Zubarev, Ivan; Vranou, Gavriela; Parkkonen, Lauri

MNEflow: Neural networks for EEG/MEG decoding and interpretation

Published in:
SoftwareX

DOI:
10.1016/j.softx.2021.100951

Published: 01/01/2022

Document Version
Publisher's PDF, also known as Version of record

SoftwareX 17 (2022) 100951

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

MNEflow: Neural networks for EEG/MEG decoding and interpretation

Ivan Zubarev [a,*], Gavriela Vranou [a], Lauri Parkkonen [a,b]

[a] Department of Neuroscience and Biomedical Engineering, Aalto University School of Science, FI-00076, Aalto, Finland
[b] Aalto Neuroimaging Center, Finland

ARTICLE INFO

Article history:
Received 21 December 2020
Received in revised form 17 November 2021
Accepted 14 December 2021

Keywords:
Neural networks
Electroencephalography
Magnetoencephalography
Tensorflow
Machine learning

ABSTRACT

MNEflow is a Python package for applying deep neural networks to multichannel electroencephalograpic (EEG) and magnetoencephalographic (MEG) measurements. This software comprises Tensorflow-based implementations of several popular convolutional neural network (CNN) models for EEG–MEG data and introduces a flexible pipeline enabling easy application of the most common preprocessing, validation, and model interpretation approaches. The software aims to save time and computational resources required for applying neural networks to the analysis of EEG and MEG data.

Code metadata

Current code version	0.3.1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-20-00108
Code Ocean compute capsule	7934771
Legal Code License	BSD-3
Code versioning system used	git
Software code languages, tools, and services used	Python
Compilation requirements, operating environments & dependencies	Python >= 3.6 Tensorflow >= 2.1.0 MNE-Python >= 0.19 NumPy, SciPy, MatPlotLib
If available Link to developer documentation/manual	https://mneflow.readthedocs.io
Support email for questions	ivan.zubarev@aalto.fi

1. Motivation and significance

Deep neural networks are becoming increasingly popular in the analysis of the measurements of electromagnetic brain activity [1]. While outperforming traditional methods in many other domains, their success in decoding brain signals has been limited mainly due to small dataset sizes and high dimensionality of the measurements. With advances in data-sharing initiatives, however, large standardized neuroimaging datasets are becoming increasingly available. Yet, besides the data, a successful brain-decoding study also requires a robust and reproducible machine-learning workflow that would take into account the specificity of the measurement technique, experimental design and other method-specific factors. In other words, these studies require combined expertise from machine learning and neuroimaging.

The motivation behind MNEflow is to provide neuroscientists with a robust, reproducible, and time-efficient tool for applying neural networks to large EEG and MEG datasets efficiently. Implementing a (deep) convolutional neural network (CNN) involves numerous design choices such as selecting the model architecture, objective function, and evaluation approach. Unless the user is well-informed in the machine-learning domain, exploring all possible options requires a considerable investment of time. Moreover, as many neuroimaging studies seek to discover new knowledge, it is often desirable to gain insights in patterns that a machine-learning model extracts from the data. Interpretation of patterns that neural networks learn to extract from data, however, is a non-trivial task and remains an area of active research [2–4].

* Corresponding author.
E-mail address: ivan.zubarev@aalto.fi (Ivan Zubarev).

https://doi.org/10.1016/j.softx.2021.100951

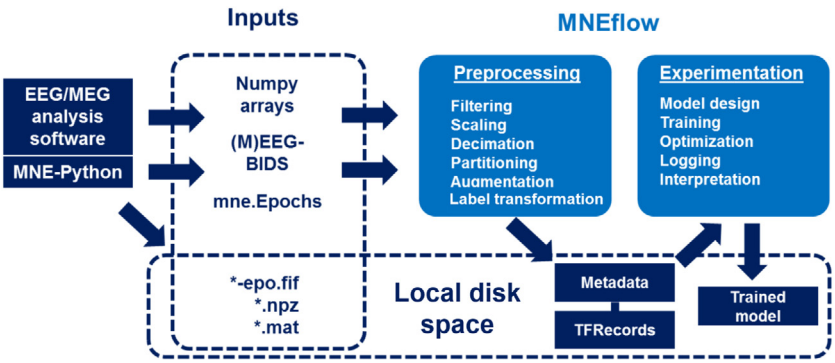

Fig. 1. MNEflow architecture and workflow.

