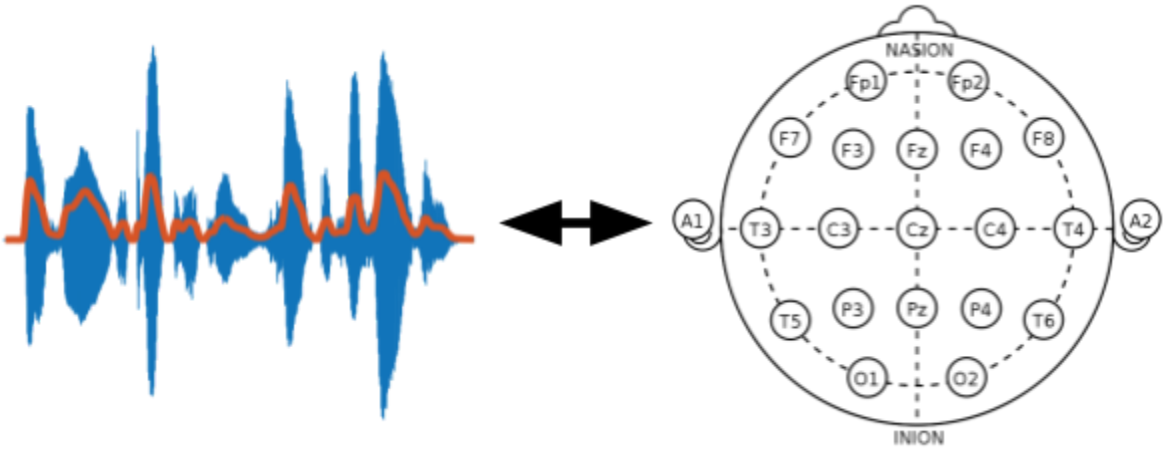
Telluride Decoding Toolbox



Sahar Akram (UMD), Alain de Cheveigné (ENS), Peter Udo Diehl (ETH), Emily Graber (Stanford), Carina Graversen (Oticon), Jens Hjortkjaer (DTU), Nima Mesgarani (Columbia), Lucas Parra (NYU), Ulrich Pomper (UCL), Shihab Shamma (UMD), Jonathan Simon (UMD), Malcolm Slaney (Google), Daniel Wong (ENS)

Telluride Decoding Toolbox

Introduction

This document introduces the Telluride Decoding Toolbox. The toolbox offers a test bed for algorithms for decoding brain signals in relation to a stimulus. Our goal is to provide a standard set of tools to allow users to decode brain signals into the signals that generated them—whether the signals come from visual or auditory stimuli, and whether they are measured with EEG, MEG, ECoG or any other response for decoding.

In line with the dictionary definition of decoding, we want to convert (a coded) message into intelligible language, or at least determine which of several messages was intended. The tools in this toolbox allow any perceptual stimulus to be connected to any neural signal. Although the developers of this toolbox are largely researchers who meet in Telluride Colorado for a Neuromorphic workshop and use EEG to analyze auditory experiments, the tools in this toolbox allow any perceptual stimulus to be connected to any neural signal.

Classically, temporal responses of EEG signals have been analyzed using event-related potentials (ERP) or time-frequency analysis. ERP studies repeat a discrete short signal many times, average the response over repetitions and look for changes in the peaks and valleys of the stimulus-locked response. MMN (mismatch negativity) is a particular event-related response that examines oddball responses in the context of repetitive stimuli. Time-frequency analysis examines the changes in the EEG spectrum in particular bands over time. BCI (brain-computer interface) experiments may look for some spectral change in the brain response that researchers can use to indicate a choice—the problem is treated as a classification problem.

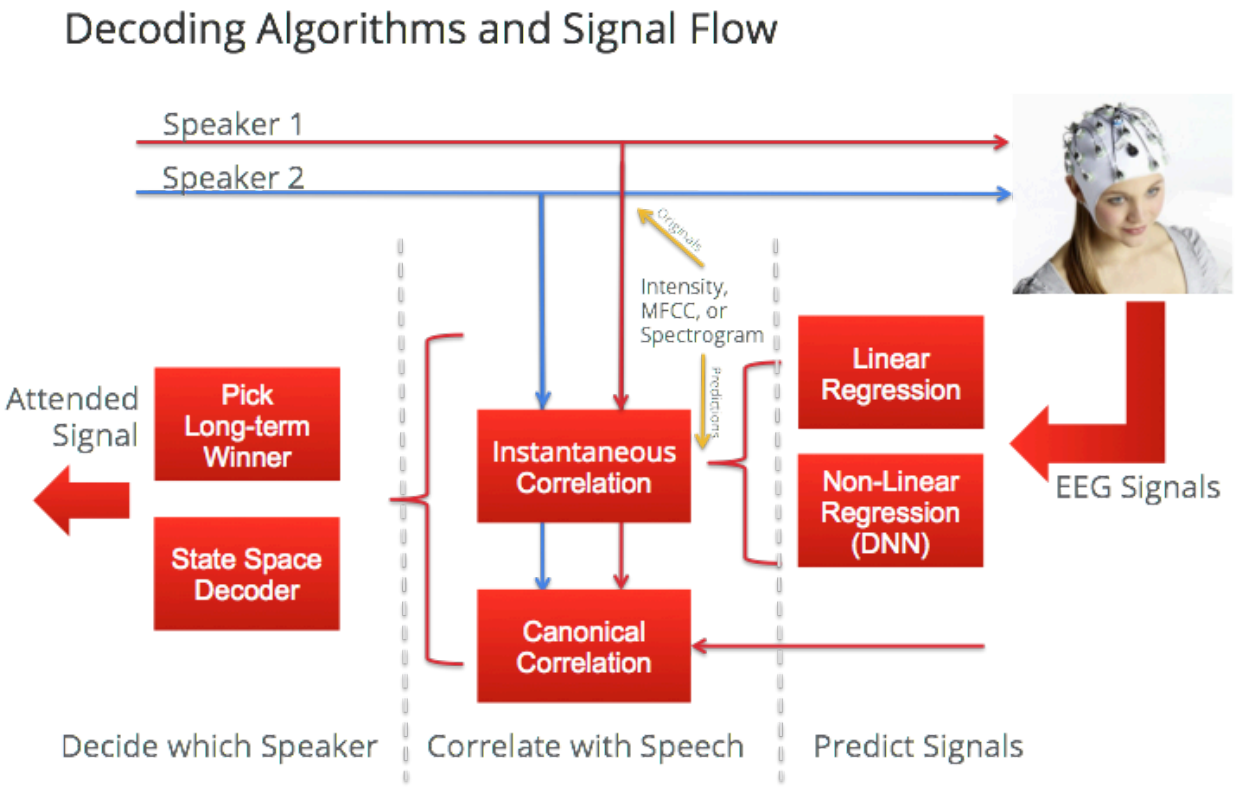
The Telluride Decoding Toolbox takes a new approach. Given a stimulus and the resulting brain response, we want to find the relation between the two. The aim may be to decode the response to approximate the original stimulus, or choose among alternative stimuli, or determine the attentional state of the listener. In the simplest case we want to find the auditory signal that generated the measured EEG response, or at least predict to which of two auditory stimuli a user is attending. The goal is to produce a continuous prediction of the neural signals that are likely to be produced by a stimulus (the forward problem), or given an EEG or MEG response, find the stimulus that generated it.

This is a hard problem. Given an EEG or MEG signal, we can't yet synthesize the auditory signal with high fidelity. But we can already say with much better than chance whether the subject was listening to speaker A or speaker B. Thus there is information that can be decoded, and with better machine-learning technology and more data, the time has come to think about pushing the limits of neural decoding. People have done this with spikes [Bialek, 1991], but we want to look at a wider range of cortical signals. While the correlations obtained are still low, we think they will get better with time.

Overview

This toolbox offers general-purpose routines to implement and test new decoding methods, and a set of predefined methods and data for reference purposes and as examples.

A common application of these tools is to decode attention: estimate which auditory source is a listener attending. As shown in the attention-decoding block diagram below, this toolbox addresses three parts of the decoding problem: predicting the stimuli, correlating with the expected signals, and deciding the attended speaker.



To connect audio and EEG (or the reverse) we provides two basic forms of predictions: linear and non-linear. They each have different strengths and needs. Linear prediction is easiest to formulate, analyze, and calculate. Non-linear predictions based on deep neural networks (DNNs) represent a rich source of possibilities but with a larger computational cost.

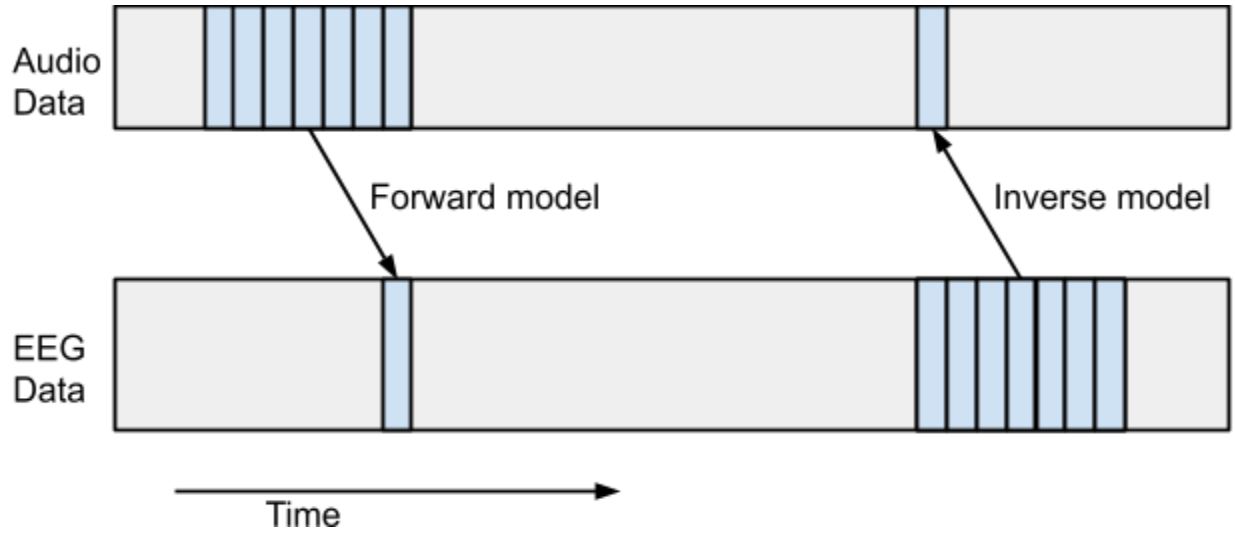
Then to decide which speaker a subject is attending, we correlate the predictions with the original stimuli. Correlation answers a simple question: are two signals related, and thus might perform better in some situations. This can be done in any of several feature domains, such as intensity, MFCC and spectrograms.

Finally, given the correlation signals, we need to decide to which speaker the subject is attending. This can be done by simply summing the correlations over time, and picking the winner. Or a more principled approach combines a probabilistic model of how long a time a subject is likely to attend to one subject along with the behavior of the correlation signal, and chooses a winner that optimizes a loss function that combines these two factors.

Most importantly, this toolbox calculates temporal response functions, TRFs. The TRF describes the connection between the stimulus and the recorded responses, or the inverse, going from

responses back to the stimuli. For the linear model (FindTRF) this is equivalent to a multi-channel convolution. While for the DNN approach the TRF is a non-linear function of its inputs.

Decoding algorithms depend on temporal context to make their predictions. For example when predicting the forward response (audio->eeg), the audio right now will generate a response over several tens of milliseconds to come. Conversely, to estimate the EEG signal now we need to know the audio signal from the previous ~200ms. This is called context, and is shown in the figure below. In these cases, the model predicts the response based on 7 frames of data. The FindTRF and DnnRegression routines take a context-window size, which is an integer representing the number of frames of data, either forward or backward in time depending on the model direction, to include as input data in the analysis.



The toolbox includes the code (MATLAB or Python) to implement the four models described above, along with necessary test code. We have also included sample EEG and MEG data so new users can verify that the algorithms work as promised.

We demonstrate the basic behavior of the three algorithms with a synthetic dataset. This dataset starts with a deterministic attention signal and creates two random 32 channel EEG system impulse responses, one for the attended and one for the unattended signal. It correlates two sinusoidal test signals with the appropriate impulse response to get the EEG response. The goal of this toolbox is to reconstruct the original attention signal---was the subject listening to speaker 1 (sinusoid test signal 1) or speaker 2 (sinusoid test signal 2). See the CreateDemoSignals function for the setup, and each algorithm for its behavior.

It is important to note that all data in this dataset is organized as MATLAB arrays, where time is *always* the first dimension. Every dataset in our applications includes time, so it makes sense to put this dimension first since it is required.

