Advances in imaging equipment and automation have led to an overabundance of data on the functioning of the brain. Technologies today can sample brain activity from a large number of neurons in a large region while organisms are actively behaving. For example, by simultaneously recording the electrical activity of every neuron of the mouse brain over an extended period of time, the amount of data generated will create completely new paradigms for biology, that will require the development of tools to extract value from such unprecedented amount of information.

In this Notebook, we use PySpark and the Thunder project (https://github.com/thunder-project/thunder), which is developed on top of PySpark, for processing large amounts of time series data in general, and neuroimaging data in particular. We will use these tools for the task of understanding some of the structure of Zebrafish brains, which is a typical (and simple) example used in Neuroimaging. Using Thunder, we will cluster different regions of the brain (representing groups of neurons) to discover patterns of activity as the zebrafish behaves over time.

**Note**: Please, use the documentation for the Thunder API (http://docs.thunder-project.org/) to learn the details of function calls!

## Goals

The main goals of this notebook are:

1. Learn about Thunder and how to use it
2. Revisit the K-Means algorithm and the method for choosing K
3. Learn alternative approaches to improve the results

## Steps

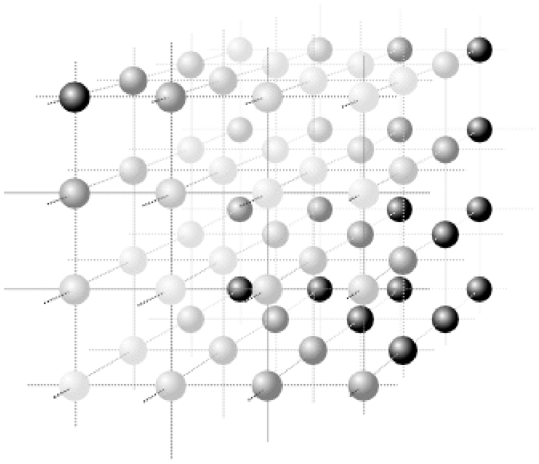
1. In section 1, we go though some background concepts that are used in this notebook.
2. Next, in section 2, we will get familiar with Thunder, its methods and its data types, by working on some simple tasks.
3. Finally, in section 3, we will build a model to cluster the neurons of a zebrafish based on their behavior. In this step, we will learn about how to use K-Means when the value of K is unknown. Finally, some tricks to improve the results are introduced.

## 1. Background concepts

In this section, we cover the terminology and the concepts that constitute the domain knowledge for this notebook.

As it should be well-known, a `pixel` is a combination of "**pic**ture **el**ement": digital images can be modeled as simple 2-dimensional (2D) matrices of intensity values, and each element in the matrix is a pixel. In color images, a pixel contains values of red, green, and blue channels. In a grayscale image, the three channels have the same value, such that each pixel is reduced to be a single value.

A single 2D image is not nearly enough to express 3D objects, which use a **voxel**, representing a value of the 3D image on a regular grid in a three-dimensional space. A possible technique to work on 3D images is to acquire multiple 2D images of different slices (or `planes`, or `layers`) of a 3D object, and stack them one on top of each other (a z-stack). This ultimately produces a 3D matrix of intensity values, where each value represents a `volume element` or `voxel`.



This z-stack image has 4 layers. A point is a voxel. It can be determined by the layer's index and the position in that layer.

In the context of the Thunder package, we use term `image` for `3D-image` or `stack image`. Thunder uses `Image` type to represent 3D-image. Each `Image` object is a collection of either 2D images or 3D volumes. In practice, it wraps an n-dimensional array, and supports either distributed operations via Spark or local operations via numpy , with an identical API.

Stack-images can represent 3D objects, but it can be difficult to take the temporal relationship of the images into account. To do that, we need another data structure that shows the changes of voxels over time. In the Thunder package, the internal `Series` type can be used exactly for this purpose. Each `Series` is a 1D array such that each element is a value of the voxel at a timestamp.

The most common series data is time series data, in which case the index is time and each record is a different signal, like a channel or pixel.

