neurosurgery, 79(6):N13–N14, 2016.

[2] Marcel Van Gerven, Jason Farquhar, Rebecca Schaefer, Rutger Vlek, Jeroen Geuze, Anton Nijholt, Nick Ramsey, Pim Haselager, Louis Vuurpijl, Stan Gielen, et al. The brain–computer interface cycle. Journal of neural engineering, 6(4):041001, 2009.

[3] Steven J Luck. An introduction to the event-related potential technique. MIT press, 2014.

[4] Piotr Szczuko. Real and imaginary motion classification based on rough set analysis of eeg signals for multimedia applications. Multimedia Tools and Applications, 76(24):25697–25711, 2017.

[5] Eduardo López-Larraz, Fernando Trincado-Alonso, Vijaykumar Rajasekaran, Soraya Pérez-Nombela, Antonio J Del-Ama, Joan Aranda, Javier Minguez, Angel Gil-Agudo, and Luis Montesano. Control of an ambulatory exoskeleton with a brain–machine interface for spinal cord injury gait rehabilitation. Frontiers in neuroscience, 10:359, 2016.

[6] Andreas Schwarz, Patrick Ofner, Joana Pereira, Andreea Ioana Sburlea, and Gernot R Müller-Putz. Decoding natural reach-and-grasp actions from human eeg. Journal of neural engineering, 15(1):016005, 2017.

[7] Kip A Ludwig, Rachel M Miriani, Nicholas B Langhals, Michael D Joseph, David J Anderson, and Daryl R Kipke. Using a common average reference to improve cortical neuron recordings from microelectrode arrays. Journal of neurophysiology, 101(3):1679–1689, 2009.

[8] Gert Pfurtscheller and FH Lopes Da Silva. Event-related eeg/meg synchronization and desynchronization: basic principles. Clinical neurophysiology, 110(11):1842–1857, 1999.

[9] Kai Keng Ang, Zheng Yang Chin, Chuanchu Wang, Cuntai Guan, and Haihong Zhang. Filter bank common spatial pattern algorithm on bci competition iv datasets 2a and 2b. Frontiers in neuroscience, 6:39, 2012.

[10] Alexandre Barachant, Stéphane Bonnet, Marco Congedo, and Christian Jutten. Multiclass brain–computer interface classification by riemannian geometry. IEEE Transactions on Biomedical Engineering, 59(4):920–928, 2012.

Neurotechnology | Brain Computer Interface | Machine Learning | Artificial Intelligence

Follow

Written by Tim de Boer

314 Followers · Writer for Towards Data Science

Master graduate AI @VU Amsterdam. Currently learning and writing about building brain-computer interfaces. Support me: https://timdb877.medium.com/membership