This toolbox includes both code and real data. The datasets include (to be finalized.. just a guess now):

1) EEG data with a single subject (data/Telluride-demo.mat)
2) MEG data with two sound sources and subject attends to one

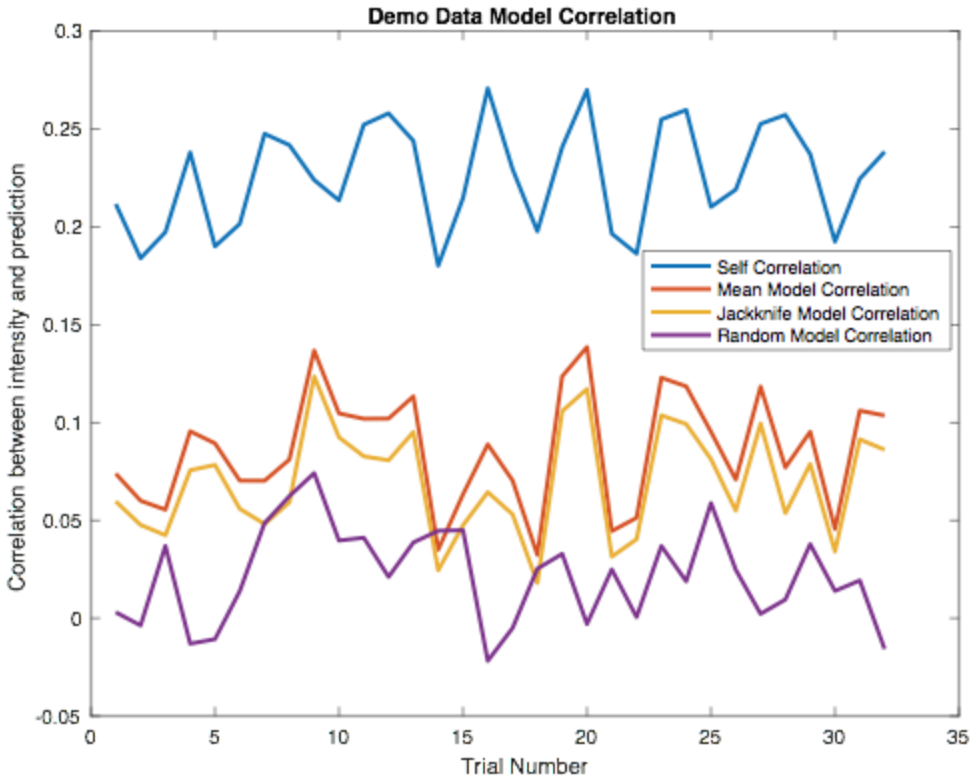
For more details see Appendix 1.

Comparisons and Evaluations

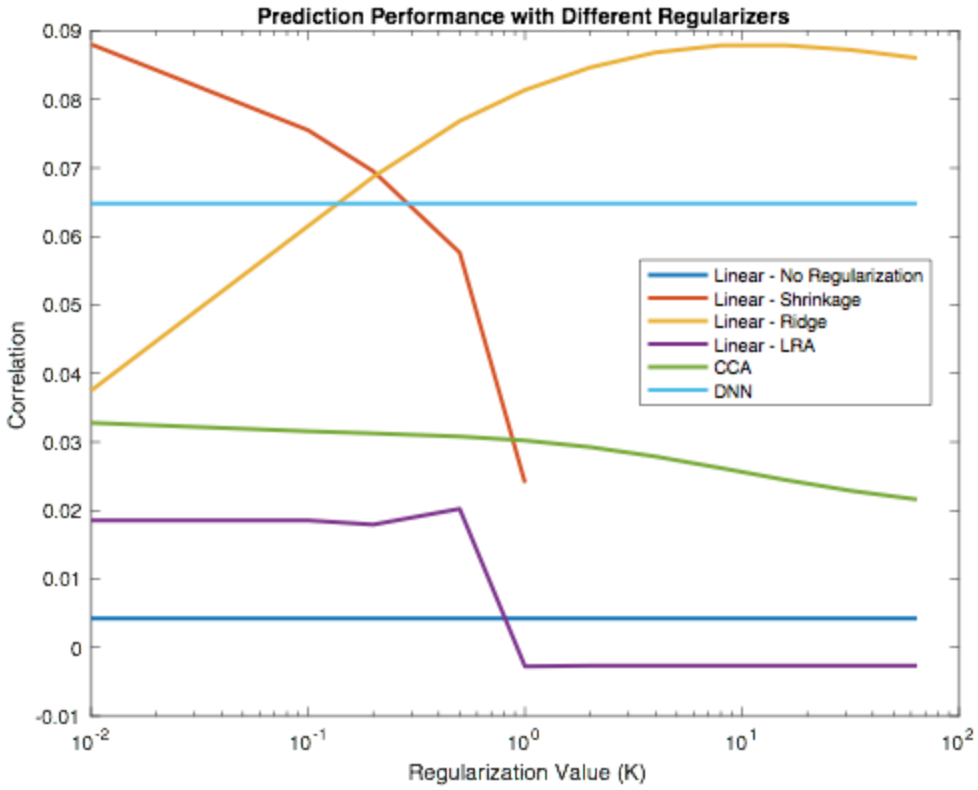
Having all of these routines in one place, along with some sample data, allows us to compare and contrast the different approaches. In this section, we use the EEG Demo dataset for this evaluation (See Appendix 1). The data happened to be recorded from four subjects at once, 16 channels per subject, all listening to one of four different speech signals. But conceptually, it represents a simple speech to EEG recording experiment, and we use the combined data to illustrate the various algorithms.

The data was recorded in 32 blocks, each about 4 minutes long. The plot below shows four tests, as a function of trial number, using the normal FindTRF linear regressor. In each case we trained a regressor using some of the data, and then predicted the intensity for that trial.

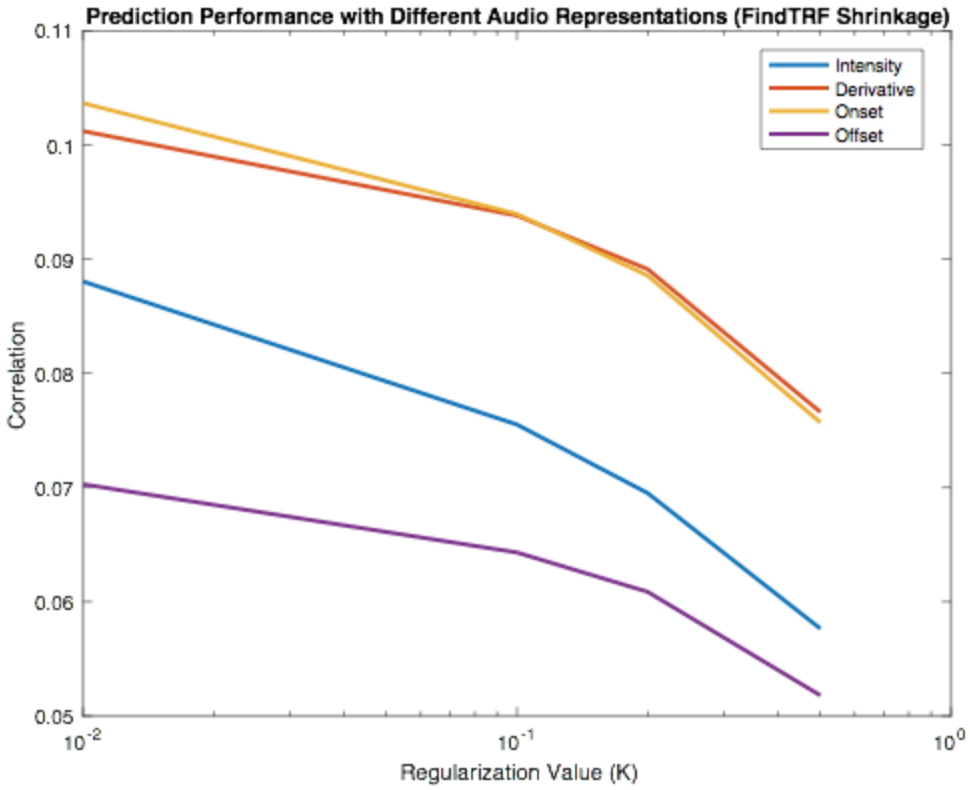
- Self correlation: Train a decoder on the same trial data used to test the regressor. This is the “cheating” approach, and should always produce the best (highest) correlation.
- Mean model: Train a decoder separately for each of the 32 trials, and then average all the regressor matrices together to form a single model. This single model is then used to estimate the intensity of each model. This is still not a good model because the test data is used in the training.
- Jackknife model: Calculate a decoder by averaging the models for a random selection of 31 trials, and test it by applying the model to the 32nd trial. This is the cleanest test because the test data is not used in any way to calculate the model.
- Random model: The data from a single trial (not the test trial) was randomly selected and used to train a TRF. This TRF was then used to estimate the intensity for the test trial. This is a clean test, since the test data and the training data are separate, but it is low performing because the model is estimated from only 4 minutes of training data (compared to 31x4 minutes in the jackknife model above.)



The figure below shows the estimated correlation in a speech-decoding task as a function of the algorithm's regularization parameter. Note, not all regularization values make sense---the meaning of any regularization parameter depends on the algorithm used. [The first four listings in the legend below represent variations of the FindTRF linear regressor.]



This toolbox lets us easily explore other representations of sound. One possibility is that our auditory system is not sensitive to intensity, but to changes in intensity. The plot was computed using this Toolkit, and shows that the derivative of the intensity and the positive portions of the derivative (onsets) works better than raw intensity. Furthermore, the negative portions (offsets) are not very predictive of the EEG signal.



Code

This toolbox is implemented with a combination of MATLAB and Python. The basic decoding routines (FindTRF, CCA, ViterbiSearch) only use the basic MATLAB software, while the StateSpace model depends on the Mathworks Optimization toolbox. The DNN code is implemented in Python, using the PyLearn2 and Theano packages, and there is a MATLAB routine that calls the Python routines so you can do all your work in MATLAB if you choose.

The code for this toolbox is available from GitHub at

https://github.com/Neuromorphs/telluride-decoding-toolbox

You can download the software by cloning the repository, or download all the software in a ZIP archive (look for the button that says download ZIP).

License

This software is provided under the Apache 2 license. We encourage you to use this toolbox in your work, and we are eager to receive improvements, comments and bug reports.

Publications

Please reference this toolkit as follows:

> Sahar Akram, Alain de Cheveigné, Peter Udo Diehl, Emily Graber, Carina Graversen, Jens Hjortkjaer, Nima Mesgarani, Lucas Parra, Ulrich Pomper, Shihab Shamma, Jonathan Simon, Malcolm Slaney, Daniel Wong. Telluride Decoding Toolbox. Institute for Neuromorphic Engineering, http://www.ine-web.org/software/decoding, 2015.

Acknowledgements
This toolbox was conceived and the authors were brought together by the Telluride Neuromorphic Cognition Engineering Workshop, held annually in Telluride, Colorado.

We are especially grateful for support we received from BrainVision which provided EEG equipment for our wild experiments, some of which turned into interesting results. In addition, Mathworks provided MATLAB licenses to many of our students. All the authors had support from their home institutions to attend the workshop and to continue this work.

References
W Bialek, F Rieke, RR de Ruyter van Steveninck, & D Warland. Reading a neural code. *Science* 252, 1854–1857 (1991).

CCA

Canonical correlation analysis

Syntax

[Wx, Wy, r] = cca(X, Y, regularization)

Description

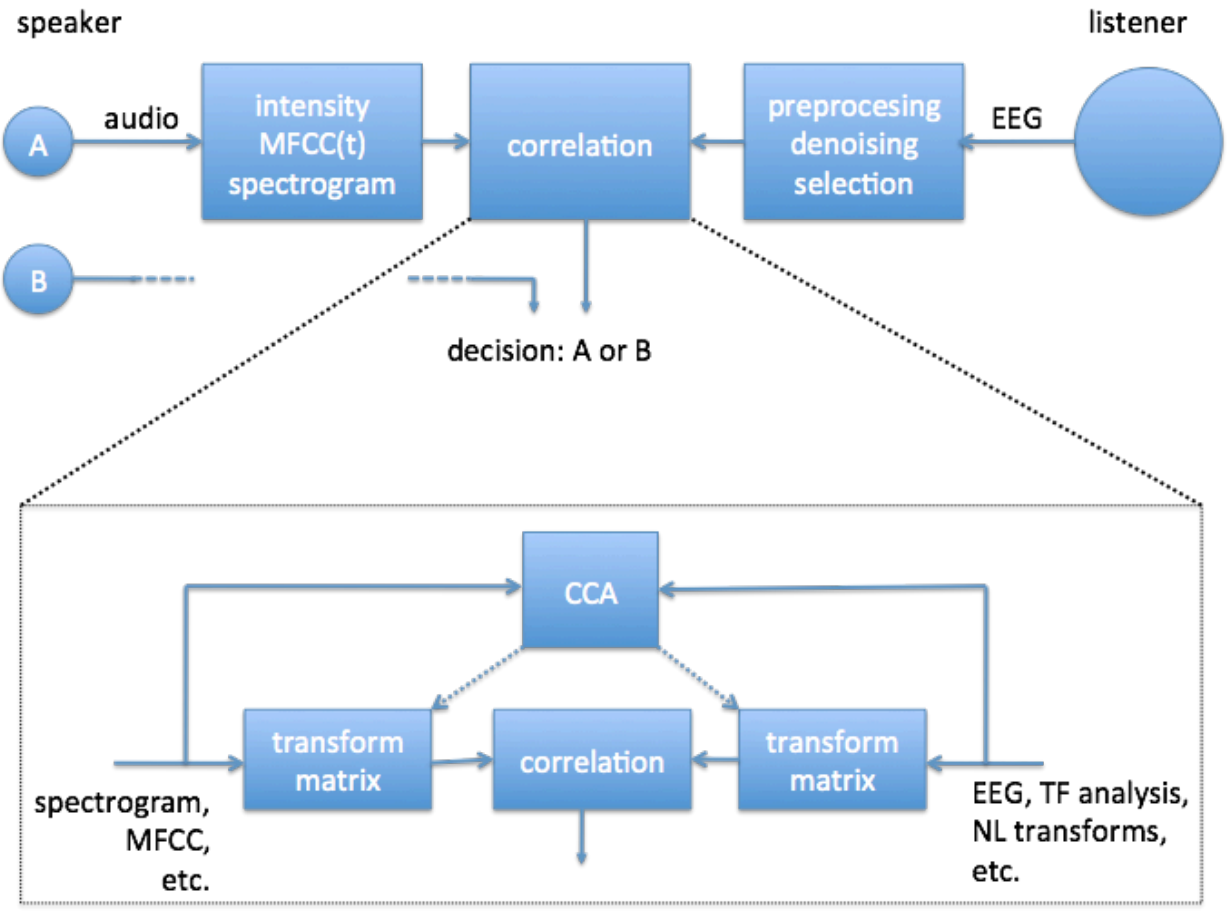
Canonical Correlation Analysis (CCA) is a tool to find a linear relation between two datasets. Specifically, it finds a linear transform of set A with components that are best correlated to set B, and a similar transform of set B. CCA is closely related to PCA. Like PCA, the components are mutually orthogonal (the time series are decorrelated two-by-two), their variance sums up to that of the original dataset (the transform is a rotation), and they have a clearly defined order. However, contrary to PCA, which orders principal components by decreasing *variance*, CCA orders them by decreasing variance *of their projection* on the other dataset. The greatest difference between CCA and PCA is of course that CCA produces two transforms, one for each dataset, whereas PCA produces only one.