MNEflow aims at alleviating these problems by providing a general, streamlined workflow for model development, optimization, and interpretation. The proposed workflow is designed for optimal memory usage and minimal repetition of operations, which is achieved by storing intermediate results, trained models, and training logs on a PC storage device, and allowing to access, reproduce and inspect the results.

MNEflow allows e.g. running different models on the exactly same training, validation and test sets enabling their fair benchmarking. Furthermore, the modular structure of the software allows for an easy implementation of new models. Thus, we hope that MNEflow also contributes to the reproducibility of science.

Recent review identified low reproducibility of EEG studies applying deep-learning techniques as one of the core challenges of the field [1]. Among the key contributing factors are unavailability of data, code, and overall low transparency in describing the experimental pipelines.

2. Software description

Machine learning methods are usually applied to probe if information encoded by an experimental design can be decoded from measurements of the brain activity. The goal of the experiment is often to discriminate between a set of discrete conditions, leading to a classification problem. Alternatively, the goal could be to predict the value of a continuous variable, leading to a regression problem. Once a model is trained, it is often desirable to be able to explore the patterns that allow the model to make successful predictions [4–7].

MNEflow provides utilities allowing to streamline processing of EEG/MEG data by solving classification or regression problems using a community-supported and expanding pool of implementations of (deep) neural networks and several domain-specific utilities for preprocessing, evaluation, and interpretation of the findings.

2.1. Software architecture

The functionality of MNEflow can be divided into two major blocks:

- *Preprocessing* includes various manipulations aiming to prepare EEG/MEG data for the neural network, including scaling, resampling, partitioning into training/validation/test sets, and augmentation of the measurement data as well transformations (e.g. log-transform) of the target variables.
- *Experimentation* includes selecting an architecture of the neural network, adjusting hyperparameters, training, and assessing performance. Since this process typically requires several iterations, MNEflow also keeps record of the training runs, facilitating parameter optimization.

EEG/MEG measurements typically comprise large amounts of multidimensional data and require pre-processing which can be very time-consuming. MNEflow makes use of the local file system to store the processed measurement data using the Tensorflow serialized record (TFRecord) format [8] as well as associated metadata. Storing intermediate results avoids unnecessary re-running of the preprocessing pipeline, optimizes memory usage, and speeds up model training at the expense of using disk space. Additional benefits of this approach stem from the fact that comparing different models can be done using the exact same partitions of the dataset. A trained model is also saved to a file and can be readily tested with a different data set, applied in real-time decoding of on-going MEG/EEG, or used for investigating the patterns of the brain activity informing the model performance.

2.2. Software functionality

Import. MNEflow supports several input data formats. EEG/MEG data can be imported directly from MNE-Python [9] by providing the `mne.Epochs` object as an argument to MNEflow. When using other EEG/MEG signal-processing software, the data can be provided as a 3d-array with the structure [trials, sensors, time points]. Alternatively, one can provide paths to data files stored in any of the supported formats indicated in Fig. 1.

Preprocessing. MNEflow implements several basic preprocessing functions that can be applied to the data when producing TFRecord datasets. These include filtering, scaling, selecting a subset of channels, and resampling. While all these utilities are already implemented in MNE-Python, this step is done so that the most basic preprocessing techniques can be applied also to the data exported from other software packages as well.

Preprocessing utilities specific to machine-learning include partitioning the dataset into training/validation/test sets, splitting the data into smaller and possibly overlapping segments (augmentation), producing sequences for RNN-type models, and manipulating data with target variables.

Part of these preprocessing functions can be also applied once the TFRecord files have been already produced. This is done in order to minimize time and disk space in case user wants to try a different preprocessing approach.

Model development. MNEflow implements several published models that were shown to perform efficiently on EEG and MEG data [4–6]. These models inherit the same parent class (mneflow.BaseModel) and thus differ only in terms of their computational graphs and hyperparameter specifications. Such structure ensures that the same optimizers, datasets, and validation routines can be used interchangeably on different models. Similarly, users can easily specify their own models by e.g. only designing a custom computational graph, while re-using other components. This saves time when experimenting with new model designs.