More from Tim de Boer and Towards Data Science

Tim de Boer in Building a bedroom BCI

Collecting Brain Signal Data Using The Muse 2 EEG Headset

A first journey in DIY brain computer interfaces, part 2

· 9 min read · Aug 30, 2021

39

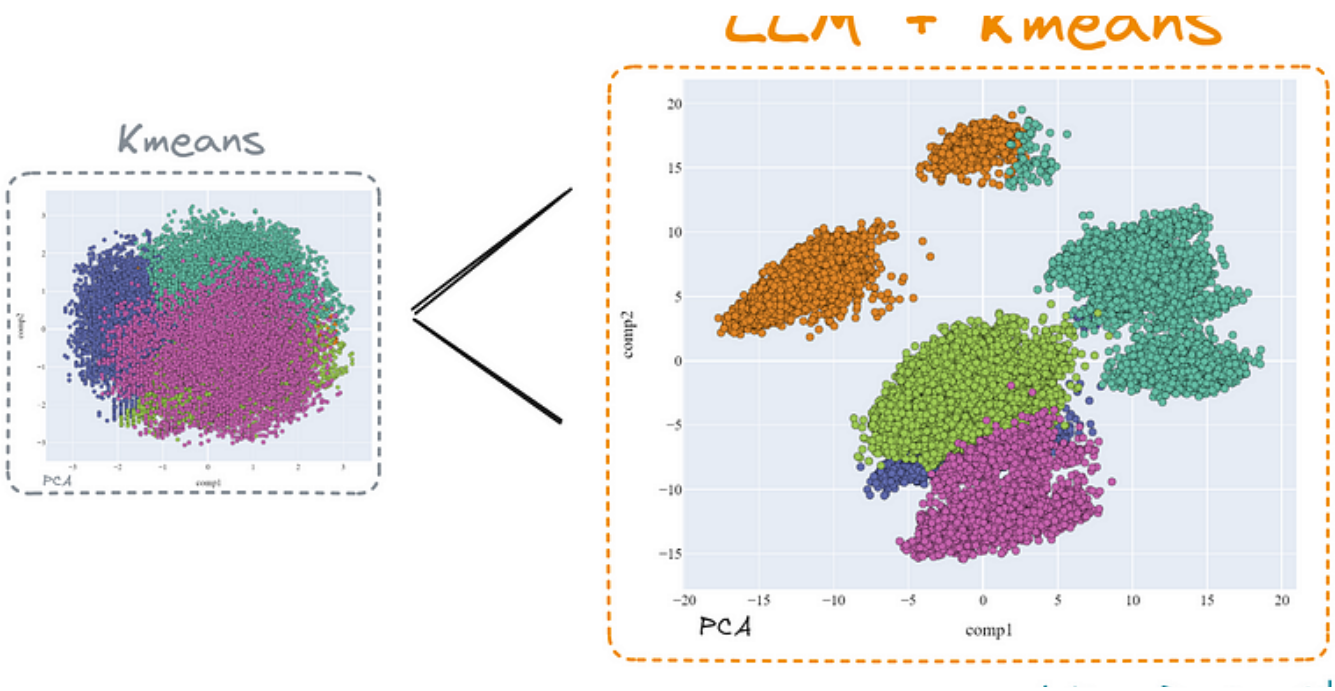
Giuseppe Scalamogna in Towards Data Science

New ChatGPT Prompt Engineering Technique: Program Simulation

A potentially novel technique for turning a ChatGPT prompt into a mini-app.

9 min read · Sep 3

1.6K 16



Damian Gil in Towards Data Science

Mastering Customer Segmentation with LLM

Unlock advanced customer segmentation techniques using LLMs, and improve your clustering models with advanced techniques

23 min read · 5 days ago

1.6K 17

Tim de Boer in Building a bedroom BCI

Why, What and How to Start Your Own Brain Computer Interface Project

A first journey in DIY brain computer interfaces, part 1

· 10 min read · Aug 27, 2021

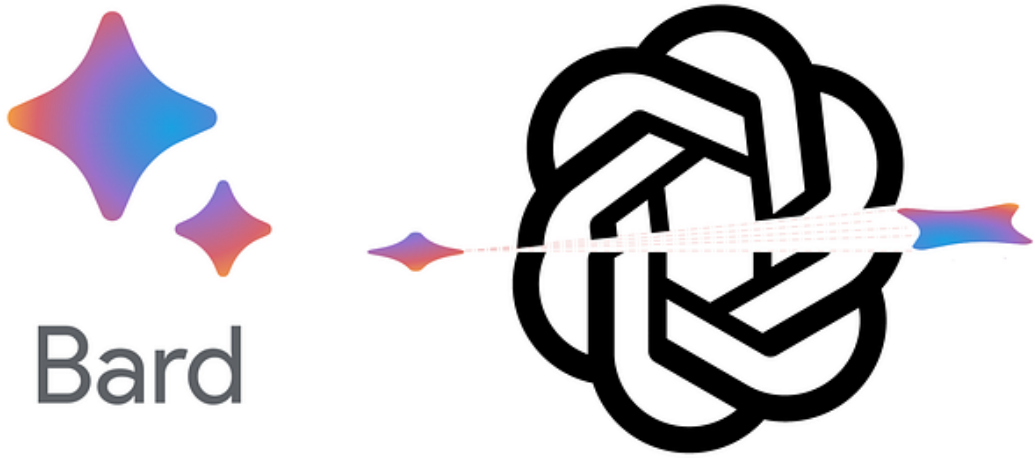
Unbecoming

10 Seconds That Ended My 20 Year Marriage

It's August in Northern Virginia, hot and humid. I still haven't showered from my morning trail run. I'm wearing my stay-at-home mom...