CCA requires that data sets A and B have the same number of rows (time samples), but they can have a different number of columns. However CCA is really only useful if both A and B have more than one column, otherwise it boils down to a simple projection of one set on the other. CCA is powerful in that it can "discover" the presence of a pattern that is hidden within both A and B, even if that pattern has such a poor SNR that no other technique could find it.

CCA is a great tool for audio-to-brain decoding, because it can help find the transform to apply to the audio to get something predictive of the EEG response that it might elicit, and a transform of the EEG to find some aspect of brain activity that cares about sound. This is much more powerful than just hopefully choosing some arbitrary representation of the audio (e.g. envelope or TF-analysis channel) or some selected electrode to use for decoding. Concretely: (1) create a multichannel representation of the audio (e.g. TF analysis), (2) apply CCA to find two transforms, one for the audio the other for the EEG, (3) apply them and select the first component on each side. These two component signals (one from audio the other from EEG) have the highest possible correlation of any linear transforms of the data.

CCA assumes that the relation between A and B is scalar (instantaneous) and linear. However it is easy to extend it to convolutional relations between A and B by augmenting either dataset (or both) with a set of time shifts (or a filter bank). Likewise it is possible to address non-linear relations by applying an appropriate set of non-linearities to the data. For example, one can find a linear relation between A and the *instantaneous power* of some component hidden within B by applying CCA to A and to the set of quadratic forms derived from B (as in de Cheveigné 2012). This will find the component even if its SNR within B is very small.

As for any data-driven analysis technique, the greatest peril in using CCA is overfitting, possibly more severe with CCA than other techniques because the temptation is strong to throw in extra dimensions (e.g. time shifts and non-linearities), to see if they improve correlation factors. To limit overfitting one needs to reduce dimensionality at every stage, and apply standard cross-validation techniques.



Examples

We use the CreateDemoSignals script to create test signals. We use about +/-12 seconds of data on either side of the first attention switch as training data. Anything less, for this example, is not enough to get a good answer. We then add context to the data (38 frames) before computing the CCA model.

```
Lags = 1:38;

ccaTrainingWindow = attentionDuration/2;    % Choose a small window for training
iTrain = find(recordingT>attentionDuration-ccaTrainingWindow & ...
    recordingT < attentionDuration+ccaTrainingWindow);

laggedTrainResponse = LagGenerator(response(iTrain, :), Lags);
[wAttendedAudio,wAttendedEEG] = cca(attendedAudio(iTrain)', laggedTrainResponse');
[wUnattendedAudio,wUnattendedEEG] = cca(unattendedAudio(iTrain)',
laggedTrainResponse');
```

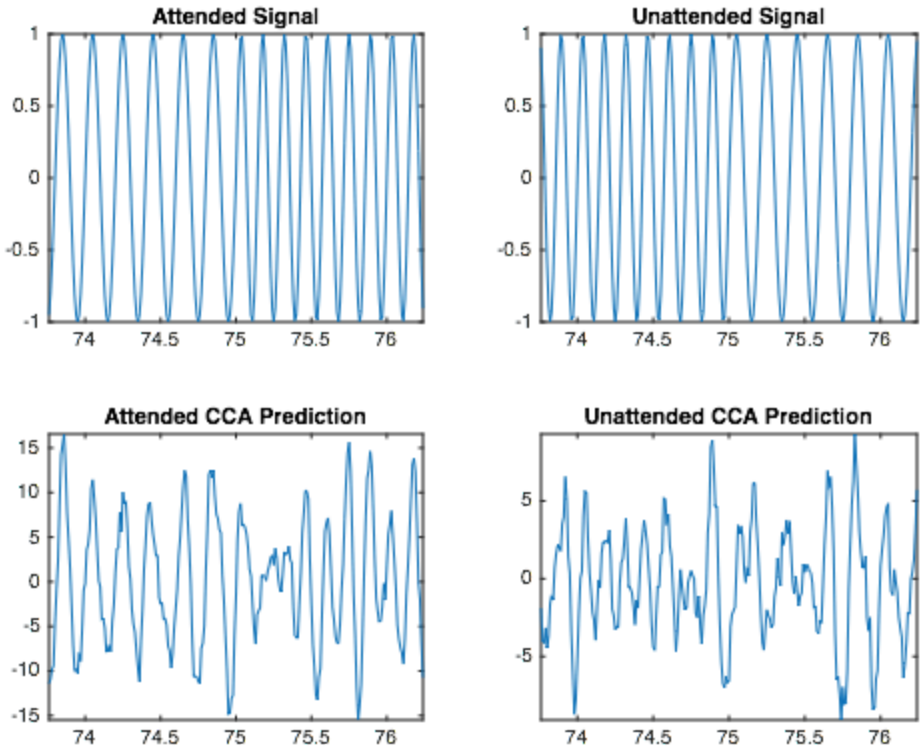
In this case, the audio data is single-dimensional and the audio transform becomes a scalar. The EEG transform then rotates the EEG data so that it is maximally correlated with the audio data.

```
laggedTestResponse = LagGenerator(response, Lags);
ccaAttendedPrediction = laggedTestResponse * wAttendedEEG;
ccaUnattendedPrediction = laggedTestResponse * wUnattendedEEG;
```

We can plot the attended and unattended responses:

```
clf
attentionSwitchPick = 3;
plotWindow = attentionDuration/20;
iPlot = find(recordingT>attentionSwitchPick*attentionDuration-plotWindow & ...
    recordingT < attentionSwitchPick*attentionDuration+plotWindow);

subplot(2, 2, 1);
plot(recordingT(iPlot), attendedAudio(iPlot()));
title('Attended Signal'); axis tight

subplot(2, 2, 2);
plot(recordingT(iPlot), unattendedAudio(iPlot()));
title('Unattended Signal'); axis tight

subplot(2, 2, 3);
plot(recordingT(iPlot), ccaAttendedPrediction(iPlot()));
title('Attended CCA Prediction'); axis tight

subplot(2, 2, 4);
plot(recordingT(iPlot), ccaUnattendedPrediction(iPlot()));
title('Unattended CCA Prediction'); axis tight
```

to get this summary plot. Note: the sinusoid has a lower frequency in the first half of each attended plot. Furthermore, the prediction of the attended signal has higher fidelity than the unattended signal, since its signal-to-noise ratio is higher.

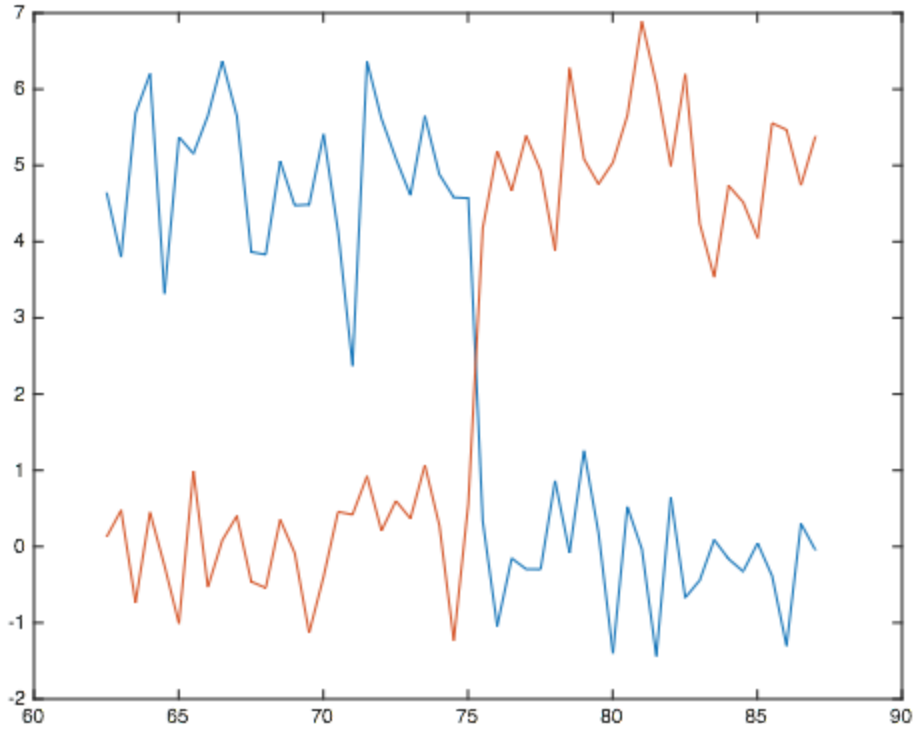


We can then compute the correlation between the original and predicted audio signals, downsampling to 2Hz so we can see a less noisy signal.

```
cors1 = ccaAttendedPrediction(1:length(audioS1)) .* audioS1;
cors2 = ccaAttendedPrediction(1:length(audioS2)) .* audioS2;

newFs = 2;
```

```
cors1d = resample_fft(cors1, fs, 2);
cors2d = resample_fft(cors2, fs, 2);
recording2T = recordingT(1:(fs/newFs):end-1);
i2Plot = find(recording2T>attentionSwitchPick*attentionDuration-1*ccaTrainingWindow &
...
    recording2T < attentionSwitchPick*attentionDuration+1*ccaTrainingWindow);

clf; plot(recording2T(i2Plot), [cors1d(i2Plot) cors2d(i2Plot)]);
legend('Signal 1', 'Signal 2'); xlabel('Seconds'); ylabel('Correlation');
```

This results in a summary correlation, around the third attention switch, that looks like this.



Note, the model that was trained for this CCA example needed 25 seconds of training data, while the linear model trained with FindTRF needed 2.5 seconds. Better results are possible by subsampling the lag vector, and/or using different normalization methods.

Author

We are grateful to Magnus Borga from Linkopings Universitet who provided this software: http://www.imt.liu.se/~magnus/cca/. For more information see his tutorial http://www.imt.liu.se/~magnus/cca/tutorial/tutorial.pdf

References

de Cheveigné, A. (2012). Quadratic component analysis. NeuroImage, 59(4), 3838–3844. http://doi.org/10.1016/j.neuroimage.2011.10.084

CreateDemoSignals

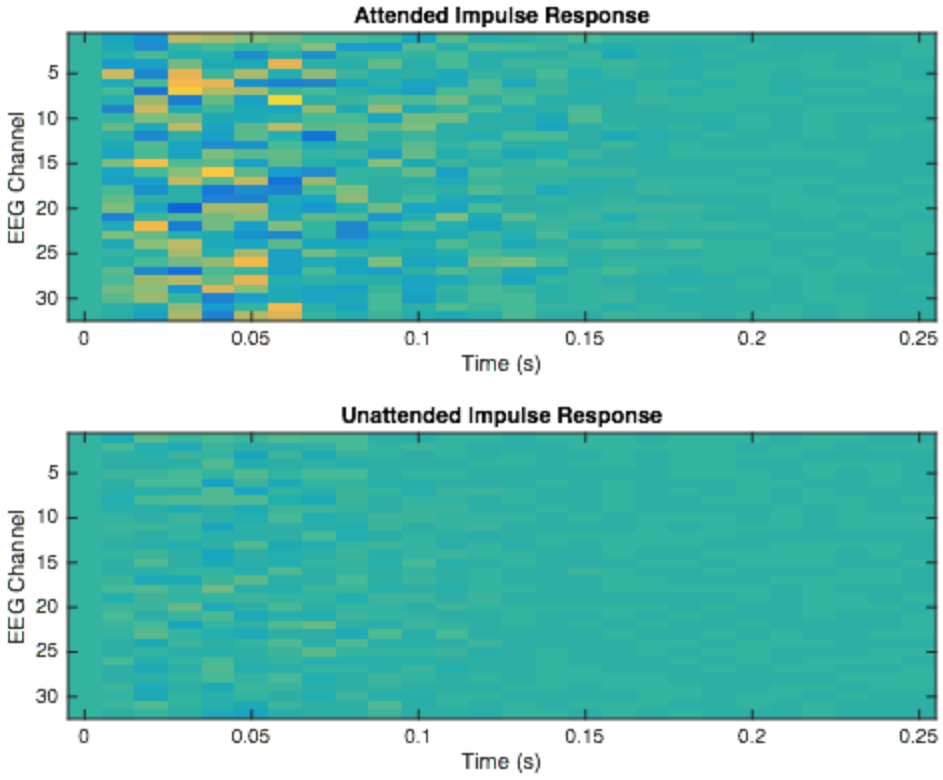
Create the signals used to describe the primary methods in this toolbox

Syntax

CreateDemoSignals

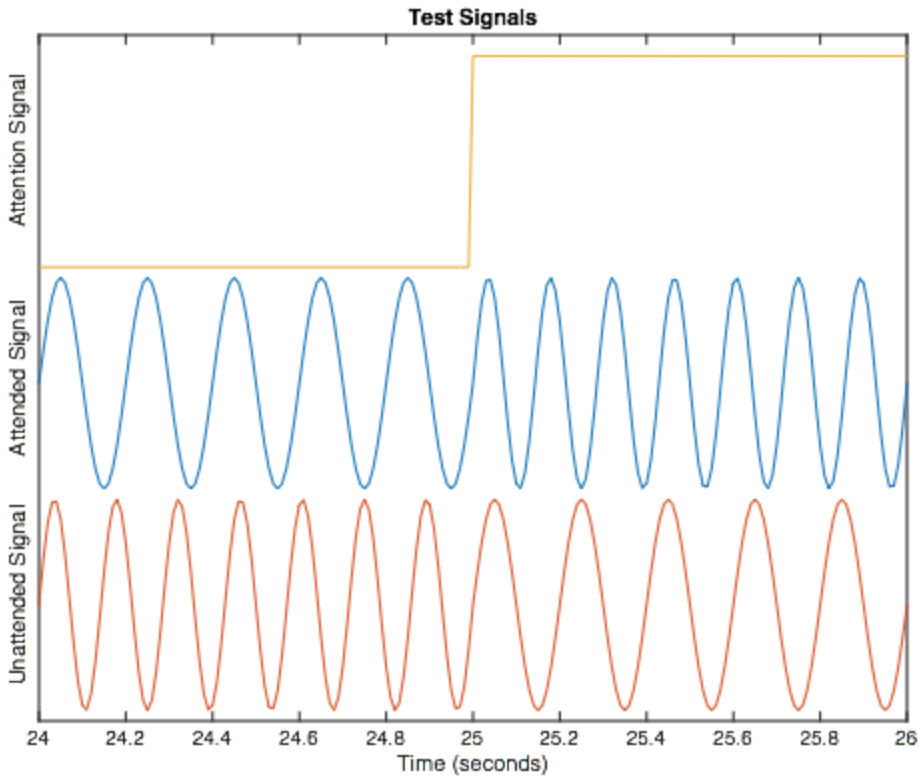
Description

The CreateDemoSignals script creates a number of synthetic signals that are used to demonstrate the different algorithms in this toolbox. Most importantly, it creates a random linear TRF that describes the (simulated) EEG signal for a pair of auditory inputs: the attended and unattended signals. The toolbox assumes that the impulse response peaks around 50ms, and is finished after 250ms. In addition, the impulse response of the unattended signal is attenuated, as might be expected for a signal that is harder to pick up. With added noise, as is always present in a neural measurement, the resulting unattended signals will be harder to decipher.