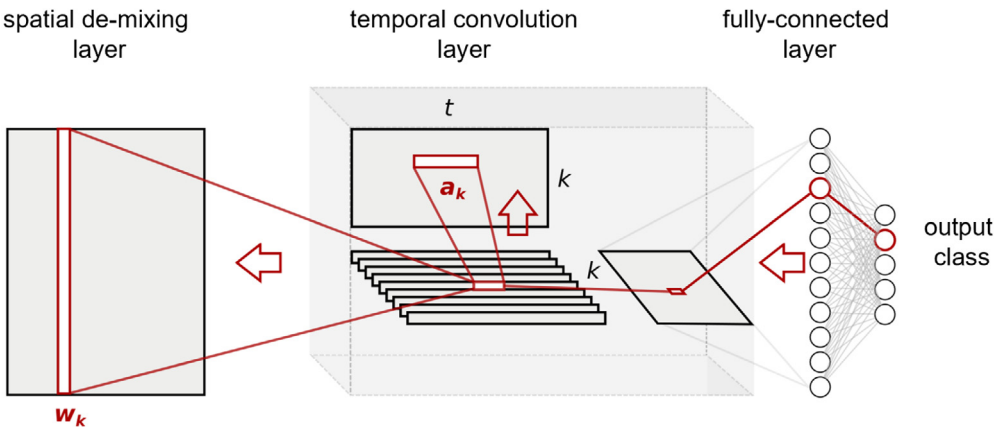

Fig. 2. Illustration of the recursive elimination approach. Weights of each node of the output layer (red circle) is set to zero, effectively disabling the influence of the corresponding spatial $\mathbf{w}_k$ and temporal $\mathbf{a}_k$ filters comprising the kth latent component on the classification performance. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Training and evaluation. Optimization and training consist of specifying an objective function, performance metric and optimization algorithm. The current version of MNEflow (0.3.1) requires Tensorflow version 2.1.0 or newer and relies on the Keras API. The optimizer used by all MNEflow models is Adam [10] with categorical cross-entropy [11] as the default objective function for classification tasks, and mean-squared error (MSE) [12] - for regression tasks. Early stopping is used as the default model training approach. Different optimizers, objective functions and performance metrics can be specified by providing additional arguments to `model.build()`.

Model inspection. MNEflow aims at providing users with a set of tools to inspect the patterns that the model learns from the data. These methods, although potentially very informative, are highly model-specific and still need to be studied and validated. At present, model interpretation is only available for LF-CNN [4] and includes several heuristic approaches to identify spatial patterns, waveforms and frequency properties of the signal components contributing to the one-dimensional target variable (e.g. class in a classification problem). LF-CNN introduces a conditional independence assumption on the latent components learned from the data. Thus, features are grouped into a small number of conditionally-independent spatial–temporal latent factors allowing a researcher to inspect their spatio-temporal properties. To date, three general approaches for identifying relevant features are implemented:

- Recursive elimination. Nodes of the network corresponding to each latent component are switched off by setting their corresponding weights to zero. Performance of the network with $n - 1$ latent components is then evaluated on the validation set [13]. Feature relevance is ranked based on the effect that their removal has on the value of the loss function. This approach is illustrated by Fig. 2.
- Correlation with the target variable, computed for each feature individually. For regression problems, feature relevance is ranked by its (absolute) Spearman correlation coefficient with the target variable. For classification problems, categorical cross-entropy [11] is used as the distance metric.
- Weight-based contributions. Feature relevance is ranked by the value of the corresponding weights. Example of component selection based on the $l2$ norm of its weights is illustrated in Fig. 3.

3. Illustrative examples

Generating TFRecords and metadata. If MNE-Python is used to import and process measurement data, the user just has to provide an `mne.Epochs` object (or a list of `mne.Epochs` objects) to the `produce_tfrecords` utility, specifying the data id and path to save the serialized TFRecord files. Optionally, additional import and preprocessing parameters can be specified. Alternatively, one can use `produe_tfrecords` which takes path(s) to files in other compatible formats that were save to disk. When importing datasets saved in the `*.mat` or `*.npz` format, an additional keyword argument `array_keys` may be required to identify the measurement data and the target variables. Finally, in case of more complex preprocessing pipelines, one can feed data and labels as a tuple of `numpy` arrays.

```
# Read MEG/EEG epochs from a file
import mne
import mneflow

epochs = mne.read_epochs(filename)

# Specify import options
import_opt = dict(savepath='C:\\data\\tfr\\',
                  out_name='mne_sample_epochs',
                  fs=600,
                  input_type='trials',
                  target_type='int',
                  n_folds=5,
                  scale=True,
                  test_set='holdout')

# Process inputs and write TFRecord files and metadata
    file to disk
meta = mneflow.produce_tfrecords(epochs, **import_opt)
```

Listing 1: Data importing and preprocessing.