We now have sufficient material to start playing with Thunder !!!

## 2. Let's play

Well, wait a second before we play... Remember, we're going to use Spark to perform some of the computations related to this Notebook. Now, when you spin a Zoe Notebook application (this comment is valid for students at Eurecom), you'll gain access to an individual, small Spark cluster that is dedicated to your Notebook. This cluster has two worker machines, each with 6 cores. As such, a good idea to obtain smooth performance and a balanced load on the workers, is to `repartition` your data (i.e., the RDDs you use to represent images or time series).

In this Notebook we **expect** students to take care of repartitioning, and such care will be compensated by bonus points.

## 2.1. Play with Image objects

### a. Loading image data

Both `images` and `series` can be loaded from a variety of data types and locations. You need to specify whether data should be loaded in 'local' mode, which is backed by a numpy array, or in 'spark' mode, which is backed by an RDD by using the optional argument `engine`. The argument `engine` can be either None for local use or a SparkContext for` distributed use with Spark.

```
import thunder as td

# load data from tif images
data = td.images.fromtif('/path/to/tifs')

# load data from numpy-arrays
data = td.series.fromarray(somearray)
data_distributed = ts.series.fromarray(somearray, engine=sc)
```

We can load some example image data by:

In [1]:

```
import thunder as td
import numpy as np

# load some example image data
image_data = td.images.fromexample('fish', engine=sc)

# print the number of images
print(image_data.count())
```