We generate an artificial attention signal that tells which of two sounds the subject is attending. This signal is binary and switches sides every 25s. We generate two sinusoids, at 5 and 7Hz, to simulate the input stimuli. Under control of the attention signal, the “subject” switches between the 5 and 7Hz input signals. These three signals are shown in the plot below. The top plot shows the attention signal, and the switch at t=25s. The middle curve shows the attended signal, where the subject listens to the 5Hz signal for the first 25s, and then switches to listening

to the 7Hz signal. The bottom curve show the unattended signal, with the opposite frequencies at each time.



The final output of this script are the attendedResponse and unattendedResponse matrices. Each of these signals is a 10000x32 matrix (100 seconds of audio at 100Hz). These matrices, and the attendant experimental parameters, are used as demonstration signals for all the routines in this toolbox.

Note: the “audio” signal can represent any high-level feature describing the audio waveform. In many experiments [Mesgarani 2012, O’Sullivan 2015] this is the waveform energy (intensity). But others have investigated MFCC [ ], and spectrograms [ ]. In this demo we use the simplest, the intensity.

Examples

CreateDemoSignals is a MATLAB script. When you run it, it creates several variables that are useful in testing and demonstrating the tools in this toolbox.

```
>>> CreateDemoSignals
```

This code produces the following variables

Variable	Size	Meaning

T	1x1	Length (seconds) of total signals
recordingT	10000x1	Time of each sample frame (in seconds)
attentionDuration	1	Length of time subject attends to each signal (seconds)
fs	1	Sample rate for audio and EEG signals (100Hz).
nEEGChannels	1	Number of EEG channels in this demonstration (32)
attentionSignal	10000x1	Signal that determines whether subject is attending to signal 1 (value = 0) or signal 2 (value = 1). Flips between 0 and 1, every 25 seconds.
attendedAudio	10000x1	Composite audio signal to which the subject is attending. A mix, based on the attentionSignal, of the two simulated audio signals.
unattendedAudio	10000x1	Composite audio signal to which the subject is NOT attending.
attendedImpulseResponse	26x32	Random impulse response that connects audio to EEG. There are 26 time samples and 32 channels.
unattendedImpulseResponse	26x32	Random impulse response that connects audio to EEG for unattended response.
response	10025x32	Total EEG signals for attended plus unattended audio.

References

Mesgarani, N., Chang, E. F., (2012), “Selective cortical representation of attended speaker in multi-talker speech perception”, Nature 485

James A. O'Sullivan, Alan J. Power, Nima Mesgarani, Siddharth Rajaram, John J. Foxe, Barbara G. Shinn-Cunningham, Malcolm Slaney, Shihab A. Shamma and Edmund C. Lalor. "Attentional Selection in a Cocktail Party Environment Can Be Decoded from Single-Trial EEG." *Cerebral Cortex,* January 2014.

CreateLoudnessFeature

Analyze an audio signal and estimate its energy.

Syntax

loudnessFeature = CreateLoudnessFeature(audioData, audioFS, loudnessFS)

Description

Compute the loudness (intensity) of an audio signal by averaging the squared energy.

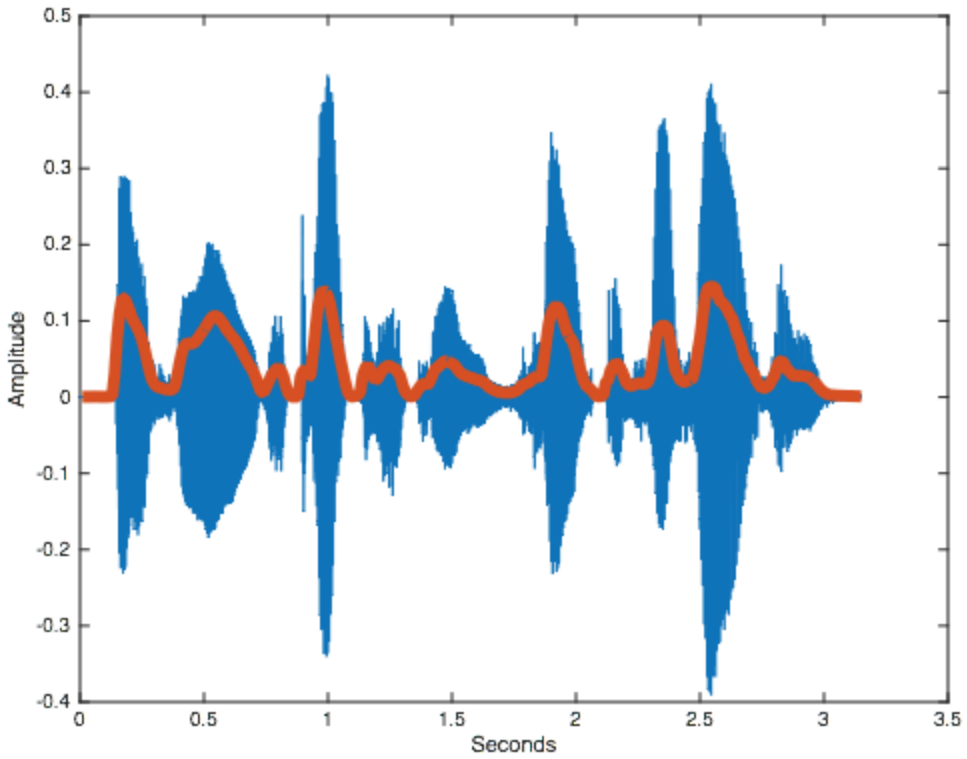
Given audioData at a sample rate of audioFS, compute the loudnessFeature of the sound. This is done by finding the RMS energy within +/-1.5 times the windowSize (loudness sample interval divided by audio sampling interval) samples of the center point.

Example

To compute the loudness (energy) of the signal, this sample code works.

```
[tap,fs] = audioread('tapestry.wav');
lfs = 100;
loudness = CreateLoudnessFeature(tap', fs, lfs);
h  = plot((1:length(tap))/fs, tap, ...
    (1:length(loudness))/lfs, sqrt(loudness));
xlabel('Seconds'); ylabel('Amplitude');

set(h(2), 'LineWidth', 6)
```

This produces the plot shown here (best seen in color):



Inputs

audioData - Monophonic audio data, a single column of data

audioFS - The sampling rate for the audio siganl.

loudnessFS - The desired sampling rate for the loudness signal.

Outputs

loudnessFeature - A vector representing the energy (loudness) of the input signal at the new sample rate.

DNNRegression

Use a deep neural network (DNN) to find a model connecting stimulus and response.

Syntax

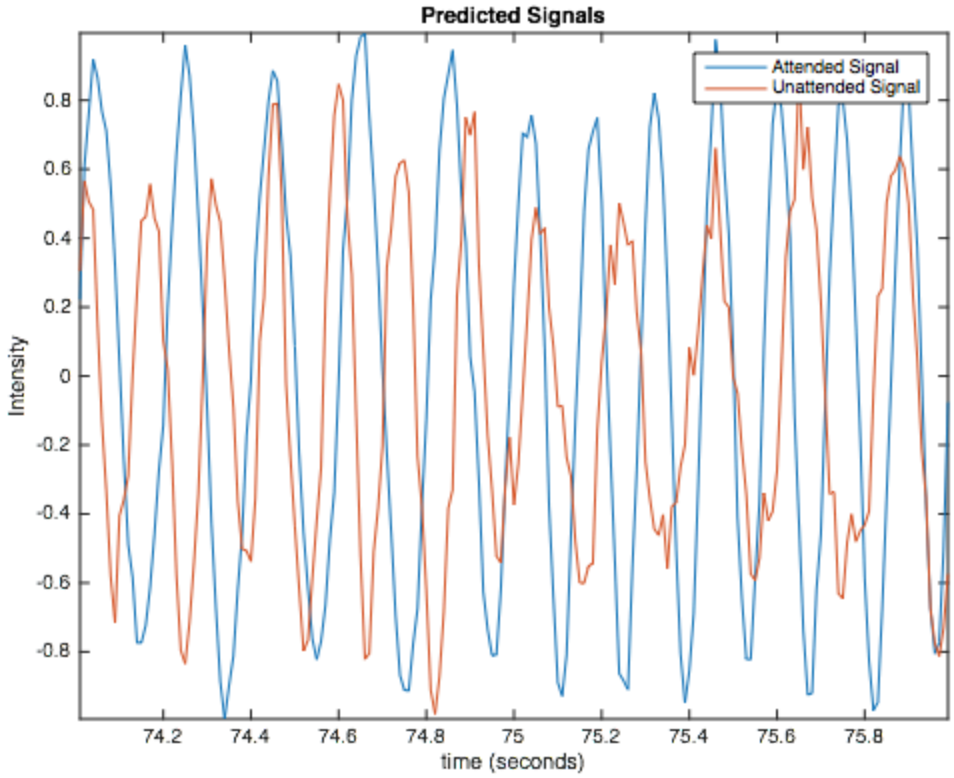
[g,pred] = DNNRegression(stimulus, response, direction, testdata, g, Lags)

Description

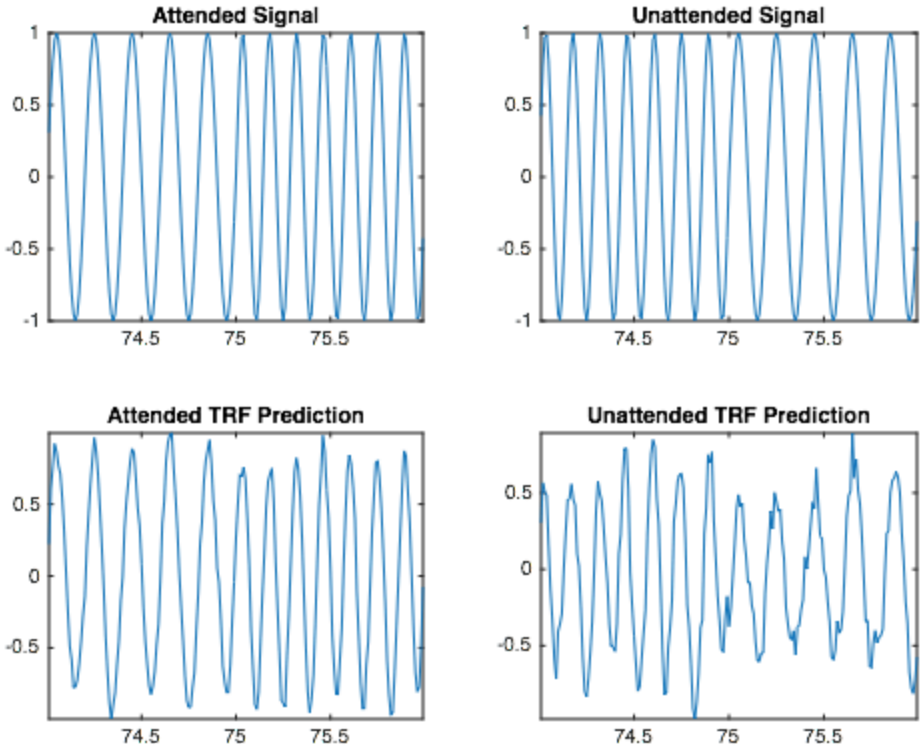
This function provides a MATLAB wrapper so that the Python DNN code can be called from MATLAB. This function takes the given data arrays, writes them to temporary files, call the Python interpreter to do the necessary calculations, and then reads back into MATLAB the model (a matrix since it is a single-layer DNN) and the model's predictions from the test data. See the documentation for DNNRegression.py for more details.

Examples

This is plot shows the output of the DNN decoder on the standard synthetic dataset.



Here is a comparison of the real (top) and predicted (bottom) signals, for the attended (left) and unattended (right) signals.



Inputs

stimulus - The (audio) stimulus response data of size $N_{time\text{-}samples}$ x $N_{channels}$. If both the stimulus and the response matrices are non-null, then this routine calculates a new model (g).

response - The brain signals of size $N_{time\text{-}samples}$ x $N_{channels}$. If both the stimulus and the response matrices are non-null, then this routine calculates a new model (g).

direction - One of the two strings: forward or reverse, to indicate whether to compute the forward or reverse predictions.

testdata - If present, then the computed model (if the stimulus and response are given) or conversely the pre-computed model, is used to predict the corresponding result. Thus if direction is ‘forward’ then the test data is used to predict a neural signal.

g - The model (currently a single matrix) if you do not wish to calculate a new model. It is used to process the test data.

Lags - The number of temporal frames of data that are added to the present sample as context in the prediction.

Outputs

g - A new model to describe the DNN prediction

pred - A prediction from the model. The meaning of the prediction depends on the direction. If the direction argument is ‘inverse,’ then the testdata is a brain signal, and the prediction is a estimate of the original audio.

