In this tutorial, we'll see how the Python library `nilearn` allows us to easily perform machine learning analyses with neuroimaging data, specifically MRI and fMRI.

You may notice that the name `nilearn` is reminiscent of `scikit-learn` (https://scikit-learn.org), a popular Python library for machine learning. This is no accident! Nilearn and scikit-learn were created by the same team, and nilearn is designed to bring machine **LEARN**ing to the NeuroImaging (**NI**) domain.

In [1]:
```
import warnings
warnings.filterwarnings("ignore")
```

For our purposes, what's most interesting is the structure of this data set. That is, the data is structured in a tabular format, with pre-extracted features of interest. This makes it easier to consider issues such as: which features would we like to predict? Or, how should we handle cross-validation?

But if we're starting with neuroimaging data, how can create this kind of structured representation?

Neuroimaging data

Neuroimaging data does not have a tabular structure. Instead, it has both spatial and temporal dependencies between successive data points. That is, knowing *where* and *when* something was measured tells you information about the surrounding data points.

We also know that neuroimaging data contains a lot of noise that's not blood-oxygen-level dependent (BOLD), such as head motion. Since we don't think that these other noise sources are related to neuronal firing, we often need to consider how we can make sure that our analyses are not driven by these noise sources.

These are all considerations that most machine learning software libraries are not designed to deal with! Nilearn therefore plays a crucial role in bringing machine learning concepts to the neuroimaging domain.

To get a sense of the problem, the quickest method is to just look at some data. You may have your own data locally that you'd like to work with. Nilearn also provides access to several neuroimaging data sets and atlases (we'll talk about these a bit later).

These data sets (and atlases) are only accessible because research groups chose to make their collected data publicly available. We owe them a huge thank you for this! The data set we'll use today was originally collected by Rebecca Saxe (https://mcgovern.mit.edu/profile/rebecca-saxe/)'s group at MIT and hosted on OpenNeuro (https://openneuro.org/datasets/ds000228/versions/1.1.0).

The nilearn team preprocessed the data set with fMRIPrep (https://fmriprep.readthedocs.io) and downsampled it to a lower resolution, so it'd be easier to work with. We can learn a lot about this data set directly from the Nilearn documentation (https://nilearn.github.io/modules/generated/nilearn.datasets.fetch_development_fmri.html). For example, we can see that this data set contains over 150 children and adults watching a short Pixar film. Let's download the first 30 participants.

In [2]:
```
from nilearn import datasets

development_dataset = datasets.fetch_development_fmri(n_subjects=30)
```

Now, this `development_dataset` object has several attributes which provide access to the relevant information. For example, `development_dataset.phenotypic` provides access to information about the participants, such as whether they were children or adults. We can use `development_dataset.func` to access the functional MRI (fMRI) data.

Let's use the nibabel library (https://nipy.org/nibabel/) to learn a little bit about this data:

In [3]:
```
import nibabel as nib

img = nib.load(development_dataset.func[0])
img.shape
```

Out[3]:
```
(50, 59, 50, 168)
```

This means that there are 168 volumes, each with a 3D structure of (50, 59, 50).

Getting into the data: subsetting and viewing

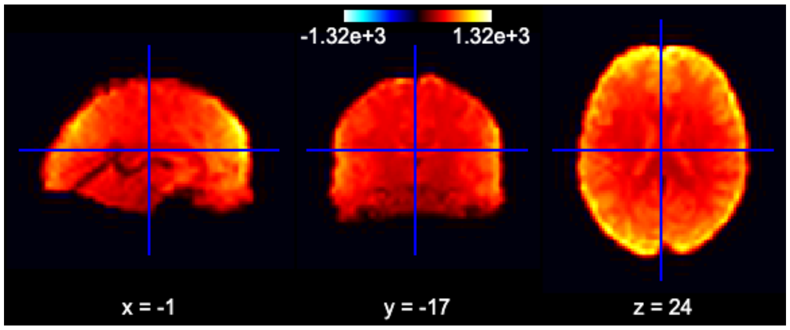
Nilearn also provides many methods for plotting this kind of data. For example, we can use `nilearn.plotting.view_img` (https://nilearn.github.io/modules/generated/nilearn.plotting.view_img.html) to launch at interactive viewer. Because each fMRI run is a 4D time series (three spatial dimensions plus time), we'll also need to subset the data when we plot it, so that we can look at a single 3D image. Nilearn provides (at least) two ways to do this: with `nilearn.image.index_img` (https://nilearn.github.io/modules/generated/nilearn.image.index_img.html), which allows us to index a particular frame--or several frames--of a time series, and `nilearn.image.mean_img` (https://nilearn.github.io/modules/generated/nilearn.image.mean_img.html), which allows us to take the mean 3D image over time.