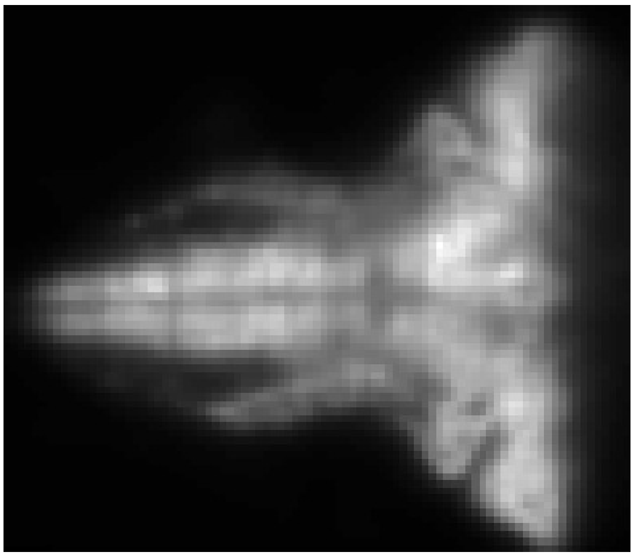
```
20
```

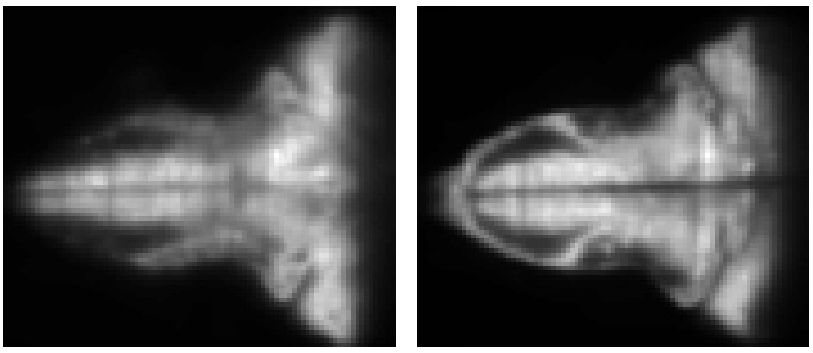
### b. Inspecting image data

In [3]:

```
%matplotlib inline
import matplotlib.pyplot as plt

# import two function to draw images easier
from showit import image as draw_image
from showit import tile as draw_tile

print("Shape of the data:", image_data.shape)

first_image = image_data.first() # get the values of Image object
# or first_image = image_data[0] # get the Image object

print("Shape of the data of the first image:", first_image.shape)

print("Data of the first image:", first_image)


# draw the first layer of the first image
draw_image(first_image[0])

# draw all layers of the first image
draw_tile(first_image)

# we can use index slices to take images
samples = image_data[0:6]
```

```
Shape of the data: (20, 2, 76, 87)
Shape of the data of the first image: (2, 76, 87)
Data of the first image: [[[26 26 26 ..., 26 26 26]
  [26 26 26 ..., 26 26 26]
  [26 26 26 ..., 27 27 26]
  ...,
  [26 26 26 ..., 27 27 26]
  [26 26 26 ..., 27 26 26]
  [25 25 25 ..., 26 26 26]]

 [[25 25 25 ..., 26 26 26]
  [25 25 25 ..., 26 26 26]
  [26 26 26 ..., 26 26 26]
  ...,
  [26 26 26 ..., 26 26 26]
  [26 26 26 ..., 26 26 26]
  [25 25 25 ..., 26 26 26]]]
```





From the result above, the shape of the loaded data is (20, 2, 76, 87). It means we have total 20 3D images objects. Each image has 2 layers, each layer has size 76x87.

Note that, although data is not itself an array (it can be a kind of RDD), we can index into it using bracket notation, and pass it as input to plotting methods that expect arrays. In these cases, the data will be automatically converted.

One of the advantages of working in Python is that we can easily visualize our data stored into Spark RDDs using the Matplotlib library. Function `draw_image` and `draw_tile` that take advantages of Matplotlib are examples.

**Question 1**

a) Use the function `imgshow` from matplotlib to plot each layer of the first image in `image_data`. b) Discuss the choice of parameters you use for the method `imgshow`

let's as requested partion the data to the 12 cores we have before any process :

In [4]:

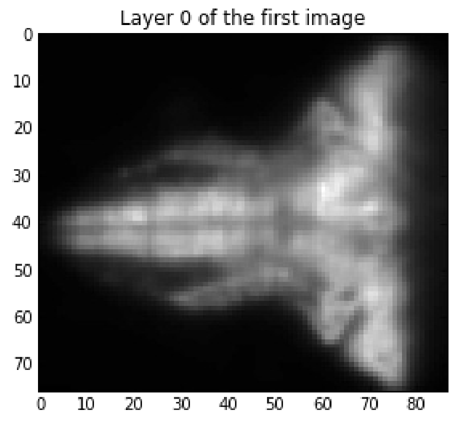
```
image_data.repartition(12)
```

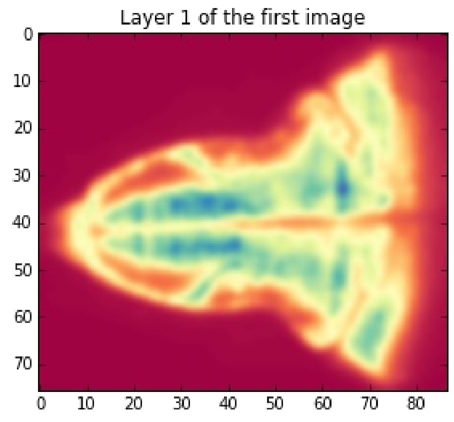
Out[4]:

```
Images
mode: spark
dtype: uint8
shape: (20, 2, 76, 87)
```

In [5]:

```
img = image_data.first()
# or:
# img = image_data[1]

# show the first layer
plt.imshow(img[0], interpolation='nearest', aspect='equal', cmap='gray')
plt.title("Layer 0 of the first image")
plt.show()

# show the second layer
plt.imshow(img[1], interpolation='bicubic', aspect='equal', cmap='Spectral')
plt.title("Layer 1 of the first image")
plt.show()
```





For the first layer we chose **'nearest'** for the interpolation. This simply display the image without interpolating between pixels if the display resolution is different from the image resolution. And since it's often the case, the result is a pixelated image. This is useful if we want to have a fine inspection of intensity variations.