DNNRegression.py (Python)

A Python program to find a non-linear model connecting stimulus and response.

Syntax

python DNNRegression.py {--train, --predict}
-m model.yaml
-s stimulus_data [i/o, default text, or .mat file]
-r response_data [i/o, default text, or .mat file]
-w weights [i/o, default text, or .pkl]
[--visual visualize.png]
[--context N for N>=1 *25]
[--numEpochs N default is 100]
[--dir forward/backward*]
[--valid which parts are valid, in case of concatenating trials, default all valid.]

Description

This program uses a deep neural network (DNN) to find a non-linear model that connects the (audio) stimulus and the (EEG/MEG/eCog) response. The DNN, as configured by default in the model.yaml file is a single layer with a rectified linear non-linearity. The dimensions of the DNN depend on the input and output, and the direction of the regression.

You must supply either the -predict or -train argument to specify the type of calculation. Estimating the model from the stimulus and response when -training is used, and predicting one or the other when the -predict flag is used.

The -direction flag specifies the direction of the regression. In the forward case, the response is predicted from the stimulus. While in the backward case, the stimulus is predicted from the measured response.

The data for the regression, specified by the -stimulus and -response arguments, are supplied in one of two ways. The default is a text file, with one line containing all the channels for one time step, different columns are separated by whitespace. Time goes down the file. Alternatively, the data can be supplied in a binary file ending in ".mat" and containing a single rectangular matrix (matrix named 'data') of the same format as the text file. Both these file names must be specified, although when predicting with the model, depending on the direction, one file is used as input, and the other file is written (overwriting any data already in that file!)

The output weights are either a text file, as long as the model only contains a single layer, or if the filename ends in ".pkl" then the data is written out in Python's pickle format. The model weights (for each layer) have a size of #input_channels x #output_channels. These numbers, and their connection to stimulus and response, depend on the direction the model is run.
The -context argument specifies the number of frames (N) to use as context in the regression. The exact meaning depends on whether the model is predicting the forward or the reverse response. As shown in the figure at the start of this manual, when predicting the response N frames starting at the current stimulus frame and going backwards are used as input. When

predicting the stimulus from the response (the backward model) then N frames starting at the current frame of the response are used as input to the DNN.

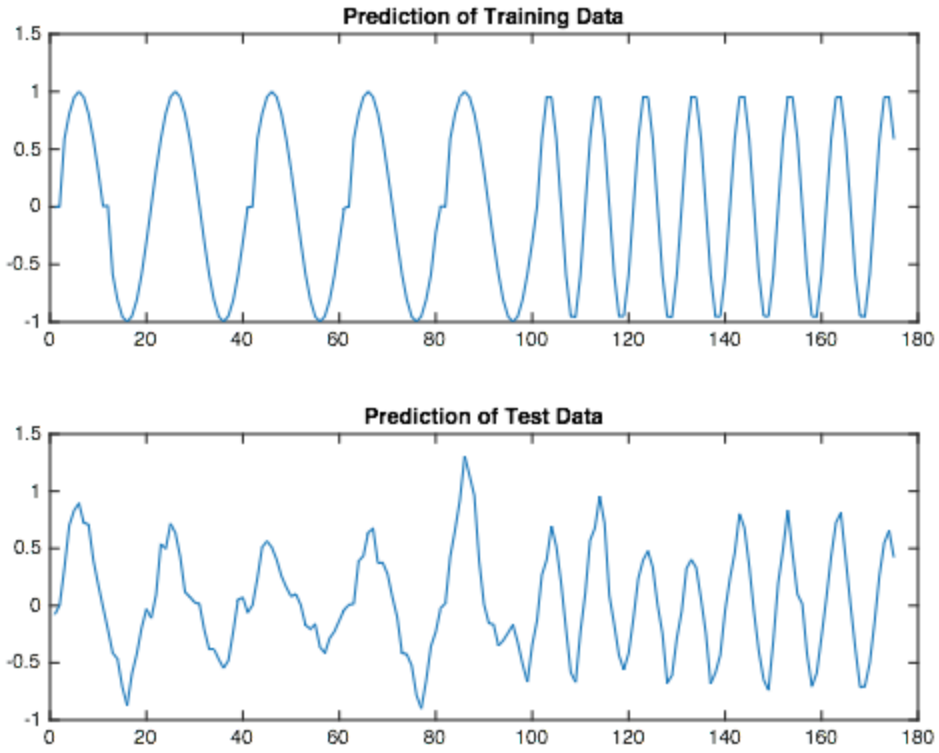
Examples

For training:

```
python DNNRegression.py -t -m "./network.yaml" -s "trainUnattendedAudio.mat" \
       -r "trainResponse.mat" -w "./network_best_3.pkl" --debug
```

For testing:

```
python DNNRegression.py -p -m "./network.yaml" -s "testUnattendedAudio.mat" \
       -r "testResponse.mat" -w "./network_best_3.pkl" --debug
```



Similar to the results of the TRF method, the prediction of the attended signal shows a correlation of ~0.986 with the actual attended signal. The unattended signal has a lower correlation of ~0.922 which is slightly better than the TRF method. In both cases, attended and unattended, +/- 1.0 second of training data was used with a window size of 0.25 seconds.

Input Parameters

-train - Train a model

Either this or the -predict argument must be used. The -train argument specifies that the stimulus and response files will be used to train a model, thus producing a weight matrix. The direction of the model depends on the -direction parameter.

-predict - Use the model to predict some data
Either this or the -train argument must be used. The -predict argument specifies that either the stimulus (reverse prediction) or the response (forward prediction) data will be predicted using the weight matrix trained on a previous use of this program. The direction of the prediction depends on the -direction parameter.

-stimulus filename - path to a stimulus dataset
-response filename - path to a response dataset
Mandatory. The stimulus and response datasets are specified as file paths. The default data format is a text file, with one time step per line, channels separated by whitespace. Time goes down the file. Alternatively, if the file name ends in ".mat" then the file is assumed to be a binary MATLAB file containing a single variable, the variable is named 'data'. Note when using this program to predict data, one of these two files is read and the other file is written, overwriting any data already in this file.

-model filename - path to YAML file describing the DNN
This parameter is mandatory and specifies a file containing the YAML description of the network. See the YAML manual/web page for more details of this file format and the file contents (http://deeplearning.net/software/pylearn2/yaml_tutorial/index.html).

-weights filename - path to a file for the model weights.
Optional in training, mandatory for predicting. A file that is used to store the DNN weights. The default is a text file, containing a rectangular matrix (N_inputs X N_outputs) containing the weight matrix for a single-layer DNN. If the weights filename ends in .pkl, then the file contains a Python pickle string.

-visual - Generate a graphical summary
Generate a graphical summary of the training and testing results. This is a file in Portable Network Graphics (PNG) format.

-context N - Additional temporal context for the prediction
The context argument specifies how many temporal frames should be provided to the DNN as context for its prediction. The meaning of this parameter depends on the direction of the prediction. If the direction is forward, predicting EEG from audio, then the context goes backward in time, to pick up older data from the audio. If the direction is reverse, predicting the stimulus, then N frames of sensor data after the current frame are added as context. See the figure on page 3 of this report for a graphical explanation.

-dir {forward/backward} - Direction of the prediction
This Parameter specifies whether the prediction goes forward from the stimulus to the response, or in reverse, from response to the stimulus. The default is reverse.

-valid filename - Which parts of the data are valid

When concatenating the data from multiple trials into one longer stimulus or response file, the temporal context is no longer valid across trial boundaries. The valid data is a binary signal, one value per time step that says whether the data at this frame should be used in the training and testing. [Peter, we should automatically extend any invalid data points forward or backward depending on the direction and the context.] DNNRegression automatically extends the region of validity based on the direction and the requested context.

-verbosity N - How much information o see during the training and testing
The default value of this parameter is 1. Reduce it to zero to see fewer status messages, and increase it to see more information.

Output Parameters

The output from this program depends on whether it is training or testing a model.

When training a model the only output is a weight matrix, indicating the best DNN parameters for the training data. This weight matrix can be used as input to the test phase.

When testing a model (provided as the weight matrix and the yaml file), the output is either a stimulus or a response file, depending on whether the predictions are reverse, or forward. The test output is written to the appropriate file, overwriting any contents there already.

Dependencies

This code uses PyLearn2, which is built on top of machine-learning system known as Theano. PyLearn2 is useful because it make it easy to describe a neural network, and thus it easier to modify. Building upon Theano adds the ability for the training calculations to be done on a GPU, if there is one available in your computer.

FindTRF

Find the temporal response function (TRF) connecting stimulus and response using a linear model.

Syntax

[g,pred] = FindTRF(stimulus, response, Dir, testdata, g, Lags, Method, K, doscale)

Description

Find the Temporal Response Function (TRF). This function uses linear regression to find a model (g) that linearly maps between stimulus and response. Because the stimulus causes the response, the best model should be causal, with a number of input frames in the past determining the output at the current frame. The stimulus might be the intensity of the presented audio signal, while the response is a multi-channel EEG signal measured while the subject listens to the audio. The basic process was used in O'Sullivan et al (2014)

This function can compute the response direction in either direction: from stimulus to response (forward) or from response to stimulus (backward). Forward mapping uses a stimulus in testdata to predict a neural response, while the backward mapping uses a response in testdata to reconstruct a stimulus. Although forward and backward mapping are mathematically identical operations, the two usually yield very different results. This is because of the different nature of the two signals (EEG vs audio). In the forward mapping we are predicting on EEG data while controlling for the noise in the audio. Backward mapping predicts on audio data while controlling for noise in the EEG.

An important part of obtaining good results is using regularization to avoid overfitting. EEG or audio data have higher-order statistical properties that are not relevant but which the decoder might still pick up during model fitting. The decoder usually have many parameters (lags x features x channels) which leads to systematic errors due to overfitting to random noise. Overfitting results in good results on training data but poor generalization to test data where the noise is different. Regularization puts constraints on the decoder weights to avoid this. FindTRF offers several options for regularizing the decoder described below.

The function both computes the model (if the stimulus and response variables are not empty) and uses a model to make predictions (if g and testdata are provided in the function call.) In either case, the model is returned. If testdata is supplied then the pred is returned with the appropriate prediction (depending on the direction).

Examples

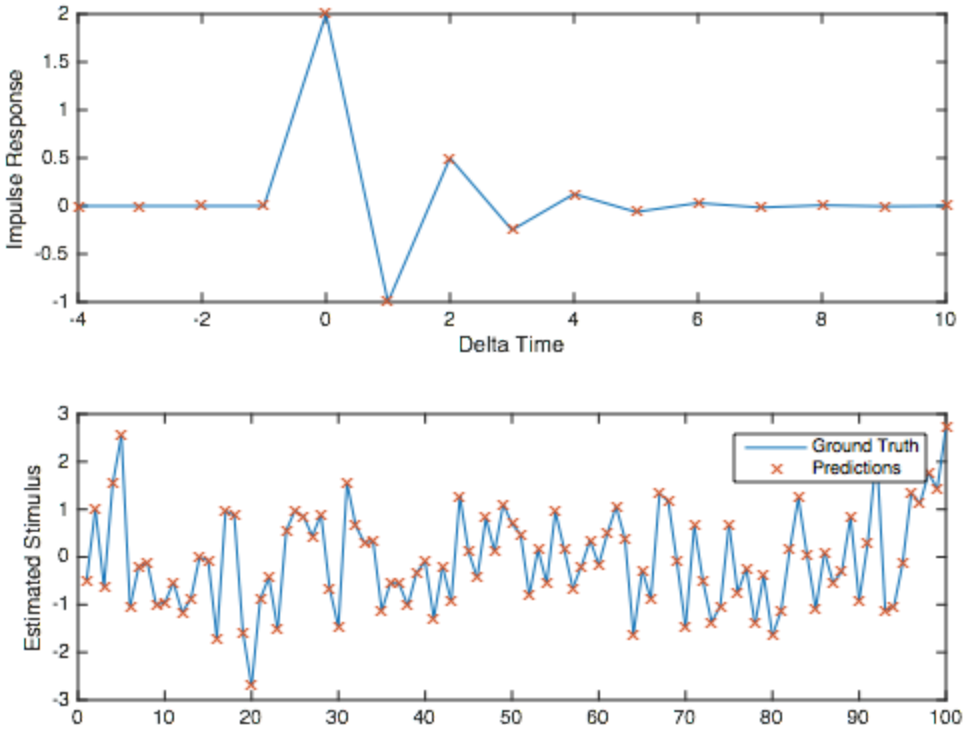
We provide a number of examples that demonstrate how this function behaves.

First we start with a simple two-channel simulation, where the EEG features depend on only two audio samples, the current and the preceding, from the first channel. In signal processing terms, this represents a two-sample FIR (finite impulse response) filter. The inverse filter is then an IIR (infinite impulse response) filter with geometrically decaying terms. The code to perform this test is shown below.

```
N = 10000;
StimulusData = randn(N,2);      % Generate a bunch of 2-channel data
lags = -4:10; Dir = -1;
```

```
% Create the response.  First channel is 0.5 times current time, plus
% 0.25 times previous time.  Second channel is all noise.
ResponseData = [StimulusData(:,1)*.5 + [StimulusData(2:end,1)*.25; 0] randn(N,1)];
TestingTimes = 1:1000;          % Use first part for testing
TrainingTimes = max(TestingTimes)+1:N;  % Use rest of data for training
TrainingStim = StimulusData(TrainingTimes,:);
TrainingResp = ResponseData(TrainingTimes,:);
TestingStim = StimulusData(TestingTimes, :);
TestingResp = ResponseData(TestingTimes, :);

% Now compute the response.
K = 1.0; doScale = 0;
[g,TestingStimPredict] = FindTRF(TrainingStim, TrainingResp, Dir, ...
    TestingResp, [], lags, 'none', K, doScale);

% Impulse response for feature 1, channel 1 should be powers of 2.
subplot(2,1,1);
plot(fliplr(lags), g(:,1,1), fliplr(lags), g(:,1,1), 'x');
xlabel('Delta Time');
ylabel('Impulse Response');
subplot(2,1,2);
t=1:100; plot(t, TestingStim(t,1), t, TestingStimPredict(t),'x')
legend('Ground Truth', 'Predictions'); ylabel('Estimated Stimulus');
```

This produces the output shown below. Note, the response before time 0 is flat. With no noise the simulated predictions are a perfect match to the original stimulus.