Calling `mneflow.produce_tfrecords` returns a metadata file that is saved to disk along with the TFRecord files. If the metadata file already exists at the specified path and `data_id`, MNEflow will load the existing TFRecords unless the `overwrite` option is specified. This is done to avoid re-running time-consuming `produce_tfrecords` multiple times for the same dataset.

Initializing the dataset object. The metadata file is then used to initialize the `Dataset` object. This object includes several methods that allow experimenting with the dataset without the need to repeat the preprocessing or overwriting the TFRecord files each time. For example, one can train a classifier model using any subset of classes, channels or reduce the sampling rate by decimating in the time domain.

Computational graph. MNEFlow implements several published neural network architectures that have been designed for EEG/MEG decoding. In the most basic case, the user can pick one of these models from `mneflow.models`. Since all models inherit from the same `mneflow.BaseModel` parent class, switching from one model to another only requires changing the computational graph accordingly. Furthermore, designing custom architectures in MNEflow can be done easily, as it only requires specifying a new computational graph. This can be done by overriding the default computational graph in the parent class (see the advanced examples in Documentation for details). The regularization parameters may include l1 and l2 penalty on the trainable weights, which can be applied separately to different layers (e.g. convolution kernels, dense-layer weights etc.).

Once all the parameters are specified, the model is compiled and can be trained.

```
# Initialize the dataset object
dataset = mneflow.Dataset(meta, train_batch=100)

# Specify parameters for LF-CNN
lf_params = dict(n_latent=32, #  Number of latent factors
                 filter_length=17, #  Convolutional filter length in time samples
                 nonlin=tf.nn.relu,
                 padding='SAME',
                 pooling=5,
                 stride=5,
                 pool_type='max',
                 model_path=import_opt['savepath'],
                 dropout=0.5,
                 l1_scope=["weights"],
                 l1=3e-4)

# Build and train the model
model = mneflow.models.LFCNN(dataset, lf_params)
model.build()
model.train(n_epochs=10, eval_step=100, early_stopping=3)
```

Listing 2: Building and training the model.

Investigating model parameters. Because methods implemented in MNEflow belong to discriminative (as opposed to generative) learning techniques, their primary goal is to approximate a set of conditional distributions of target variables given the measurement. Thus, generally speaking, there is no reason to believe that patterns that these models extract from the data to make their predictions can be used to adequately describe the underlying data-generating (in our case — neurophysiological) process. In some cases, however, inspecting these parameters can lead to useful insights, particularly when used in combination with standard encoding approaches [14]. Alternatively, feature interpretation can be useful to e.g. estimate the contribution of physiological artifacts in model performance, or as a tool for a quick explorative analysis when designing new experimental protocols.

In either case, however, interpreting features that discriminative models use to produce their predictions is always dependent on the quality of these predictions and only produces useful information if model performance is close to optimal.

MNEflow implements basic interpretation tools for some of the implemented models. These may include spatial patterns, latency, or frequency content of the activity informing the model. In this example, we will use LF-CNN [4], an interpretable neural network following the generative model of an EEG/MEG signals, to explore patterns of the brain activity providing the greatest contribution to each class of stimuli.

```
# Compute and show the informative patterns
model.compute_patterns()
f1 = model.plot_patterns(sensor_layout='Vectorview-grad',
                         sorting='l2',
                         scale=True)
f2 = model.spectra(sorting='l2')
```

Listing 3: Inspecting informative patterns.

4. Impact

MNEflow aims to provide researchers with a tool for reproducible, time-efficient and streamlined application of neural networks for decoding brain states from EEG/MEG measurements. This goal is achieved by providing a general workflow that minimizes the risk of falling into the most common pitfalls and thus, ultimately, optimizing time spent doing meaningful research.

MNEflow implements a number of domain-specific utilities to streamline the interface between standard M/EEG processing and machine-learning pipelines. These include various preprocessing, scaling, and data augmentation approaches, efficient ways to log training iterations as well as utilities for typical cross-validation procedures such as leave-one-subject-out etc.

Using an efficient intermediate data storage format avoids repeating the same time-consuming preprocessing operations and allows running the analysis on massive open datasets that begin to emerge in the field.

MNEflow implements a constantly-extended pool of popular neural network models that can be easily applied to classification and regression tasks as well as used as feature extraction methods for long-term sequence modeling with Recurrent Neural Networks within a single API [4–6].