Putting these together, we can interatively view the mean image of the first participant using:

In [4]:
```
import matplotlib.pyplot as plt
from nilearn import image
from nilearn import plotting

mean_image = image.mean_img(development_dataset.func[0])
plotting.view_img(mean_image, threshold=None)
```

Out[4]:



Extracting signal from fMRI volumes

As you can see, this data is decidedly not tabular! What we'd like is to extract and transform meaningful features from this data, and store it in a format that we can easily work with. Importantly, we *could* work with the full time series directly. But we often want to reduce the dimensionality of our data in a structured way. That is, we may only want to consider signal within certain learned or pre-defined regions of interest (ROIs), and when taking into account known sources of noise. To do this, we'll use nilearn's Masker objects. What are the masker objects ? First, let's think about what masking fMRI data is doing:

```
---
height: 350px
name: masking
---
Masking fMRI data.
```

Essentially, we can imagine overlaying a 3D grid on an image. Then, our mask tells us which cubes or “voxels” (like 3D pixels) to sample from. Since our Nifti images are 4D files, we can’t overlay a single grid – instead, we use a series of 3D grids (one for each volume in the 4D file), so we can get a measurement for each voxel at each timepoint.

Masker objects allow us to apply these masks! To start, we need to define a mask (or masks) that we'd like to apply. This could correspond to one or many regions of interest. Nilearn provides methods to define your own functional parcellation (using clustering algorithms such as *k-means*), and it also provides access to other atlases that have previously been defined by researchers.

Choosing regions of interest

In this tutorial, we'll use the MSDL (multi-subject dictionary learning; {cite} `Varoquaux_2011`) atlas, which defines a set of *probabilistic* ROIs across the brain.

In [5]:
```
import numpy as np

msdl_atlas = datasets.fetch_atlas_msdl()

msdl_coords = msdl_atlas.region_coords
n_regions = len(msdl_coords)

print(f'MSDL has {n_regions} ROIs, part of the following networks :\n{np.unique(msdl_atlas.networks)}.')
```

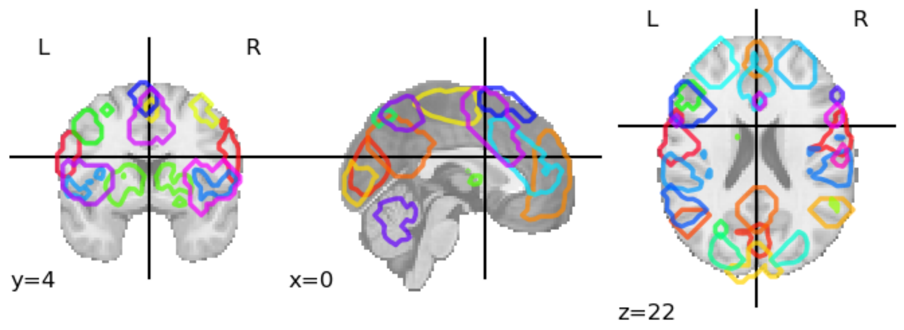
```
MSDL has 39 ROIs, part of the following networks :
['Ant IPS' 'Aud' 'Basal' 'Cereb' 'Cing-Ins' 'D Att' 'DMN' 'Dors PCC'
 'L V Att' 'Language' 'Motor' 'Occ post' 'R V Att' 'Salience' 'Striate'
 'Temporal' 'Vis Sec'].
```

Nilearn ships with several atlases commonly used in the field, including the Schaefer atlas and the Harvard-Oxford atlas.

It also provides us with easy ways to view these atlases directly. Because MSDL is a probabilistic atlas, we can view it using:

In [6]:
```
plotting.plot_prob_atlas(msdl_atlas.maps)
```

Out[6]:
```
<nilearn.plotting.displays._slicers.OrthoSlicer at 0x7fc5fd2e3cd0>
```



A quick side-note on the NiftiMasker zoo

We'd like to supply these ROIs to a Masker object. All Masker objects share the same basic structure and functionality, but each is designed to work with a different kind of ROI.

The canonical `nilearn.input_data.NiftiMasker` (https://nilearn.github.io/modules/generated/nilearn.input_data.NiftiMasker.html) works well if we want to apply a single mask to the data, like a single region of interest.

But what if we actually have several ROIs that we'd like to apply to the data all at once? If these ROIs are non-overlapping, as in "hard" or deterministic parcellations, then we can use `nilearn.input_data.NiftiLabelsMasker` (https://nilearn.github.io/modules/generated/nilearn.input_data.NiftiLabelsMasker.html). Because we're working with "soft" or probabilistic ROIs, we can instead supply these ROIs to `nilearn.input_data.NiftiMapsMasker` (https://nilearn.github.io/modules/generated/nilearn.input_data.NiftiMapsMasker.html).

Applying a Masker object

We can supply our MSDL atlas-defined ROIs to the `NiftiMapsMasker` object, along with resampling, filtering, and detrending parameters.

