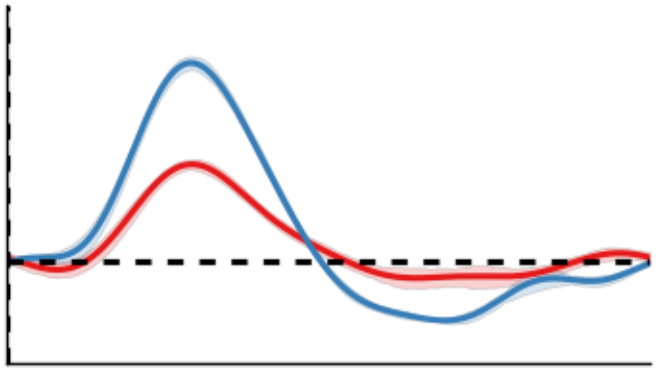
nideconv Documentation

Tomas Knapen

Jul 09, 2018

Getting started



Nideconv is an easy-to use Python library that can perform automated deconvolution of (primarily) slowly fluctuating (proxies of) neural signals like pupil size and BOLD fMRI. It was developed at the Vrije Universiteit and the Spinoza Centre for Neuroimaging by Gilles de Hollander and Tomas Knapen.

CHAPTER 1

Installing nideconv

1.1 From PyPi

When the first Beta of nideconv will be released, it will be possible to install nideconv from pip:

```
pip install nideconv
```

1.2 From Github

Right now you can clone the main branch of nideconv using git

```
git clone https://github.com/VU-Cog-Sci/nideconv
```

Or download and unpack the zip file from Github under **Clone and download -> Download ZIP**

Then go to the directory in which the package was cloned

```
cd nideconv
```

and install the Python package

```
python setup.py install
```

Note: Click *here* to download the full example code

CHAPTER 2

What is Deconvolution?

Neuroscientists (amongst others) are often interested in time series that are derived from neural activity, such as fMRI BOLD and pupil dilation. However, for some classes of data (notably, pupil dilation and fMRI BOLD), neural activity gets temporally delayed and dispersed. This means that if the time series is related to some behavioral events that are close together in time, these event-related responses will contaminate each other.

```
from nideconv import simulate
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('white')
sns.set_context('notebook')
```

2.1 Simulate data

Here we simulate fMRI data with a "cue - stimulus" design. There are four cues and stimulus pairs. The cue is always followed by a stimulus in 1, 2, 3, or 4 seconds. The cue leads to a small de-activation (0.5 % signal change), the stimulus a slight activation (1.0 % signal change)

```
cue_onsets = [5, 15, 25, 35]
stim_onsets = [6, 17, 28, 39]

cue_pars = {'name':'cue',
            'mu_group':-.5,
            'std_group':0,
            'onsets':cue_onsets}

stim_pars = {'name':'stim',
             'mu_group':1,
             'std_group':0,
             'onsets':stim_onsets}

conditions = [cue_pars,
```

(continues on next page)

(continued from previous page)

```
                stim_pars]

data, onsets, parameters = simulate.simulate_fmri_experiment(conditions,
                                                             run_duration=60,
                                                             noise_level=0.1)
```

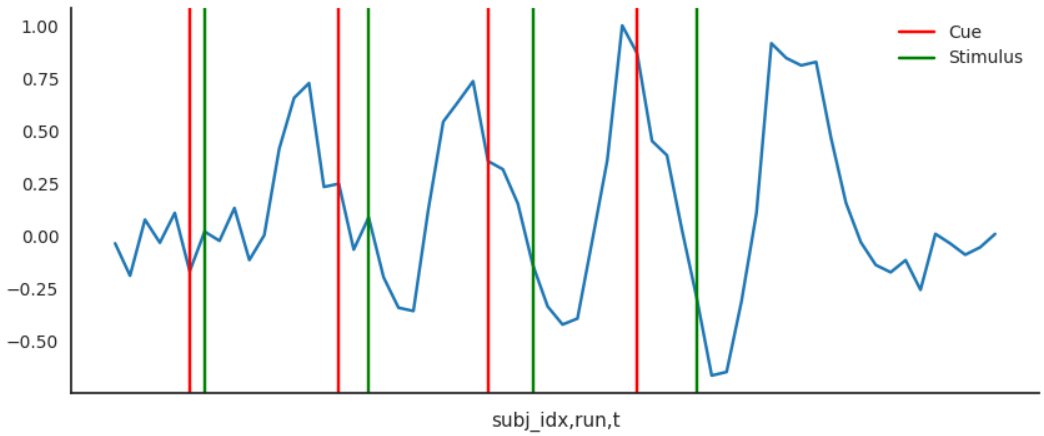
2.2 Plot simulated data

```
data.plot()
sns.despine()

for onset in cue_onsets:
    l1 =plt.axvline(onset, c='r')

for onset in stim_onsets:
    l2 =plt.axvline(onset, c='g')

plt.legend([l1, l2], ['Cue', 'Stimulus'])
plt.gcf().set_size_inches(10, 4)
```