Apart from using the implemented models, designing custom models in MNEflow can also be performed efficiently. The user can only specify the computational graph while making use of the existing data handling, optimization, performance measuring and model inspection pipelines. The pool of implemented models can then be conveniently used as benchmarks to compare to.

Because knowledge discovery is of particular interest in decoding brain data, MNEflow implements several methods for it, allowing convenient exploration of spatial and temporal patterns that inform the model.

5. Conclusions

Taken together, we believe that MNEflow will contribute to promoting the use of (deep) neural networks as a general research and knowledge discovery tool in imaging neuroscience and specifically in brain–computer interfacing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to thank Alexandre Gramfort for his valuable comments on the development process and implementation. The research has been supported by European Research Council, grant No. 678578 to L.P.

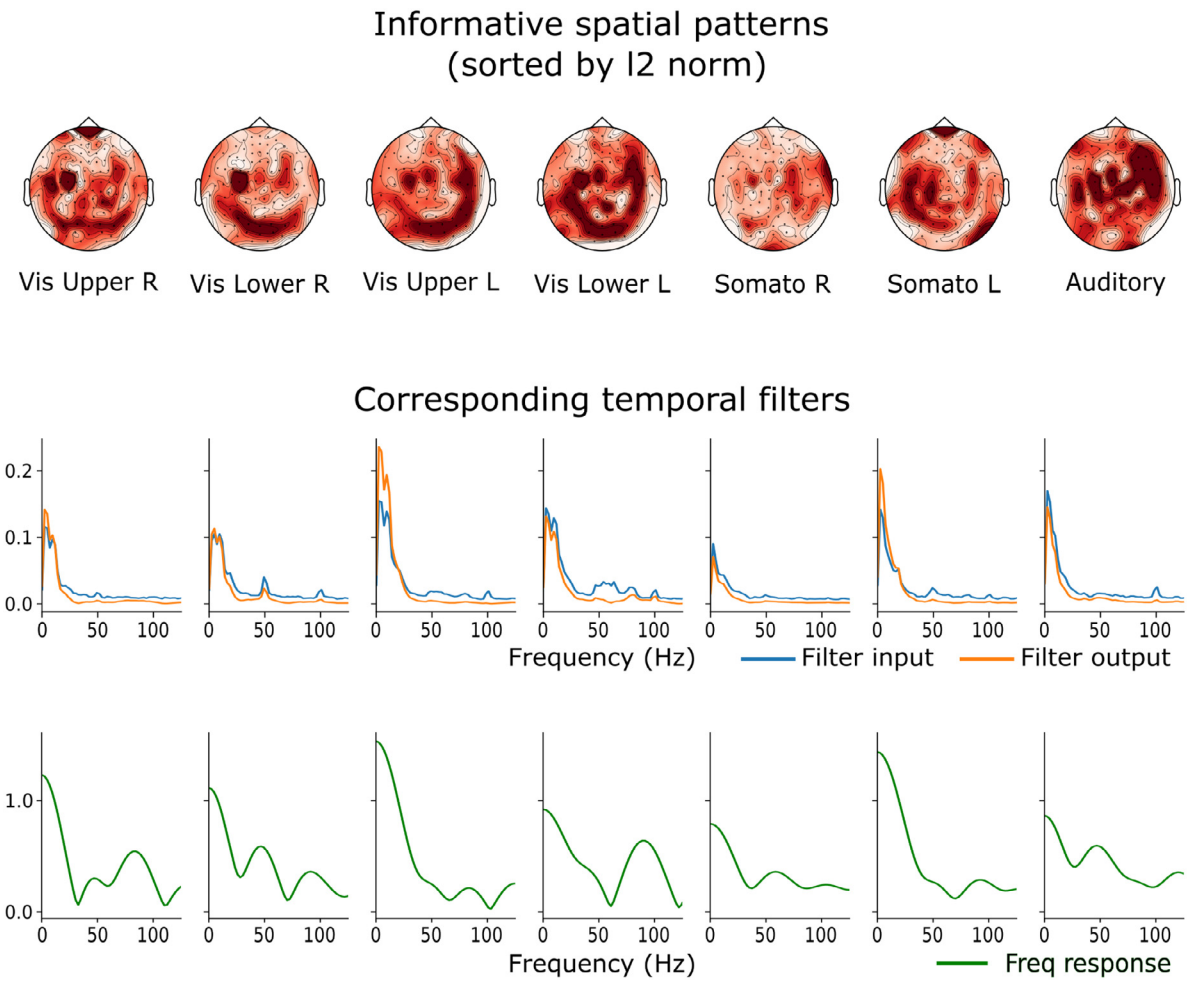

Fig. 3. Example of model inspection output for the 'multimodal' example dataset from MNE-Python. Informative features are grouped into spatio-temporal components allowing to visualize their topographies (top row) and spectral properties (middle and bottom rows). In this example, components having maximum l2-norm of their spatial and temporal weight vectors are selected for display for each class separately.