In [7]:
```
from nilearn import input_data

masker = input_data.NiftiMapsMasker(
    msdl_atlas.maps, resampling_target="data",
    t_r=2, detrend=True,
    low_pass=0.1, high_pass=0.01).fit()
```

One thing you might notice from the above code is that immediately after defining the masker object, we call the `.fit` method on it. This method may look familiar if you've previously worked with scikit-learn estimators!

You'll note that we're not supplying any data to this `.fit` method; that's because we're fitting the Masker to the provided ROIs, rather than to our data.

Dimensions, dimensions

We can use this fitted masker to transform our data.

In [8]:
```
roi_time_series = masker.transform(development_dataset.func[0])
roi_time_series.shape
```

Out[8]:
```
(168, 39)
```

If you'll remember, when we first looked at the data its original dimensions were (50, 59, 50, 168). Now, it has a shape of (168, 39). What happened?!

Rather than providing information on every voxel within our original 3D grid, we're now only considering those voxels that fall in our 39 regions of interest provided by the MSDL atlas and aggregating across voxels within those ROIS. This reduces each 3D volume from a dimensionality of (50, 59, 50) to just 39, for our 39 provided ROIs.

You'll also see that the "dimensions flipped;" that is, that we've transposed the matrix such that time is now the first rather than second dimension. This follows the scikit-learn convention that rows in a data matrix are *samples*, and columns in a data matrix are *features*.

```
---
height: 250px
name: samples-features
---
The scikit-learn conventions for feature and target matrices.
From Jake VanderPlas's _Python Data Science Handbook_.
```

One of the nice things about working with nilearn is that it will impose this convention for you, so you don't accidentally flip your dimensions when using a scikit-learn model!

Creating and viewing a connectome

The simplest and most commonly used kind of functional connectivity is pairwise correlation between ROIs. We can estimate it using `nilearn.connectome.ConnectivityMeasure` (https://nilearn.github.io/modules/generated/nilearn.connectome.ConnectivityMeasure.html).

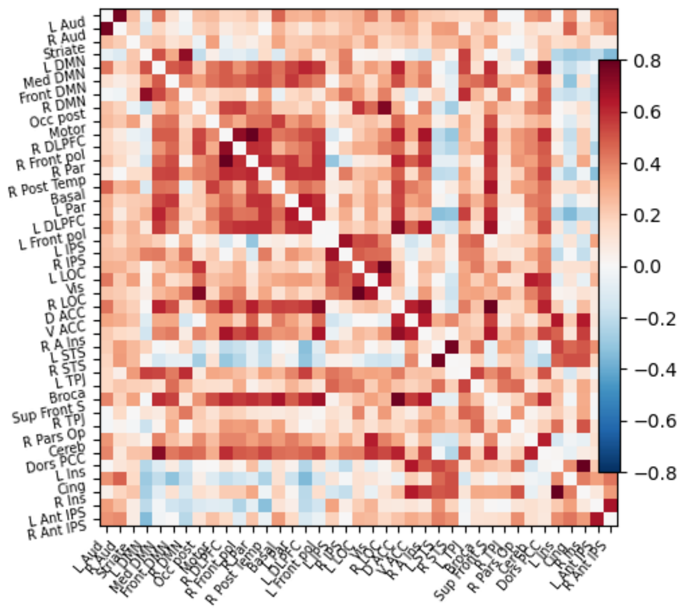
In [9]:
```
from nilearn.connectome import ConnectivityMeasure

correlation_measure = ConnectivityMeasure(kind='correlation')
correlation_matrix = correlation_measure.fit_transform([roi_time_series])[0]
```

We can then plot this functional connectivity matrix:

In [10]:
```
np.fill_diagonal(correlation_matrix, 0)
plotting.plot_matrix(correlation_matrix, labels=msdl_atlas.labels,
                     vmax=0.8, vmin=-0.8, colorbar=True)
```

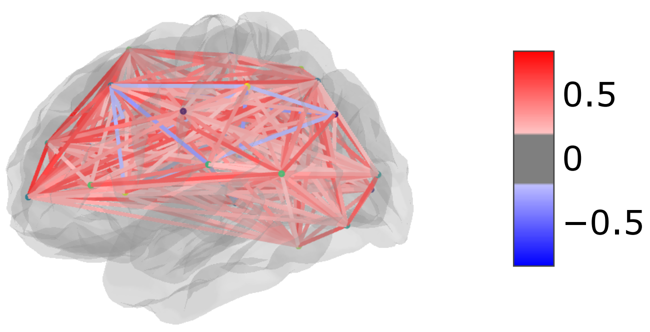
Out[10]:
```
<matplotlib.image.AxesImage at 0x7fc5fd425550>
```



Or view it as an embedded connectome:

In [11]:
```
plotting.view_connectome(correlation_matrix, edge_threshold=0.2,
                         node_coords=msdl_atlas.region_coords)
```

Out[11]:



view: Left

Accounting for noise sources

As we've already seen, maskers also allow us to perform other useful operations beyond just masking our data. One important processing step is correcting for measured signals of no interest (e.g., head motion). Our `development_dataset` also includes several of these signals of no interest that were generated during fMRIPrep pre-processing. We can access these with the `confounds` attribute, using `development_dataset.confounds`.