2.3 Underlying data-generating model

Because we simulated the data, we know that the event-related responses should exactly follow the *canonical Hemodynamic Response Function* [1]_are

```
from nideconv.utils import double_gamma_with_d
import numpy as np

plt.figure(figsize=(12, 4))

t = np.linspace(0, 20, 100)
ax1 = plt.subplot(121)
plt.title('Ground truth cue-related response')
```

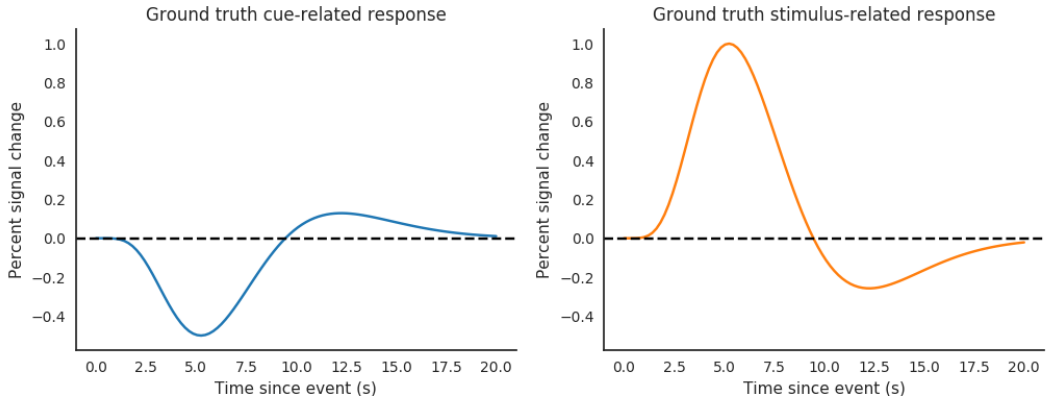
(continues on next page)

(continued from previous page)

```
plt.plot(t, double_gamma_with_d(t) * -.5,
         c=sns.color_palette()[0])
plt.xlabel('Time since event (s)')
plt.ylabel('Percent signal change')
plt.axhline(0, c='k', ls='--')

plt.subplot(122, sharey=ax1)
plt.title('Ground truth stimulus-related response')
plt.plot(t, double_gamma_with_d(t),
         c=sns.color_palette()[1])
plt.axhline(0, c='k', ls='--')
plt.xlabel('Time since event (s)')
plt.ylabel('Percent signal change')

sns.despine()
```



2.4 Naive approach: epoched averaging

A simple approach that is more appropriate for fast electrphysiological signals like EEG and MEG would be to select little chunks of the time series, corresponding to the onset of our events-of-interest and the first 20 seconds ("epoching").

nideconv

2.5 References

Total running time of the script: (0 minutes 1.155 seconds)

CHAPTER 3

Usage examples

Note: Click *here* to download the full example code

3.1 Cortical depth estimation from MGDM segmentation

This example shows how to obtain a cortical laminar depth representation from blabla.

This is like super cool.

```
import response_fytter
```

Total running time of the script: (0 minutes 0.000 seconds)

CHAPTER 4

Deconvolution of a single time series

4.1 ResponseFitter

class nideconv.**ResponseFitter**(*input_signal*, *sample_rate*, *oversample_design_matrix=20*, *add_intercept=True*, ***kwargs*)

ResponseFitter takes an input signal and performs deconvolution on it. To do this, it requires event times, and possible covariates. ResponseFytter can, for each event_type, use different basis function sets, see Event.