For the second layer we chose **'bicubic'** which allows to have a smoother display, yet blurry.In fact, Bicubic interpolation is often used when blowing up photos because people tend to prefer blurry over pixelated pictures.

For the aspect we kept **'equal'** for both displays. It represents the Aspect ratio,i.e the ratio of the sizes of the two axes. When it's set to equal, the axes have equal unit length.

Regarding the Color of displaying, we chose **'gray'** for the first layer. This colormap is approximately monochromatic varying smoothly between two color tones. It's one of the Sequential colormaps and they are ideal for representing most scientific data since they show a clear progression from low-to-high values.

And for the second layer we chose **'Spectral'** .It's one of the Diverging colormaps that have a median value. They are Ideal when the data has a significant median value. By using an RGB scale rather than a grayscale we can better see the variations of the level of activity.

Then, we can perform operations that aggregate information across images.

**Question 2**

Calculate the standard deviation across all images you have in `image_data` (that is, our dataset). To clarify, let's focus on an individual layer (say the first layer of each image). For every `voxel`, compute the standard deviation of its values across different images for the same layer. Visualize the standard deviation you obtain, for example concerning a single layer (as before, say the first layer).

**HINT 1** to avoid wasting time and energy, make sure you lookup for methods that could help answer the question from the Thunder documentation.

**HINT 2** We can also use function `draw_image(<data>)` to plot an image in a simple way instead of using many statements with matplotlib as before.

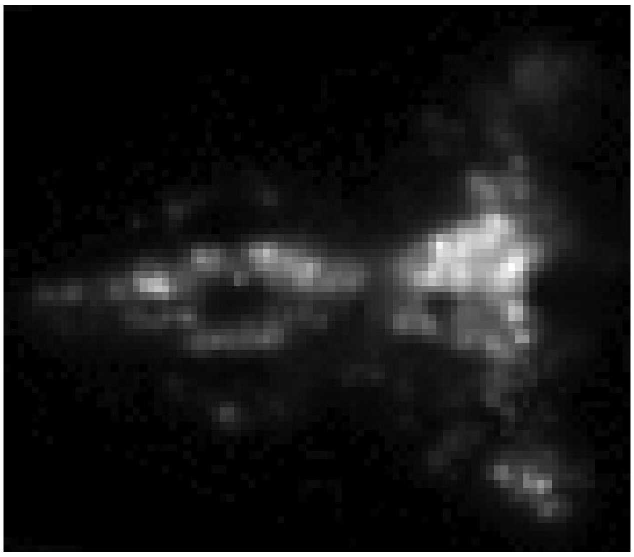
**NOTE** Comment the image you obtain. What does it mean to display the standard deviation across all images in a single layer?

In [6]:

```
####!@SOLUTION@!####
# calculate standard deviation of images
std_imgs = image_data.map(lambda x : x[0]).std()
draw_image(std_imgs)
```

Out[6]:

```
<matplotlib.image.AxesImage at 0x7f4da0ebc4a8>
```



Displaying the standard deviation across all images in a single layer allows us to see the hot areas, i.e the areas where the activity changes the most.

## c. Selecting samples of image data

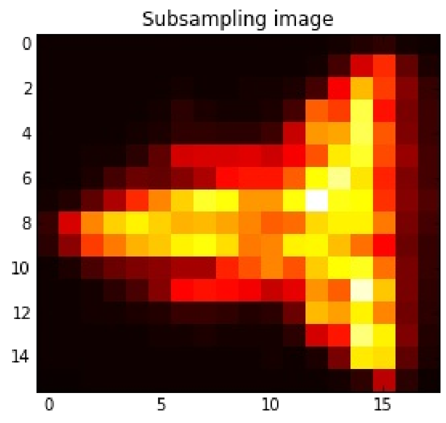
The Images API offers useful methods for working with large image data. For example, in some cases it is necessary to subsample each image, to make sure we can analyze it efficiently.

**Question 3**

The source code below subsamples image data with different ratios on different dimensions. a) Complete the source code to plot the first layer of the first image. b) What is the shape of `image_data` before and after subsampling?