Let's quickly check what these look like for our first participant:

In [12]:
```
import pandas as pd
```

In [13]:
```
pd.read_table(development_dataset.confounds[0]).head()
```

Out[13]:

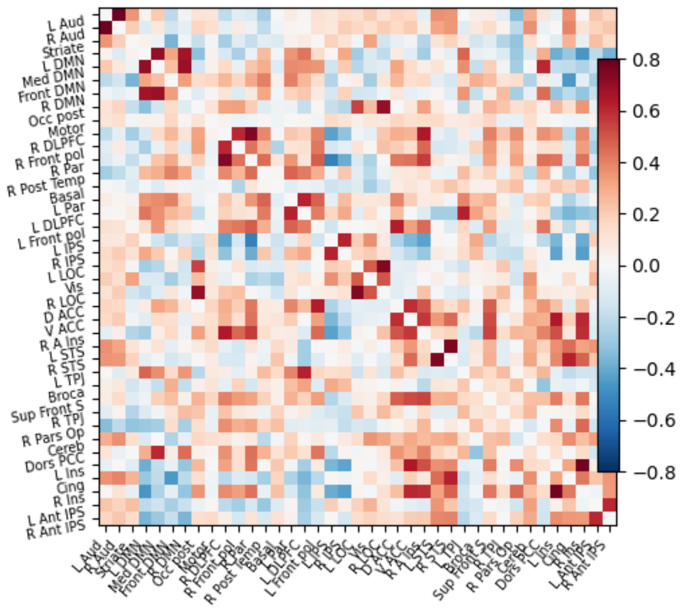
	trans_x	trans_y	trans_z	rot_x	rot_y	rot_z	framewise_displacement	a_comp_cor_00	a_comp_cor_01	a_comp_cor_02	a_comp_cor_03	a_c
0	-0.000233	-0.076885	0.062321	0.000732	0.000352	0.000841	0.000000	-0.099871	-0.007286	0.001780	-0.008073	
1	-0.006187	-0.078395	0.056773	0.000112	0.000187	0.000775	0.055543	-0.019437	-0.042308	0.016735	-0.012099	
2	-0.000227	-0.069893	0.083102	0.000143	0.000364	0.000716	0.054112	0.009096	-0.053206	-0.030388	-0.052925	
3	0.002492	-0.074707	0.060337	0.000202	0.000818	0.000681	0.057667	0.060195	-0.083195	0.003578	-0.037011	
4	-0.000226	-0.084204	0.085079	0.000183	0.000548	0.000682	0.051438	0.049833	-0.089819	-0.020825	-0.079329	

We can see that there are several different kinds of noise sources included! This is actually a subset of all possible fMRIPrep generated confounds that the Nilearn developers have pre-selected. We could access the full list by passing the argument `reduce_confounds=False` to our original call downloading the `development_dataset`. For most analyses, this list of confounds is reasonable, so we'll use these Nilearn provided defaults. For your own analyses, make sure to check which confounds you're using!

Importantly, we can pass these confounds directly to our masker object:

In [14]:
```
corrected_roi_time_series = masker.transform(
    development_dataset.func[0], confounds=development_dataset.confounds[0])
corrected_correlation_matrix = correlation_measure.fit_transform(
    [corrected_roi_time_series])[0]
np.fill_diagonal(corrected_correlation_matrix, 0)
plotting.plot_matrix(corrected_correlation_matrix, labels=msdl_atlas.labels,
                     vmax=0.8, vmin=-0.8, colorbar=True)
```

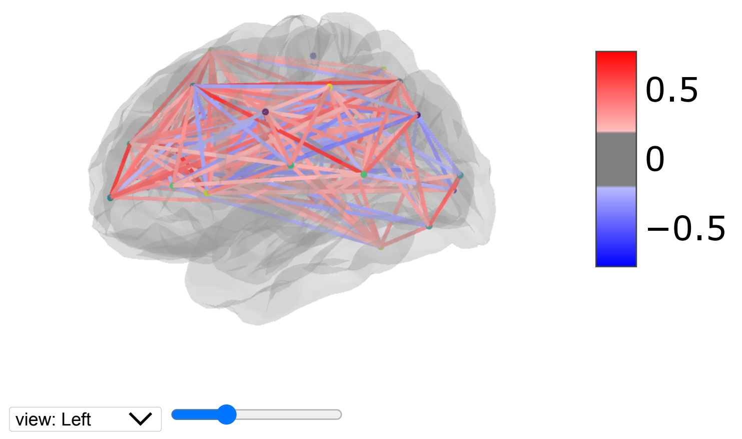
Out[14]:
```
<matplotlib.image.AxesImage at 0x7fc5d84428b0>
```



As before, we can also view this functional connectivity matrix as a connectome:

In [15]:
```
plotting.view_connectome(corrected_correlation_matrix, edge_threshold=0.2,
                         node_coords=msdl_atlas.region_coords)
```

Out[15]:



In both the matrix and connectome forms, we can see a big difference when including the confounds! This is an important reminder to make sure that your data are cleaned of any possible sources of noise *before* running a machine learning analysis. Otherwise, you might be classifying participants on e.g. amount of head motion rather than a feature of interest!