Here is a more realistic example of using the FindTRF function to estimate the attended and unattended audio. We use the CreateDemoSignals script to generate test signals, and then find the best linear model to predict the audio stimulus from the response. The code shown here performs this calculation.

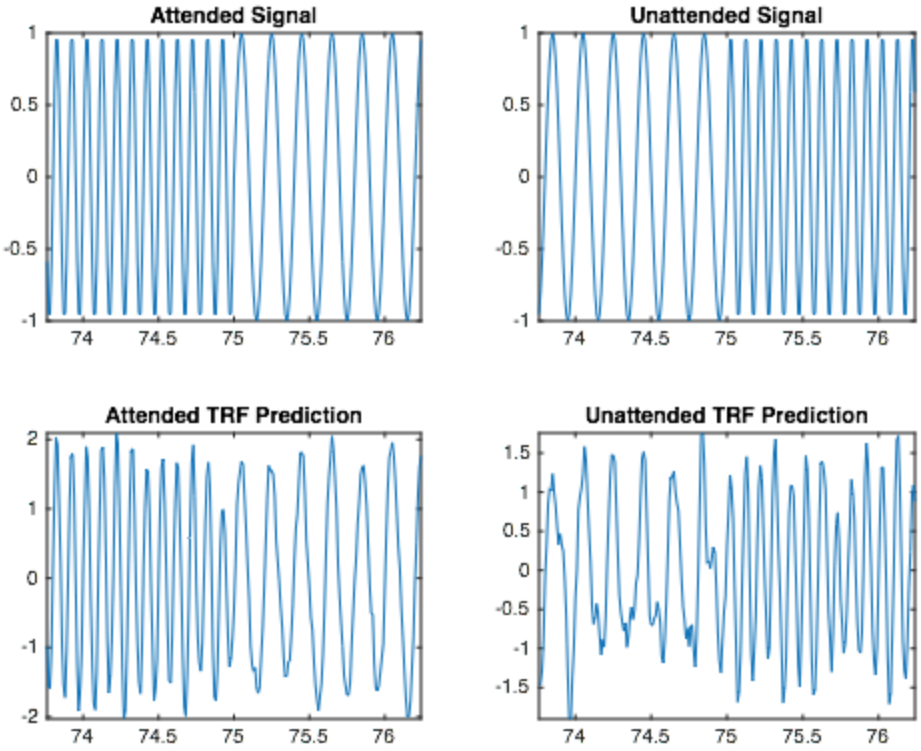
Use FindTRF to calculate the attended and unattended signal models. These are impulse responses that predict the input audio given an EEG response. We'll do it for both the attended and unattended signals. We just use a fraction of the input signals around the first attention switch. The attention switches every attentionDuration seconds, and we'll use trfTrainingWindow seconds on either side of one switch as training data.

```
trfTrainingWindow = attentionDuration / 20;
iTrain = find(recordingT > attentionDuration - trfTrainingWindow & ...
              recordingT < attentionDuration + trfTrainingWindow);
iTest = find(recordingT > 2*attentionDuration);

% Now calculate the models for the attended and unattended signals.
trfDirection = -1;
Lags = 0:round(1.5*impulseLength*fs);
Method = 'Shrinkage';
K = [];
doScale = 1;
attentionModel = FindTRF(attendedAudio(iTrain), response(iTrain, :), ...
    trfDirection, [], [], Lags, Method, K, doScale);
unattentionModel = FindTRF(unattendedAudio(iTrain), response(iTrain, :), ...
    trfDirection, [], [], Lags, Method);
```

Now calculate the predicted responses for the entire signal. (Note, this includes the already seen training data as part of the entire signal, but we only use the last half the signal (unseen by training) when computing the correlations.

```
[~, attendedPrediction] = FindTRF([], [], ...
    trfDirection, response, attentionModel, Lags, Method, K, doScale);
[~, unattendedPrediction] = FindTRF([], [], ...
    trfDirection, response, unattentionModel, Lags, Method, K, doScale);

ca = corrcoef([attendedAudio(iTest) attendedPrediction(iTest)]);
cu = corrcoef([unattendedAudio(iTest) unattendedPrediction(iTest)]);
fprintf('Attended correlation: %g, Unattended correlation: %g.\n', ...
    ca(1,2), cu(1,2));
%%
% Plot the predicted stimuli
clf
attentionSwitchPick = 3;
iPlot = find(recordingT>attentionSwitchPick*attentionDuration-trfTrainingWindow & ...
    recordingT < attentionSwitchPick*attentionDuration+trfTrainingWindow);
plot(recordingT(iPlot), [attendedPrediction(iPlot) unattendedPrediction(iPlot)]')
legend('Attended Signal', 'Unattended Signal');
axis tight
```

The results are shown here.



As predicted the unattended response is noisier because the signal-to-noise ratio is higher--the attended correlation: 0.988244, unattended correlation: 0.861533. The model used to predict these particular responses were based on +/- 1.25 of training data right around the first attention switch (at t=25s). We are plotting the response above around the 3rd attention switch (near t=75s).

Input Parameters

stimulus - Stimulus training set

A N_{time} x $N_{features}$ matrix of data that represents the data the subject heard. This data is used as training data and from this data the FindTRF function computes the best linear regression model. The sample rate of the stimulus and the response must be the same, and both matrices must have the same number of rows (samples).

response - Neural responses

A N_{time} x $N_{channels}$ matrix of data that represents the measured EEG/MEG responses of the subject. The sample rate of the stimulus and the response must be the same, and both matrices must have the same number of rows (samples).

Dir - specifies the direction of the mapping.

With Dir = 1 we compute the forward mapping (predict neural response from stimulus, or encoding). While with Dir = -1 we compute the backward mapping (predict stimulus from neural response, decoding).

testdata - test data used to run the model.
An optional N_{time} x $N_{features}$ matrix that is optional data used to test the model. If the data is empty then there are no output predictions. If it is supplied it represents the appropriate data for the desired direction. If direction is positive then supply the test stimulus data and the response is calculated. If direction is negative, then supply the measured response data, and the stimulus is calculated.

g - TRF model
A N_{lags} x $N_{features}$ x $N_{channels}$ matrix containing a previously computed TRF. If empty, it will be calculated from training data. Supply a pretrained g to predict based on testdata (be sure to specify the same lags as used in training). If both the stimulus and response are provided then g is always estimated. Singleton dimensions are removed.

Lags - The time lags to include in the model.
A vector of time lags from which to use in making a prediction. Are the time delays between stimulus and response in samples. The lags should always be positive if the data is causal, meaning that the response follows the stimulus. The time lags can be specified in one of three ways.

- Specify the [start end] as a two-element vector. The intermediate values are added.
- Specify a vector with a full vector of lags.
- The default value for this parameter is a vector from 0:100

See the LagGenerator function for more information about how this parameter is used.

Method - Regularization method to use.
Use this parameter to specify a regularization method to avoid overfitting. An additional input K specifies the regularization parameter(s). Possible methods are:

- 'Shrinkage', K (Default) (default K=0.2, range [0:1]).
  Shrinkage regularization shrinks the decoder weights by reducing the amount of off-diagonal sample variance. This is done by adding a penalty term during the decoder estimation. K(1) is a tuning parameter that balances small and large variance dimensions in the data. K=0 corresponds to the unregularized case, and K=1 corresponds to maximum regularization. See Blankertz et al. (2011) for details.
- 'Ridge', K (default K=[10,2], range [0:Inf] [0 1])
  This option uses Tikhonov (or L2) regularization which constrains the decoder weights to be closer to zero, whereby only the more important weights are non-zero. This is done by adding a penalty term during the decoder estimation. K(1) is a tuning parameter that controls the size of this penalty. K(1)=0 corresponds to the unregularized case. The weights shrinks as K(1) grows and the off-sample variance is constrained to be closer to zero. K(2) is a flag that determines whether to use the derivative of the penalty matrix by specifying its order (either 0 or 1). K(2)=0 performs normal ridge regression (no derivative). K(2)=1 (default) uses the first order derivative. This penalizes neighboring lags and yields more smooth solutions. K(2)=1 was suggested and used successfully by Lalor et al 2006.
- 'LRA', K (default K=0.99, range [0:1]) : low rank approximation
  This option uses principal component analysis to perform the regression in a lower-dimensional subspace of the data. This is done to reduce the effect of data dimensions with lower variance. The tolerance parameter K specifies the fraction of the total variance to preserve. K=1 corresponds to the unregularized case. Lower values

removes more lower-variance dimensions (K=0.9 preserves 90% of the variance). See Theunissen et al (2001) and David et al (2007) for details
- 'Lasso', K (default K=[.01,1]) : lasso / elastic net. Uses MATLAB's Lasso (and thus needs the Statistics toolbox). K(2) controls the degree of L1 penalty (elastic net). This option may result in long computation times
- 'None' : no regularization, use ordinary least-squares

doscale - Scale inputs to zero mean and set standard deviation to 1
Offsets in the data are not estimated by the regularized regression so the data must be properly scaled. This also pertains to predicted responses.

Output Parameters

g - the TRF function, a matrix
This is the decoder that is calculated from the input data. It has a size of N_{lags} x $N_{features}$ x $N_{channels}$. The decoder represents weights at the given time lags, features, and channels.

pred - The predicted output from testdata
If Dir is positive (forward model), then this is EEG/MEG data predicted from the stimulus data provided in testdata. If Dir is negative (backward model), then this is audio predictions based on the EEG/MEG data provided in testdata.

References

B. Blankertz, S. Lemm, M. Treder, S. Haufe, K. Müller (2011). Single-trial analysis and classification of ERP components - A tutorial. NeuroImage 56: 814-825. [PDF or CiteSeer]

S. David, N. Mesgarani, S. Shamma (2007). Estimating sparse spectro-temporal receptive fields with natural stimuli. *Network: Computation in Neural Systems* 18(3): 191-212. [PDF or PDF]

E. Lalor, B. Pearlmutter, R. Reilly, G. McDarby, J. Foxe (2006). The VESPA: a method for the rapid estimation of a visual evoked potential. NeuroImage 32(4): 1549-1561. [PDF or PDF]

J. O'Sullivan et al. (2014). Attentional selection in a cocktail party environment can be decoded from single-trial EEG. Cerebral Cortex, doi:10.1093/cercor/bht355. [PDF]

F. Theunissen, S. David, N. Singh, A. Hsu, W. Vinje, J. Gallant (2001). Estimating spatial temporal receptive fields of auditory and visual neurons from their responses to natural stimuli. Network Comput. Neural Syst 12:289–316. [PDF]

LagGenerator

Create additional context for a signal by adding copies of the data at different temporal lags.

Syntax

out = LagGenerator(in, lags)

Description

Shift a temporal response so as to create multiple copies of the original data, shifted in time so each row of the output response has several temporally shifted versions of the original data. Zero pads the first part of the output for positive lags and zero pads the last part for negative lags.

All functions in this toolbox assume that each column is a channel or a feature, and time goes down each column. This function adds copies of the data, additional columns for each lag.

Examples

For example the input

```
>> in=[(1:5)' (2:2:10)']

in =

     1     2
     2     4
     3     6
     4     8
     5    10
```

With a lag request for -1:1 produces a new array, the center two channels contains the original data. The left two columns are shifted backward in time, while the right two columns are shifted forward. In DSP terms, this is caused by adding a lag (z^{-1}) to the data.

```
>> LagGenerator(in, -1:1)

ans =

     2     4     1     2     0     0
     3     6     2     4     1     2
     4     8     3     6     2     4
     5    10     4     8     3     6
     0     0     5    10     4     8
```

Inputs

in - The input data, a N_{time} x $N_{features}$ array of data.

lags - The requested temporal changes, a vector of length N_{lags}. Negative lags correspond to moving the entire data array forward in time, while positive lags delay the data.

Outputs

out - The output data, augmented with an additional copy of the data for each time lag. The output array has a size of N_{time} x ($N_{features}$*N_{lags}).

resample_fft - Resample a signal

Syntax

newS = resample_fft(oldS, oldFs, newFs)

Description

Resample an (old) signal to a new sample rate. The original (old) sample rate is oldFs, the new sample rate is newFs. This code processes one- or two-dimensional signals, N_{frames} x $N_{channels}$ in size. Each column of the data is resampled independently.

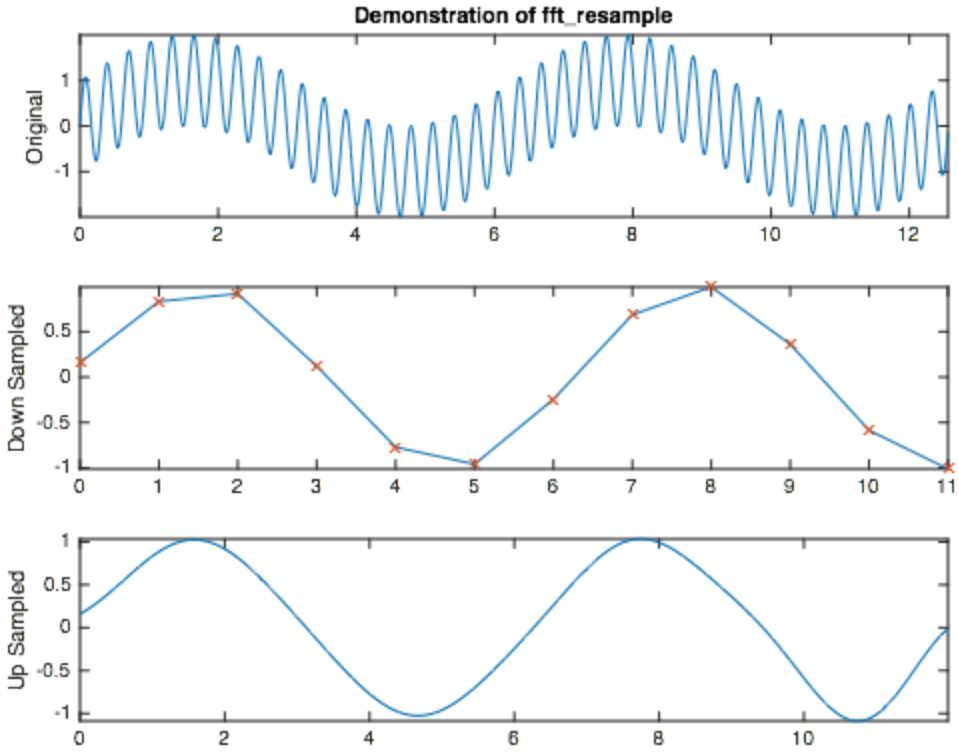
This code uses an FFT to perform interpolation and represents an optimal resampling. It does this by converting the signal to the FFT domain, adjusts the size of the FFT array, and then inverts the FFT to reconstruct the signal at the new sample rate. When upsampling, we add zeros to the high-frequency end of the spectrum. To down sample we remove the high-frequency components before inverting the FFT.