References

[1] Roy Y, Banville H, Albuquerque I, Gramfort A, Falk TH, Faubert J. Deep learning-based electroencephalography analysis: A systematic review. J Neural Eng 2019;16(5). http://dx.doi.org/10.1088/1741-2552/ab260c.

[2] Haufe S, Meinecke F, Görgen K, Dähne S, Haynes J-D, Blankertz B, et al. On the interpretation of weight vectors of linear models in multivariate neuroimaging. NeuroImage 2014;87:96–110. http://dx.doi.org/10.1016/j.neuroimage.2013.10.067.

[3] Kindermans PJ, Schütt KT, Alber M, Müller KR, Erhan D, Kim B, et al. Learning how to explain neural networks: Patternnet and patternattribution. In: 6th international conference on learning representations, ICLR 2018 - conference track proceedings. 2018, URL https://arxiv.org/pdf/1705.05598.pdf.

[4] Zubarev I, Zetter R, Halme HL, Parkkonen L. Adaptive neural network classifier for decoding MEG signals. NeuroImage 2019;197:425–34. http://dx.doi.org/10.1016/j.neuroimage.2019.04.068, http://www.ncbi.nlm.nih.gov/pubmed/31059799. http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC6609925.

[5] Schirrmeister RT, Springenberg JT, Fiederer LDJ, Glasstetter M, Eggensperger K, Tangermann M, et al. Deep learning with convolutional neural networks for EEG decoding and visualization. Hum Brain Mapp 2017;38(11):5391–420. http://dx.doi.org/10.1002/hbm.23730, URL http://doi.wiley.com/10.1002/hbm.23730.

[6] Lawhern VJ, Solon AJ, Waytowich NR, Gordon SM, Hung CP, Lance BJ. EEGNet: a compact convolutional neural network for EEG-based brain–computer interfaces. J Neural Eng 2018;15(5):056013. http://dx.doi.org/10.1088/1741-2552/aace8c, URL http://stacks.iop.org/1741-2552/15/i=5/a=056013?key=crossref.926d8be8d11477a9c1ce0a164d5c869a.

[7] Giovannetti A, Susi G, Casti P, Mencattini A, Pusil S, López ME, et al. Deep-MEG: spatiotemporal CNN features and multiband ensemble classification for predicting the early signs of alzheimer's disease with magnetoencephalography. Neural Comput Appl 2021;33(21):14651–67. http://dx.doi.org/10.1007/S00521-021-06105-4/TABLES/4, URL https://link.springer.com/article/10.1007/s00521-021-06105-4.

[8] Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, et al. TensorFlow: LArge-scale machine learning on heterogeneous distributed systems. 2016, URL http://arxiv.org/abs/1603.04467.

[9] Gramfort A, Luessi M, Larson E, Engemann DA, Strohmeier D, Brodbeck C, et al. MEG And EEG data analysis with MNE-python. Front Neurosci 2013;7(7 DEC):267. http://dx.doi.org/10.3389/fnins.2013.00267.

[10] Kingma DP, Ba JL. Adam: A method for stochastic optimization. In: 3rd International conference on learning representations, ICLR 2015 - conference track proceedings. International Conference on Learning Representations; 2015, URL https://arxiv.org/abs/1412.6980v9.

[11] Tensorflow API: tf.keras.losses.CategoricalCrossentropy. 2016, URL https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy.

[12] Tensorflow API: tf.keras.losses.MeanSquaredError. 2016, URL https://www.tensorflow.org/api_docs/python/tf/keras/losses/MeanSquaredError.

[13] Breiman L. Random forests. Mach Learn 2001;45(1):5–32. http://dx.doi.org/10.1023/A:1010933404324.

[14] Weichwald S, Meyer T, Özdenizci O, Schölkopf B, Ball T, Grosse-Wentrup M. Causal interpretation rules for encoding and decoding models in neuroimaging. NeuroImage 2015;110:48–59. http://dx.doi.org/10.1016/j.neuroimage.2015.01.036.