In [ ]:

In [ ]:

In [ ]:

An example classification problem

In [16]:
```
import warnings
warnings.filterwarnings("ignore")
```

Now that we've seen how to create a connectome for an individual subject, we're ready to think about how we can use this connectome in a machine learning analysis. We'll keep working with the same `development_dataset` , but now we'd like to see if we can predict age group (i.e. whether a participant is a child or adult) based on their connectome, as defined by the functional connectivity matrix.

We'll also explore whether we're more or less accurate in our predictions based on how we define functional connectivity. In this example, we'll consider three different different ways to define functional connectivity between our Multi-Subject Dictional Learning (MSDL) regions of interest (ROIs): correlation, partial correlation, and tangent space embedding.

To learn more about tangent space embedding and how it compares to standard correlations, we recommend {cite} `Dadi_2019` .

Load brain development fMRI dataset and MSDL atlas

First, we need to set up our minimal environment. This will include all the dependencies from the last notebook, loading the relevant data using our `nilearn` data set fetchers, and instantiated our `NiftiMapsMasker` and `ConnectivityMeasure` objects.

In [17]:
```
import numpy as np
import matplotlib.pyplot as plt
from nilearn import (datasets, input_data, plotting)
from nilearn.connectome import ConnectivityMeasure

development_dataset = datasets.fetch_development_fmri(n_subjects=30)
msdl_atlas = datasets.fetch_atlas_msdl()

masker = input_data.NiftiMapsMasker(
    msdl_atlas.maps, resampling_target="data",
    t_r=2, detrend=True,
    low_pass=0.1, high_pass=0.01).fit()
correlation_measure = ConnectivityMeasure(kind='correlation')
```

Now we should have a much better idea what each line above is doing! Let's see how we can use these objects across many subjects, not just the first one.

Region signals extraction

First, we can loop through the 30 participants and extract a few relevant pieces of information, including their functional scan, their confounds file, and whether they were a child or adult at the time of their scan.

Using this information, we can then transform their data using the `NiftiMapsMasker` we created above. As we learned last time, it's really important to correct for known sources of noise! So we'll also pass the relevant confounds file directly to the masker object to clean up each subject's data.

In [18]:
```
children = []
pooled_subjects = []
groups = []  # child or adult

for func_file, confound_file, phenotypic in zip(
        development_dataset.func,
        development_dataset.confounds,
        development_dataset.phenotypic):

    time_series = masker.transform(func_file, confounds=confound_file)
    pooled_subjects.append(time_series)

    if phenotypic['Child_Adult'] == 'child':
        children.append(time_series)

    groups.append(phenotypic['Child_Adult'])

print('Data has {0} children.'.format(len(children)))
```

```
Data has 24 children.
```

We can see that this data set has 24 children. This is roughly proportional to the original participant pool, which had 122 children and 33 adults.

We've also created a list in `pooled_subjects` containing all of the cleaned data. Remember that each entry of that list should have a shape of (168, 39). We can quickly confirm that this is true:

In [19]:
```
print(pooled_subjects[0].shape)
```

```
(168, 39)
```

ROI-to-ROI correlations of children

First, we'll use the most common kind of connectivity--and the one we used in the last section--correlation. It models the full (marginal) connectivity between pairwise ROIs.

`correlation_measure` expects a list of time series, so we can directly supply the list of ROI time series we just created. It will then compute individual correlation matrices for each subject. First, let's just look at the correlation matrices for our 24 children, since we expect these matrices to be similar:

In [20]:
```
correlation_matrices = correlation_measure.fit_transform(children)
```

Now, all individual coefficients are stacked in a unique 2D matrix.