· 4 min read · Feb 16, 2022

65K 949



AL Anany

The ChatGPT Hype Is Over—Now Watch How Google Will Kill ChatGPT.

It never happens instantly. The business game is longer than you know.

✦ · 6 min read · Sep 1

10.6K 343

Lists

AI Regulation

6 stories · 137 saves

ChatGPT prompts

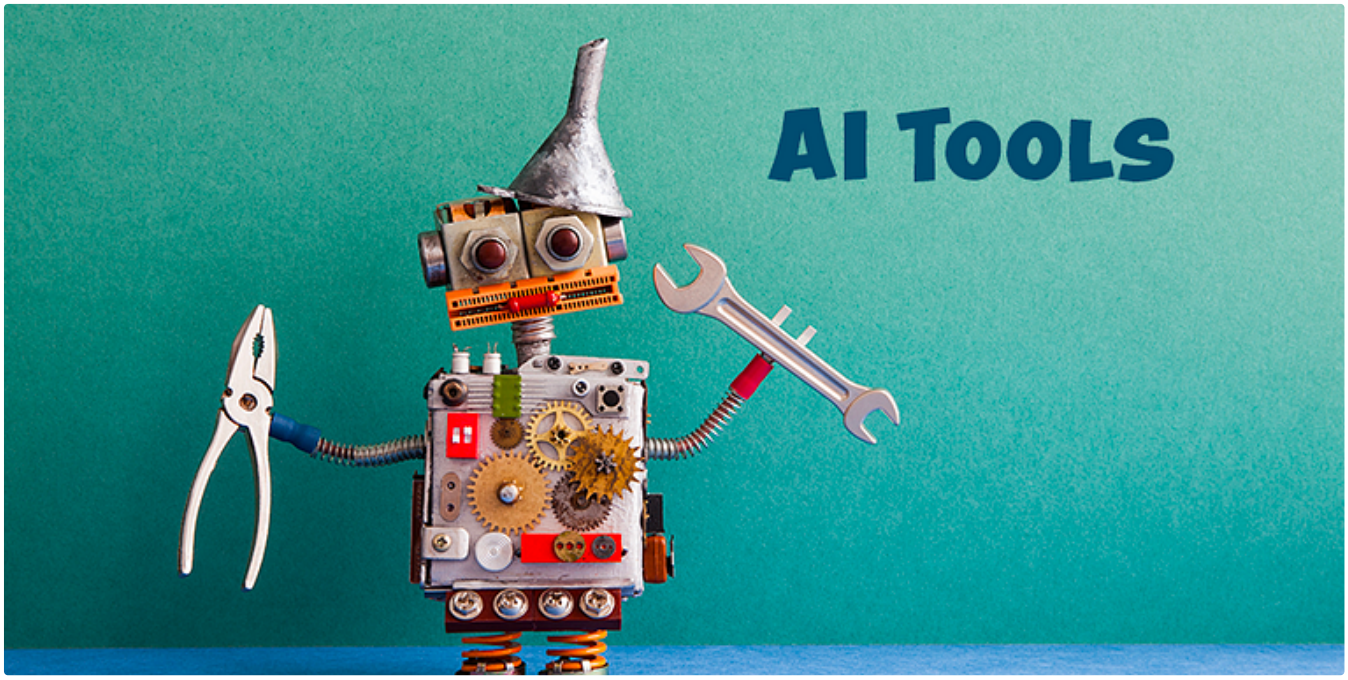
24 stories · 448 saves

ChatGPT

21 stories · 177 saves

Generative AI Recommended Reading

52 stories · 271 saves



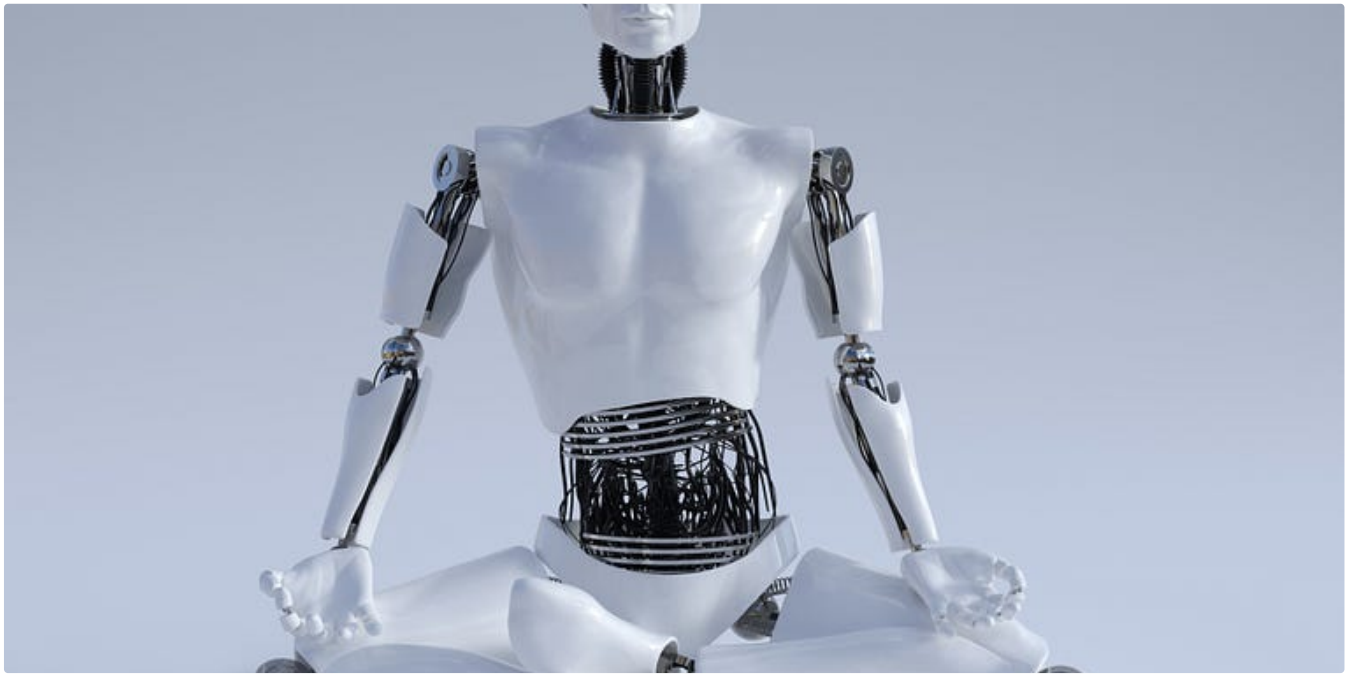
Digital Giraffes

7 Awesome and Free AI Tools You Should Know

We collected 7 free artificial intelligence(AI) tools, most of them easy to use and some more sophisticated... like building ML models.

5 min read · Nov 17, 2022

8.7K 196



The PyCoach in Artificial Corner

You're Using ChatGPT Wrong! Here's How to Be Ahead of 99% of ChatGPT Users

Master ChatGPT by learning prompt engineering.

· 7 min read · Mar 17

31K 559

Maximilian Vogel in MLearning.ai

The ChatGPT list of lists: A collection of 3000+ prompts, examples, use-cases, tools, APIs...

Updated Sep-09, 2023. Added new prompt engineering courses, masterclasses and tutorials.

10 min read · Feb 7

8.7K 114

Henry Tinker

10 Books From My English Degree I Wish More People Knew About

When considering which books to read in 2022, I started thinking about all the books I'd loved from my English Literature degree. I...

5 min read · Feb 22, 2022

8.4K 175