Note: this resampling is accomplished by zero-padding the original signal to twice its length, so we implement the filtering with a linear convolution. This means that there might be some errors at the beginning and end of the signal because of the explicit zero-padding.

Examples

We start with a sum of two sinusoids, sampled at 100Hz (top panel of the figure below). When we resample this signal at 1Hz, we must filter the signal at the Nyquist rate (½ Hz), which is done by removing high-frequency components of the FFT signal. When we reconstruct the down-sampled signal we have lost the high-frequency component because it is above the Nyquist rate. This is shown in the middle panel of the figure below and the actual sample points are indicated with an 'x.' Finally, we up-sample the down-sampled signal back to the original sample rate (100Hz). This restores the straight lines used in the middle panel to something like the original sinusoid.

```
% Create sum of two sinusoids (1/2pi Hz and 10/pi Hz) sampled at 100Hz
t = (0:.01:4*pi)';
s = sin(t) + sin(20*t);
subplot(3,1,1); plot(t,s);
axis tight; ylabel('Original');
title('Demonstration of fft\_resample');

% Resample the signal down to 1Hz.
s2 = resample_fft(s, 100, 1);
subplot(3,1,2); plot(t(1:100:end-99),s2, t(1:100:end-99),s2, 'x');
axis tight; ylabel('Down Sampled');

% Upsample the down-sampled signal back to the original 100Hz
s3 = resample_fft(s2, 1, 100);
subplot(3,1,3); plot(t(1:length(s3)), s3);
axis tight; ylabel('Up Sampled');
```



Input Parameters

oldS - Original (old) signal

The original signal to be resampled. This should be a matrix of data of size N_{frames} x $N_{channels}$. Each column is resampled to the new sample rate, adjusting the frequency content of the signal as appropriate.

oldFs - Original (old) sampling frequency

The sampling rate (Hz) of the original (old) signal.

newFs - New sampling frequency for the output data

The sampling rate (Hz) of the new signal.

Output Parameters

newS - The new resampled data

This matrix contains the new data. The size of this resampled output matrix is $N_{frames2}$ x $N_{channels}$, where $N_{frames2}$ is approximately equal to N_{frames}newFs/oldFs.

StateSpaceDecoder

Use a probabilistic model to decide to which sound the subject is attending.

Syntax

[q , q_L, q_U] = StateSpace(cors1, cors2, max_iterations, debug)

Description

This function estimates the probability of attending to speaker 1 in a two-speaker environment.

Starting with an estimate of the correlation between the response and each of the two original signals, this function uses a probabilistic state-space decoder to decide when the subject is listening to speaker 1 (and not speaker 2). With noise and other random processes, the correlation signals produced by any of the algorithms in this toolbox are noisy. This algorithm finds an optimal path through the state space (which speaker the subject is attending) that maximizes the probability of observations and minimizes the state transitions. The output is an estimate of the likelihood that speaker 1 has the subjects attention at each point in time.

The correlations used by this routine can come from any number of sources. In the example below, we use FindTRF to find an optimal linear decoder, and to predict the attended intensity signal. But we can also use the DNN approach, or even CCA, to estimate the needed correlations. Measures of correlation at a rate of about 2 to 4 Hz seem to work well.

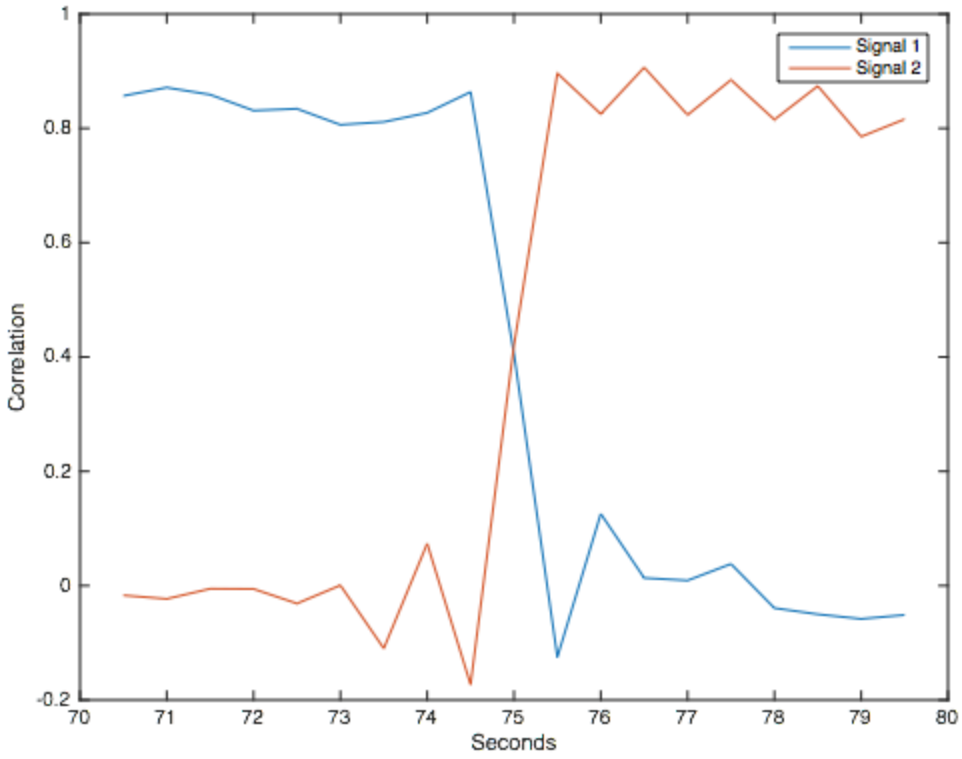
This routine is based upon principled approach to transform correlations (produced by routines like FindTRF) into a decision. An alternative approach, using an ad-hoc approximation to convert correlations into probabilities, is provided by the ViterbiSearch routine. The ViterbiSearch routine might be useful if you do not have the Optimization toolbox that this routine needs.

Examples

Using the simulated data from the CreateDemoSignals script, as well as all the demo scripts from the FindTRF documentation, we compute the correlations and apply the state space model.

```
% Create the correlation signals, and downsample them to 2Hz.
cors1 = attendedPrediction(1:length(audioS1)) .* audioS1;
cors2 = attendedPrediction(1:length(audioS2)) .* audioS2;

newFs = 2;
cors1d = resample_fft(cors1, fs, 2);
cors2d = resample_fft(cors2, fs, 2);
recording2T = recordingT(1:(fs/newFs):end-1);
i2Plot = find(recording2T>attentionSwitchPick*attentionDuration-4*trfTrainingWindow &
...
    recording2T < attentionSwitchPick*attentionDuration+4*trfTrainingWindow);

clf; plot(recording2T(i2Plot), [cors1d(i2Plot) cors2d(i2Plot)]);
legend('Signal 1', 'Signal 2'); xlabel('Seconds'); ylabel('Correlation');
```

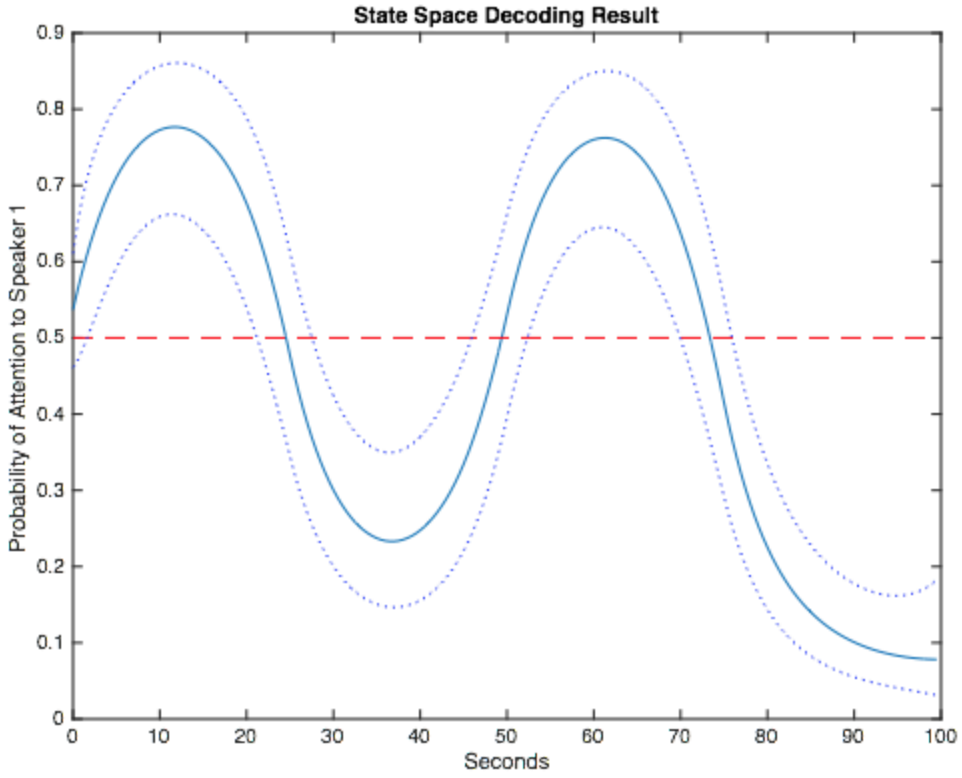
This produces correlation signals that look like this around the attention switch at 75s:



Now compute the state-space decoding, and plot the results. The dotted lines show the 70% confidence interval [defined how?], while the dashed line shows a threshold between attending to speaker 1 (above) and speaker 2 (below).

```
max_iterations = 100; debug = 1;
[q , q_L, q_U ] = StateSpace(cors1d, cors2d, max_iterations, debug);

%%
plot(recording2T, q, ...
    [min(recording2T) max(recording2T)], [0.5 0.5], 'r--', ...
    recording2T, q_L, 'b:', recording2T, q_U, 'b:');
xlabel('Seconds'); ylabel('Probability of Attention to Speaker 1');
title('State Space Decoding Result');
```

The graph below shows the decoding result over the entire 100-second long trial. The results around 25 seconds are not fair (or realistic) since this data was used for training the decoding model.



Input Parameters

[q , q_L, q_U] = StateSpace(cors1, cors2, max_iterations, debug)

cors1 - Correlation of the response to the first speaker

cors2 - Correlation of the response to the second speaker

max_iterations - Optional, default value is 100. Number of EM iterations to perform while deciding the optimal decoder.

debug - Optional, default value is 0 or off. Set to a non-zero value to see a plot of the state decoding probability develop over iteration.

Output Parameters

q - The state decoding probability. This is a probability that the subject is listening to speaker 1.

q_L - A lower bound estimate of the state decoding variable. At any one point in time, there is a 70% chance that the true decoded state is in between the values of q_L and q_U.

q_U - An upper bound estimate of the state decoding variable. At any one point in time, there is a 70% chance that the true decoded state is in between the values of q_L and q_U.

The CI parameter's default value is to compute the 70% confidence intervals around the estimation points, but it can be arbitrarily changed to obtain different confidence intervals.

Dependencies

This algorithm depends on the fsolve() routine in the MATLAB Optimization Toolbox. It will only run if you have access to this toolbox.

References

Akram, S., Simon, J. Z., Shamma, S. A., & Babadi, B. (2014). A State-Space Model for Decoding Auditory Attentional Modulation from MEG in a Competing-Speaker Environment. In *Advances in Neural Information Processing Systems*(pp. 460-468). [PDF].

Akram, S., A. Presacco, J. Z. Simon, S. A. Shamma and B. Babadi (2015), Robust Decoding of Selective Auditory Attention from MEG in a Competing-Speaker Environment via State-Space Modeling, *NeuroImage*. 2015 Oct 2. pii: S1053-8119(15)00870-8.

TFRegression.py (Not Present Yet)

Use Tensorflow (www.tensorflow.org) to estimate an inverse decoding model.

Syntax

python TFRegression.py TrainStimulusDataFile.mat TrainResponseDataFile.mat \
[TestStimulusDataFile.mat TestResponseDataFile.mat]

Description

This is preliminary Python code. It demonstrates how to use Google's Tensorflow machine-learning system to do regression. You must install Tensorflow (www.tensorflow.org) to use this routine.

Given training data (Nx1 stimulus, and NxC response data) this code uses Tensorflow to create a network and optimize it for the given training data. It reports the loss and the final correlation on the training data.

If the optional testing data files are included on the command line then these files are used to test the model.

The data files listed on the command line are Matlab data files, each containing a single array named 'data'.

ViterbiDecoder - Estimate the state sequence

Syntax

[decoded_states,all_probs] = ViterbiSearch(observations, start_p, trans_p, emit_f)

Description

From basic correlation signals, use a Viterbi decoder to make a decision about the direction of attention over time. At each time step one could simply use the larger correlation to indicate whether the subject is paying attention to speaker 1 or speaker 2, but nobody switches their attention at every frame. In the O'Sullivan EEG work the correlation values were averaged over many seconds of each trial and the winner was chosen based on the overall window. Instead we want to do something more subtle, deciding the state based on a probabilistic mixture of the strength of the correlation and the time since the last attentional switch. The code used in this toolbox is based on the Wikipedia algorithm.