In [21]:
```
print('Correlations of children are stacked in an array of shape {0}'
      .format(correlation_matrices.shape))
```

```
Correlations of children are stacked in an array of shape (24, 39, 39)
```

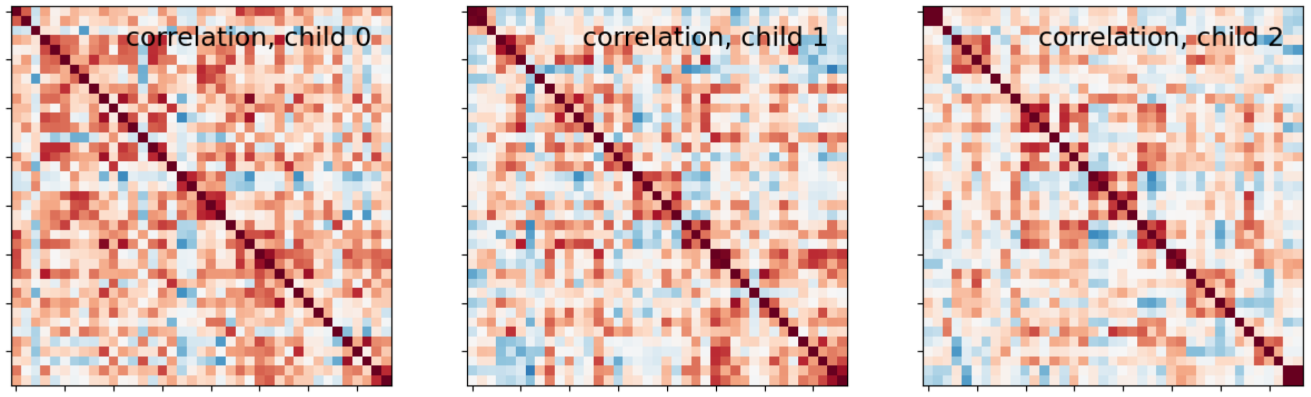
We can also directly access the average correlation across all fitted subjects using the `mean_` attribute.

In [22]:
```
mean_correlation_matrix = correlation_measure.mean_
print('Mean correlation has shape {0}.'.format(mean_correlation_matrix.shape))
```

```
Mean correlation has shape (39, 39).
```

Let's display the functional connectivity matrices of the first 3 children:

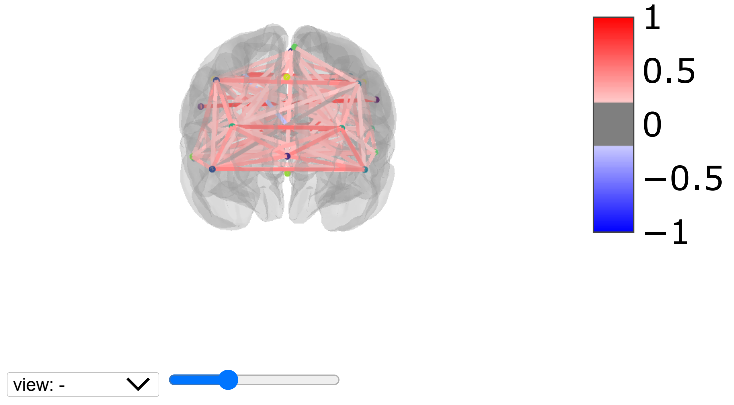
In [23]:
```
_, axes = plt.subplots(1, 3, figsize=(15, 5))
for i, (matrix, ax) in enumerate(zip(correlation_matrices, axes)):
    plotting.plot_matrix(matrix, colorbar=False, axes=ax,
                         vmin=-0.8, vmax=0.8,
                         title='correlation, child {}'.format(i))
```



Just as before, we can also display connectome on the brain. Here, let's show the mean connectome over all 24 children.

In [24]:
```
plotting.view_connectome(mean_correlation_matrix, msdl_atlas.region_coords,
                         edge_threshold=0.2,
                         title='mean connectome over all children')
```

Out[24]:



Studying partial correlations

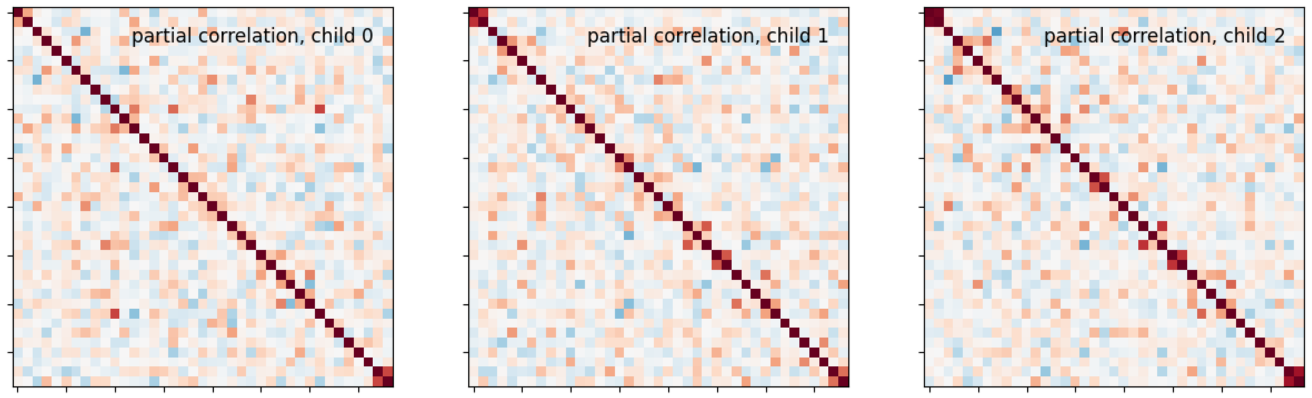
Rather than looking at the correlation-defined functional connectivity matrix, we can also study **direct connections** as revealed by partial correlation coefficients.