In [7]:

```
subsampled = image_data.subsample((1, 5, 5))
# Stride to use in subsampling. If a single int is passed, each dimension of the image
# will be downsampled by this same factor. If a tuple is passed, it must have the same
# dimensionality of the image. The strides given in a passed tuple will be applied to
# each image dimension
plt.imshow(subsampled.first()[0], interpolation='nearest', aspect='equal', cmap='hot')
plt.title("Subsampling image")
plt.show()
print("Before subsampling:",image_data.shape)
print("After subsampling:", subsampled.shape)
```



```
Before subsampling: (20, 2, 76, 87)
After subsampling: (20, 2, 16, 18)
```

Note that `subsample` is an RDD operation, so it returns immediately. Indeed, we know that in Spark you must apply a RDD action to trigger the actual computation.

## d. Converting image data

We can also convert an RDD of images to a RDD of series by:

In [8]:

```
seriesRDD = image_data.toseries()
seriesRDD.cache()
```

Out[8]:

```
Series
mode: spark
dtype: uint8
shape: (2, 76, 87, 20)
```

**Question 4**

According to your understanding about `Series` objects which was introduced in section 1, what is the shape of `seriesRDD` and its elments ? Comment your results, don't just display numbers.

In [9]:

```
print('the shape of seriesRDD and its elements :',seriesRDD.shape)
```

```
the shape of seriesRDD and its elements : (2, 76, 87, 20)
```

the shape of seriesRDD variable contains 4 numbers that can be explained as follows : 2 layers of 76*87 voxels each. And there are 20 samples of those voxels as they change over the time.

For a large data set that will be analyzed repeatedly as a `Series`, it will ultimately be faster and more convienient to save `Images` data to a collection of flat binary files on a distributed file system, which can in turn be read back in directly as a `Series`, rather than repeatedly converting the images to a `Series` object. This can be performed either through a ThunderContext method, `convertImagesToSeries`, or directly on an Images object, as done below:

In [10]:

```
# image_data.toseries().tobinary('directory', overwrite=True)
#ts = td.series.frombinary('directory', engine=sc)
```

We will study about `Series` object in the next section.

## 2.2. Play with Serises objects

### a. Loading Series data

In this section, we use a sample data to explore `Series` objects.

In [11]:

```
# series_data = td.series.fromexample('fish', engine=sc)
# series_data = td.series.frombinary(path='s3n://thunder-sample-data/series/fish', engine=sc)
series_data = image_data.toseries()
series_data.cache()
```

Out[11]:

```
Series
mode: spark
dtype: uint8
shape: (2, 76, 87, 20)
```

### b. Inspecting Series data

`Series_data` is a distributed collection of key-value records, each containing a coordinate identifier and the time series of a single `voxel`. We can look at the first record by using `first()`. It's a key-value pair, where the key is a tuple of `int` (representing a spatial coordinate within the imaging volume) and the value is an one-dimensional array.

In [12]:

```
first_series = series_data.first() # get the values of Series object
#first_series = series_data[0] # get a Series object

print("Shape of series:", series_data.shape)
print("The first series:", first_series)
print("Each element in series has", len(first_series), "values")

# print the 10th value of voxel (0,0,0)
# layer = 0
# coordinator = (0,0) in that layer
print("value 10th of voxel (0,0,0):", np.array(series_data[0,0,0,10]))
```

```
Shape of series: (2, 76, 87, 20)
The first series: [26 26 26 26 26 26 26 25 26 25 25 25 26 26 26 26 26 26 26 26]
Each element in series has 20 values
value 10th of voxel (0,0,0): 25
```

The loaded series data is a multi-dimensional array. We can access the values of a voxel in time series by using a tuple as above. In our data, each voxel has 20 values corresponding to 20 states at 20 different times.

## c. Selecting Series data

Series objects have a 1D index, which can be used to subselect values.