In a Viterbi decoder one must be able to represent observations as probabilities. It is a bit difficult to do that with correlations. In our case for each time step we have N correlation measurements. For this demonstration we use a Gaussian centered at 1 with an arbitrary standard deviation to estimate the probability that a particular correlation says we are in this state. This is equivalent to saying that the probability of being in state i is only based on correlation #i, and is independent of the other correlations.

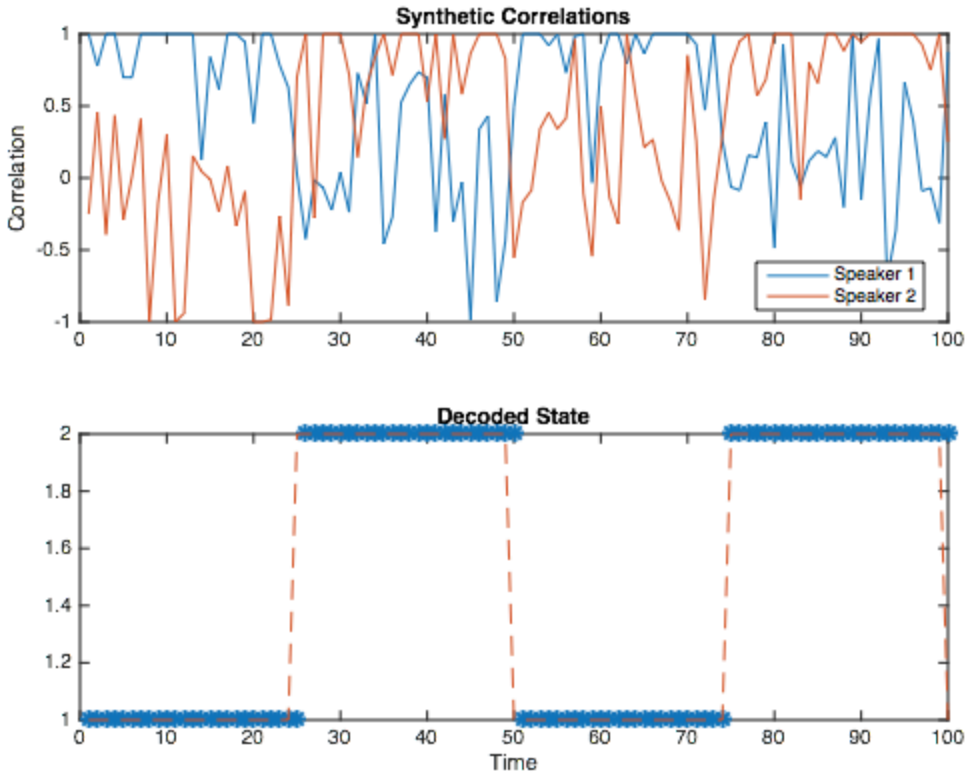
Here are some results. We are using the same kind of data used for the synthetic tests by CreateDemoSignal. One transition between speakers every 25 seconds. We generate artificial signal correlations. The emit_f parameter is a function that converts the observations (correlations) into a probability by assuming that the correlations are generated by Gaussian process centered at 0 (state or speaker 1) or 1 (state of speaker 2), and with a standard deviation of model_std. A Gaussian model is clearly not correct for this data, which is the rationale for the improved approach used in the StateDecoderModel function .

The state-transition matrix is a first-order Markov model that defines how likely one stays in the same (attentional) state. Here we simply use a parameter that is roughly equal to the expected state-duration time. Since it's a Poisson process, the mean duration is equal to 1/transition_probability.

```
% Continuous observation example. Based on correlation signals from
% an attention-switching model.
max_t = 100;
noise_std = 0.5;            % How much noise to add to correlations
model_std = 0.5;            % How much noise the model expects
T = (1:max_t)';             % A time scale, 1Hz sampling rate
start_p = [0.5 0.5];        % We don't know who is speaking first
trans_time = 25;            % 1/trans_time is approximation for transition probability
trans_p = [1-1/trans_time 1/trans_time; 1/trans_time 1-1/trans_time];
```

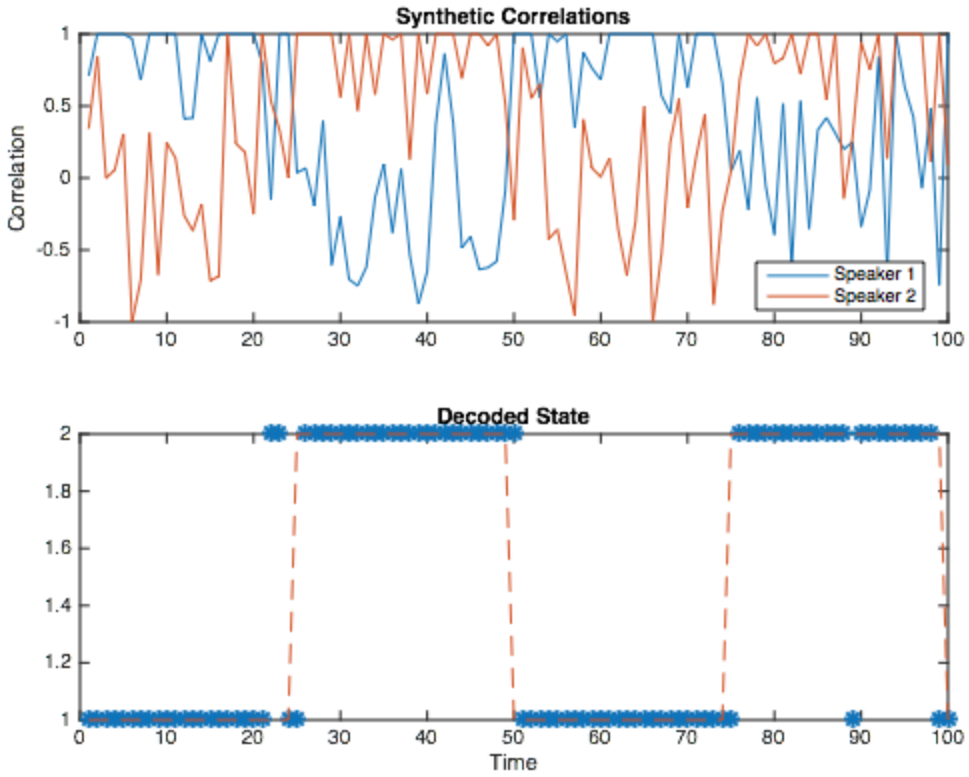
```
true_state = mod(floor(T/25),2)+1;
correlations = [randn(max_t,1)*noise_std + …
       (2-true_state) randn(max_t,1)*noise_std + (true_state-1)];
correlations = min(1, max(-1, correlations));    % Limit correlation values
subplot(2,1,1); plot(correlations); title('Synthetic Correlations');
legend('Speaker 1', 'Speaker 2', 'Location', 'best'); ylabel('Correlation');

% Probability of being in a particular state is based on the
% correlation of observations. Probability of state i is a Gaussian of
% width model_std, centered at 1. Thus the emission probability for
% state i is a function of element i of the observation (correlation)
% vector.
emit_f = @(obs_v, desired_state) exp(-((obs_v(desired_state)-1).^2/model_std.^2));
[decoded_states, all_probs] = ...
    ViterbiSearch(correlations, start_p, trans_p, emit_f);

subplot(2,1,2); hands = plot(T, decoded_states, '*', T, true_state, '--');
set(hands(1), 'LineWidth', 2)
title('Decoded State'); xlabel('Time');
```

Here is what it looks like when the expected duration in each state is 25 seconds.



There is a bit of “hangover” at the true state boundaries since the decoder wants more evidence before deciding.

And here is the result when you have an expected duration of 3 samples. There are more errors because the decoder is integrating less information before making a change.



Input Parameters

observations - A matrix of size N_{times} x N_{obs} which describes the observations made of the data. The number of observations at each time point is arbitrary, but must be what the emit_f function is expecting.

start_p - The initial probability of being in each of the N_{states}.

trans_p - The state transition probability matrix has size N_{states} x N_{states} and encodes the probability of moving from state i (the row index) to state j (the column index).

emit_f(obs_vec, s) - An emission probability function of two variables that takes an observation vector size (1 x N_{obs}) and returns the probability that we are in the given state (s). In the example above, the emit_f function calculates the probability that the observation of the correlation for state s is close enough to 1 by calculating N(1, model_std).

Output Parameters

decoded_states - A vector of length N_{times} indicating the most likely state at each time.

all_probs - A matrix of size Ntimes x Nstates that indicates the probability that we are in each state. (The decoded states are calculated from this matrix by tracing backwards from the last frame, looking for the highest probability at each time frame.

Appendix 1: Data

We are distributing two different datasets with this toolbox. Neither is very large, but they can be used to establish that this toolbox is producing sensible results. Hopefully these datasets, and the tools in this toolbox, serve as a benchmark for future work.

The EEG dataset was collected in Telluride. The dataset is a simple single-speaker speech-decoding experiment that can be used for measuring speech decoding correlations.

The second dataset is MEG data, collected by Jonathan Simon's lab.

Speech Decoding Experiment - EEG Demo Data

This data was collected as part of demonstration in Telluride Colorado during July 2015. The basic data is essentially speech in, EEG out. The task is to predict the speech or the EEG signal from the other data.

A twist to the experiment was that the EEG signals are recorded from 4 subjects simultaneously. Subjects were viewing a video of a talking face. We recorded EEG data from all 4 subjects using 16 electrodes per subject. The ground from each of the 4 subjects were wired together, and then connected via a single wire to the active ground on the BrainVision amplifier.

Preprocessing

- EEG
    - Highpass FIR filter at 0.1 Hz with a filter order of 200
    - Demean
    - Eye blink removal using NoiseTools
    - Robust PCA
    - Resample to 128 Hz (using resample_fft)
    - Bandpass FIR filter between 2-8 Hz with a filter order of 128
    - Trial epoching and alignment with start of audio
- Audio
    - Compute Hilbert envelope
    - Resample to 128 Hz (using resample_fft)
    - Lowpass FIR filter at 8 Hz with a filter order of 128

Data format

The data is stored in a large Matlab structure with a number of fields. The most important ones are:

- eeg: A cell array with the EEG recordings from 32 different trials.
- wav: A cell array with the intensity profile of the 4 different stimuli.
- data.event.eeg(i).value: The stimulus number for eeg trial i.
- data.fsample: The sampling rate for the EEG and wav data.

Speech Attention Experiment - Two Speaker MEG Data

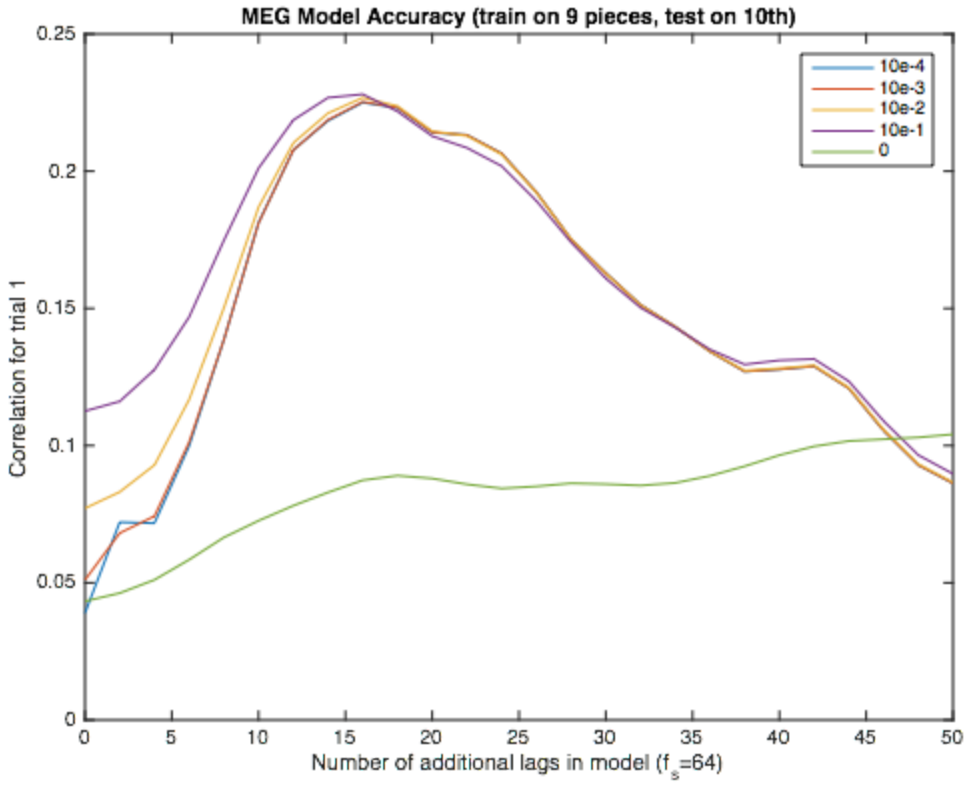
This MEG data was supplied by Jonathan Simon (U of Maryland).

There are four files, representing the response to four different conditions:

1. Male speaker alone
2. Female speaker alone
3. Simultaneous male and female speakers, subject attends to the male.
4. Simultaneous male and female speakers, subject attends to the female.

Each file contains the data from 3 separate 60 second trials. All the data was recorded from the same subject.

The plot below shows the decoding accuracy from the male data, single speaker. We trained on 9/10th of the first trial, testing on the last 1/10th. Repeat 10 times.



This data was used in the following study: Akram, S., A. Presacco, J. Z. Simon, S. A. Shamma and B. Babadi. Robust Decoding of Selective Auditory Attention from MEG in a Competing-Speaker Environment via State-Space Modeling, *NeuroImage*. 2015 Oct 2. pii: S1053-8119(15)00870-8.