To do this, we can use exactly the same procedure as above, just changing the `ConnectivityMeasure` kind:

In [25]:
```
partial_correlation_measure = ConnectivityMeasure(kind='partial correlation')
partial_correlation_matrices = partial_correlation_measure.fit_transform(
    children)
```

Right away, we can see that most of direct connections are weaker than full connections for the first three children:

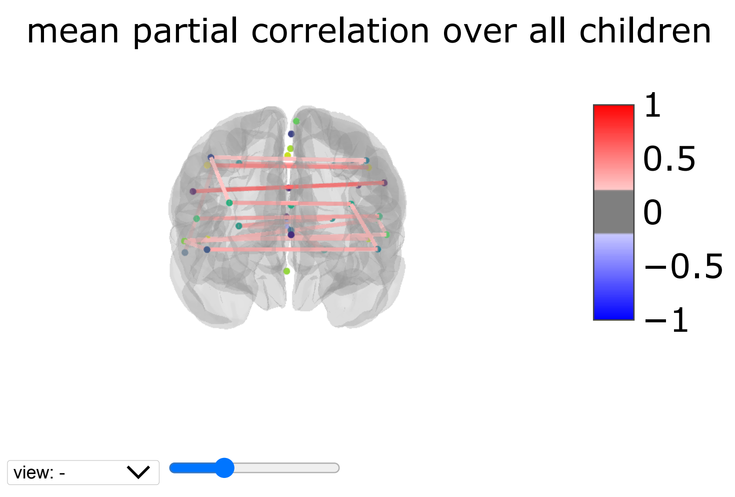
In [26]:
```
_, axes = plt.subplots(1, 3, figsize=(15, 5))
for i, (matrix, ax) in enumerate(zip(partial_correlation_matrices, axes)):
    plotting.plot_matrix(matrix, colorbar=False, axes=ax,
                         vmin=-0.8, vmax=0.8,
                         title='partial correlation, child {}'.format(i))
```



This is also visible when we display the mean partial correlation connectome:

In [27]:
```
plotting.view_connectome(
    partial_correlation_measure.mean_, msdl_atlas.region_coords,
    edge_threshold=0.2,
    title='mean partial correlation over all children')
```

Out[27]:



Using tangent space embedding

An alternative method to both correlations and partial correlation is tangent space embedding. Tangent space embedding uses **both** correlations and partial correlations to capture reproducible connectivity patterns at the group-level.

Using this method is as easy as changing the kind of `ConnectivityMeasure`

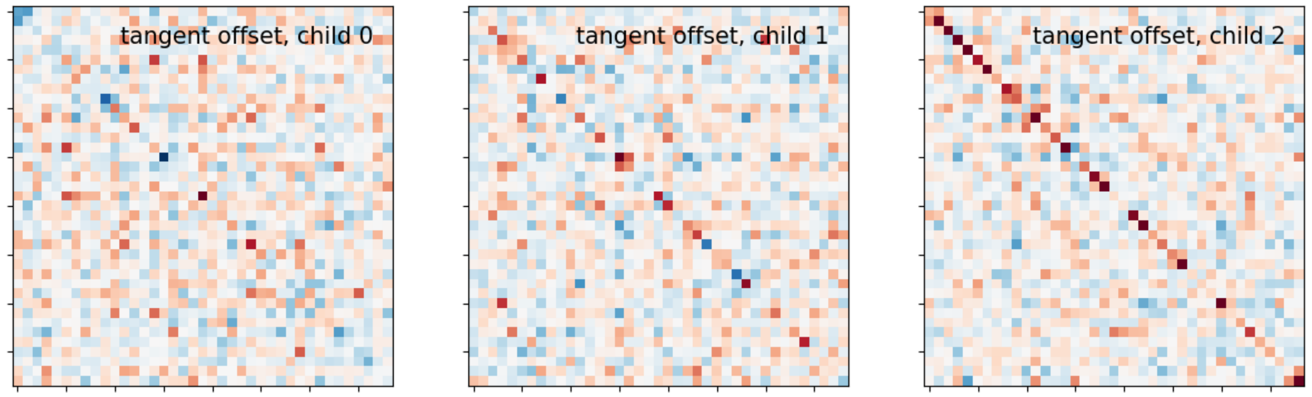
In [28]:
```
tangent_measure = ConnectivityMeasure(kind='tangent')
```

We fit our children group and get the group connectivity matrix stored as in `tangent_measure.mean_`, and individual deviation matrices of each subject from it.