Methods

add_confounds(name, confound)	Add a timeseries or set of timeseries to the general design matrix as a confound
add_event(event_name[, onset_times, ...])	create design matrix for a given event_type.
get_epochs(onsets, interval[, ...])	Return a matrix corresponding to specific onsets, within a given interval.
get_rsq()	calculate the rsq of a given fit.
predict_from_design_matrix([X])	predict a signal given a design matrix.
regress([type, cv, alphas, store_residuals])	Regress a created design matrix on the input_data.
ridge_regress([cv, alphas, store_residuals])	run CV ridge regression instead of ols fit.

add_intercept	
get_residuals	
get_time_to_peak	
get_timecourses	
plot_timecourses	

add_confounds(*name*, *confound*)

Add a timeseries or set of timeseries to the general design matrix as a confound

Parameters

confound [array] Confound of (n_timepoints) or (n_timepoints, n_confounds)

add_event(*event_name, onset_times=None, basis_set='fir', interval=[0, 10], n_regressors=None, durations=None, covariates=None, **kwargs*)
create design matrix for a given event_type.

Parameters

event_name [string] Name of the event_type, used as key to lookup this event_type's characteristics

****kwargs** [dict] keyward arguments to be internalized by the generated and internalized Event object. Needs to consist of the necessary arguments to create an Event object, see Event constructor method.

get_epochs(*onsets*, *interval*, *remove_incomplete_epochs=True*)
Return a matrix corresponding to specific onsets, within a given interval. Matrix size is (n_onsets, n_timepoints_within_interval).

Note that any events that are in the ResponseFitter-object will be regressed out before calculating the epochs.

get_rsq()
calculate the rsq of a given fit. calls predict_from_design_matrix to predict the signal that has been fit

predict_from_design_matrix(*X=None*)
predict a signal given a design matrix. Requires regression to have been run.

Parameters

X [np.array, (timepoints, n_regressors)] the design matrix for which to predict data.

regress(*type='ols'*, *cv=20*, *alphas=None*, *store_residuals=False*)
Regress a created design matrix on the input_data.

Creates internal variables betas, residuals, rank and s. The beta values are then injected into the event_type objects the ResponseFitter contains.

Parameters

type [string, optional] the type of fit to be done. Options are 'ols' for np.linalg.lstsq, 'ridge' for CV ridge regression.

ridge_regress(*cv=20*, *alphas=None*, *store_residuals=False*)
run CV ridge regression instead of ols fit. Uses sklearn's RidgeCV class

Parameters

cv [int] number of cross-validation folds

alphas [np.array] the alpha/lambda values to try out in the CV ridge regression

CHAPTER 5

Deconvolution of a group of timeseries (level 2 analysis)

In most neuroscience studies, you want to analyze a group of subjects, each of which has one or multiple timeseries (often called ‘runs‘ in fMRI).

To analyze those, you can use the GroupResponseFitter

5.1 GroupResponseFitter

class nideconv.**GroupResponseFitter**(*timeseries*, *behavior*, *input_sample_rate*, *oversample_design_matrix=20*, *confounds=None*, *concatenate_runs=True*, **args*, ***kwargs*)

Can fit a group of individual subjects and/or runs using a high-level interface.

Methods

add_event	
fit	
get_conditionwise_timecourses	
get_subjectwise_timecourses	
get_timecourses	
plot_groupwise_timecourses	

CHAPTER 6

Voxelwise deconvolution on Nifti Images

6.1 NiftiResponseFytter

class nideconv.nifti.**NiftiResponseFitter**(*func_img*, *sample_rate*, *mask=None*, *oversample_design_matrix=20*, *add_intercept=True*, *detrend=False*, *standardize=False*, *confounds_for_extraction=None*, *memory=None*, ***kwargs*)

Methods

add_confounds(name, confound)	Add a timeseries or set of timeseries to the general design matrix as a confound
add_event(event_name[, onset_times, ...])	create design matrix for a given event_type.
get_epochs(onsets, interval[, ...])	Return a matrix corresponding to specific onsets, within a given interval.
get_rsq()	calculate the rsq of a given fit.
regress([type, cv, alphas, store_residuals])	Regress a created design matrix on the input_data.

add_intercept	
get_residuals	
get_time_to_peak	
get_timecourses	
plot_timecourses	
predict_from_design_matrix	
ridge_regress	

add_confounds(*name*, *confound*)

Add a timeseries or set of timeseries to the general design matrix as a confound

Parameters

confound [array] Confound of (n_timepoints) or (n_timepoints, n_confounds)

add_event(*event_name, onset_times=None, basis_set='fir', interval=[0, 10], n_regressors=None, durations=None, covariates=None, **kwargs*)
create design matrix for a given event_type.

Parameters

event_name [string] Name of the event_type, used as key to lookup this event_type's characteristics

****kwargs** [dict] keyward arguments to be internalized by the generated and internalized Event object. Needs to consist of the necessary arguments to create an Event object, see Event constructor method.

get_epochs(*onsets*, *interval*, *remove_incomplete_epochs=True*)
Return a matrix corresponding to specific onsets, within a given interval. Matrix size is (n_onsets, n_timepoints_within_interval).

Note that any events that are in the ResponseFitter-object will be regressed out before calculating the epochs.

get_rsq()
calculate the rsq of a given fit. calls predict_from_design_matrix to predict the signal that has been fit

predict_from_design_matrix(*X=None*)
predict a signal given a design matrix. Requires regression to have been run.

Parameters

X [np.array, (timepoints, n_regressors)] the design matrix for which to predict data.

regress(*type='ols'*, *cv=20*, *alphas=None*, *store_residuals=False*)
Regress a created design matrix on the input_data.

Creates internal variables betas, residuals, rank and s. The beta values are then injected into the event_type objects the ResponseFitter contains.

Parameters

type [string, optional] the type of fit to be done. Options are 'ols' for np.linalg.lstsq, 'ridge' for CV ridge regression.

ridge_regress(**args*, ***kwargs*)
run CV ridge regression instead of ols fit. Uses sklearn's RidgeCV class

Parameters

cv [int] number of cross-validation folds

alphas [np.array] the alpha/lambda values to try out in the CV ridge regression

CHAPTER 7

Simulate data

7.1 Simulate fMRI data

nideconv.simulate.**simulate_fmri_experiment**(*conditions=None*, *TR=1.0*, *n_subjects=1*, *n_runs=1*, *n_trials=40*, *run_duration=300*, *noise_level=1*, *n_rois=1*, *oversample=20*, *kernel='double_hrf'*, *kernel_pars={}*)

Simulates an fMRI experiment and returns a pandas DataFrame with the resulting time series in an analysis-ready format.

By default a single run of a single subject is simulated, but a larger number of subjects, runs, and ROIs can also be simulated.

Parameters

conditions [list of dictionaries or *None*] Can be used to customize different conditions. Every conditions is represented as a dictionary in this list and has the following form:

```
[{'name':'Condition A',
  'mu_group':1,
  'std_group':0.1},
  {'name':'Condition B',
  'mu_group':1,
  'std_group':0.1}]
```

mu_group indicates the mean amplitude of the response to this condition across subjects. *std_group* indicates the standard deviation of this amplitude across subjects.

Potentially, customized onsets can also be used as follows:

```
{'name':'Condition A',
 'mu_group':1,
 'std_group':0.1
 'onsets':[10, 20, 30]}
```

TR [float] Indicates the time between volume acquistisions in seconds (Inverse of the sample rate).

n_subjects [int] Number of subjects.

n_runs [int] Number of runs *per subject*.

n_trials [int] Number of trials *per condition per run*. Only used when no custom onsets are provided (see *conditions*).

run_duration [float] Duration of a single run in seconds.

noise_level [float] Standard deviation of Gaussian noise added to time series.

n_rois [int] Number of regions-of-interest. Determines the number of columns of *data*.

Returns

data [DataFrame] Contains simulated time series with subj_idx, run and time (s) as index. Columns correspond to different ROIs

onsets [DataFrame] Contains used event onsets with subj_idx, run and trial type as index.

parameters [DataFrame] Contains parameters (amplitude) of the different event type.

Other Parameters

oversample [int] Determines how many times the kernel is oversampled before convolution. Should usually not be changed.

kernel [str] Sets which kernel to use for response function. Currently only '*double_hrf*' can be used.

Examples

By default, *simulate_fmri_experiment* simulates a 5 minute run with 40 trials for one subject

```
>>> data, onsets, params = simulate_fmri_experiment()
>>> print(data.head())
                     area 1
subj_idx run t
1        1   0.0 -1.280023
             1.0  0.908086
             2.0  0.850847
             3.0 -1.010475
             4.0 -0.299650
>>> print(data.onsets)
                                onset
subj_idx run trial_type
1        1   A            94.317361
             A           106.547084
             A           198.175115
             A            34.941112
             A            31.323272
>>> print(params)
                         amplitude
subj_idx trial_type
1        A                 1.0
         B                 2.0
```

With n_subjects we can increase the number of subjects

```
>>> data, onsets, params = simulate_fmri_experiment(n_subjects=20)
>>> data.index.get_level_values('subj_idx').unique()
Int64Index([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
            20],
           dtype='int64', name='subj_idx')
```

Index