In [29]:
```
tangent_matrices = tangent_measure.fit_transform(children)
```

`tangent_matrices` model individual connectivities as **perturbations** of the group connectivity matrix `tangent_measure.mean_`. Keep in mind that these subjects-to-group variability matrices do not directly reflect individual brain connections. For instance negative coefficients can not be interpreted as anticorrelated regions.

In [30]:
```
_, axes = plt.subplots(1, 3, figsize=(15, 5))
for i, (matrix, ax) in enumerate(zip(tangent_matrices, axes)):
    plotting.plot_matrix(matrix, colorbar=False, axes=ax,
                         vmin=-0.8, vmax=0.8,
                         title='tangent offset, child {}'.format(i))
```



We don't show the mean connectome here as average tangent matrix cannot be interpreted, since individual matrices represent deviations from the mean, which is set to 0.

Using connectivity in a classification analysis

We can use these connectivity matrices as features in a classification analysis to distinguish children from adults. This classification analysis can be implmented directly in scikit-learn, including all of the important considerations like cross-validation and measuring classification accuracy.

First, we'll randomly split participants into training and testing sets 15 times. `StratifiedShuffleSplit` allows us to preserve the proportion of children-to-adults in the test set. We'll also compute classification accuracies for each of the kinds of functional connectivity we've identified: correlation, partial correlation, and tangent space embedding.

In [31]:
```
from sklearn.metrics import accuracy_score
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.svm import LinearSVC

kinds = ['correlation', 'partial correlation', 'tangent']
_, classes = np.unique(groups, return_inverse=True)
cv = StratifiedShuffleSplit(n_splits=15, random_state=0, test_size=5)
pooled_subjects = np.asarray(pooled_subjects)
```

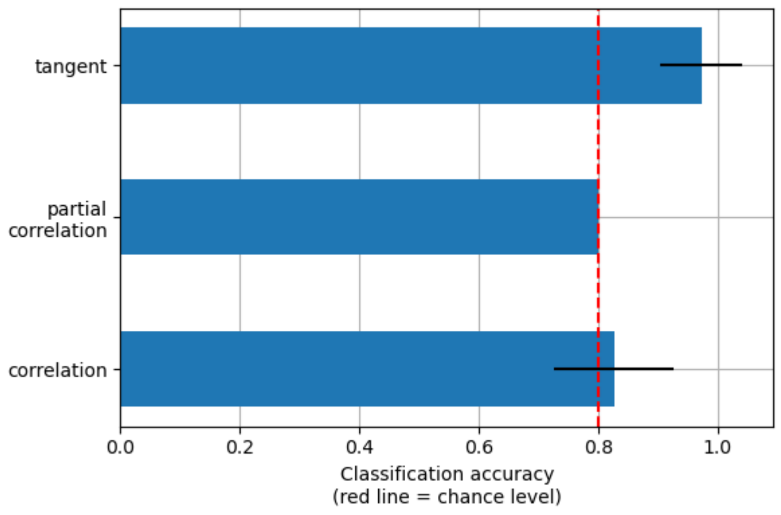
Now, we can train the scikit-learn `LinearSVC` estimator to on our training set of participants and apply the trained classifier on our testing set, storing accuracy scores after each cross-validation fold:

In [32]:
```
scores = {}
for kind in kinds:
    scores[kind] = []
    for train, test in cv.split(pooled_subjects, classes):
        # *ConnectivityMeasure* can output the estimated subjects coefficients
        # as a 1D arrays through the parameter *vectorize*.
        connectivity = ConnectivityMeasure(kind=kind, vectorize=True)
        # build vectorized connectomes for subjects in the train set
        connectomes = connectivity.fit_transform(pooled_subjects[train])
        # fit the classifier
        classifier = LinearSVC().fit(connectomes, classes[train])
        # make predictions for the left-out test subjects
        predictions = classifier.predict(
            connectivity.transform(pooled_subjects[test]))
        # store the accuracy for this cross-validation fold
        scores[kind].append(accuracy_score(classes[test], predictions))
```

After we've done this for all of the folds, we can display the results!

In [33]:
```
mean_scores = [np.mean(scores[kind]) for kind in kinds]
scores_std = [np.std(scores[kind]) for kind in kinds]

plt.figure(figsize=(6, 4))
positions = np.arange(len(kinds)) * .1 + .1
plt.barh(positions, mean_scores, align='center', height=.05, xerr=scores_std)
yticks = [k.replace(' ', '\n') for k in kinds]
plt.yticks(positions, yticks)
plt.gca().grid(True)
plt.gca().set_axisbelow(True)
plt.gca().axvline(.8, color='red', linestyle='--')
plt.xlabel('Classification accuracy\n(red line = chance level)')
plt.tight_layout()
```


tangent
partial
correlation
correlation
Classification accuracy
(red line = chance level)