In [13]:

```
print("shape of index:", series_data.index.shape)
print("the first element of a subset", series_data.between(0,8).first())
```

```
shape of index: (20,)
the first element of a subset [26 26 26 26 26 26 26 25]
```

Values can be selected based on their index:

In [14]:

```
print(series_data.select(lambda x: x > 3 and x < 8).index)
print(series_data.select(lambda x: x > 3 and x < 8).first())
```

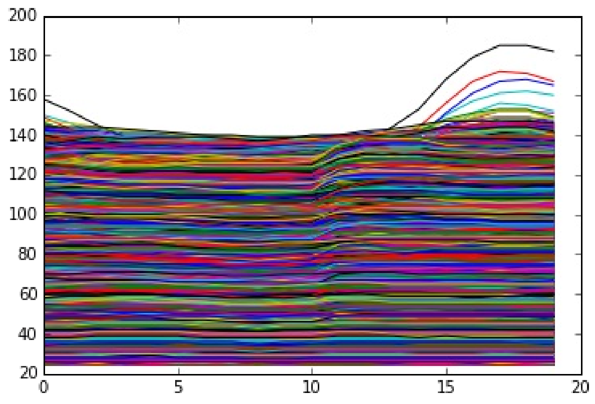
```
[4, 5, 6, 7]
[26 26 26 25]
```

**Question 5**

Plot the first 20 values of **all** series objects (that is the values of a voxel) in the series data. This means, on the same plot, you should visualize the values each voxel takes in the first 20 time intervals.

In [15]:

```
import numpy as np

# only select the first 20 states of each object
samples = series_data.between(0,20).tordd().values().collect()

plt.plot(np.array(samples).T)
plt.show()
```



Now, another objective we can have is to select specific series objects within the same series data. For example, we can select objects randomly by using function `sample`.

**Question 6**

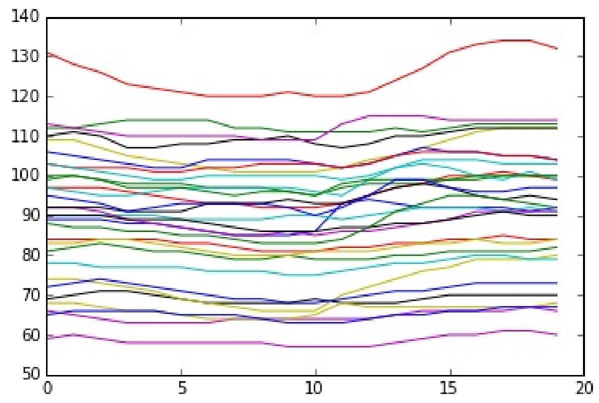
Let's plot a random subset of the data using the method `sample`. Complete the source code below to plot the first 20 values of 30 objects that are selected randomly among those that pass the condition on the standard deviation, using function `sample`.

In [16]:

```
#######################################################################################################

# select 30 objects randomly which have standard deviation > threshold
# Extract random subset of records, filtering on a summary statistic.
examples = series_data.filter(lambda x: x.std() > 1.0).sample(30)
# only plot first 20 states of each object
plt.plot(np.array(examples).T)
```

```
Out[16]:
```

```
[<matplotlib.lines.Line2D at 0x7f4d8882afd0>,
 <matplotlib.lines.Line2D at 0x7f4d88831400>,
 <matplotlib.lines.Line2D at 0x7f4d88831748>,
 <matplotlib.lines.Line2D at 0x7f4d88831a90>,
 <matplotlib.lines.Line2D at 0x7f4d88831dd8>,
 <matplotlib.lines.Line2D at 0x7f4d88837160>,
 <matplotlib.lines.Line2D at 0x7f4d888374a8>,
 <matplotlib.lines.Line2D at 0x7f4d88863b00>,
 <matplotlib.lines.Line2D at 0x7f4d88837b38>,
 <matplotlib.lines.Line2D at 0x7f4d88837e80>,
 <matplotlib.lines.Line2D at 0x7f4d8883f208>,
 <matplotlib.lines.Line2D at 0x7f4d8883f550>,
 <matplotlib.lines.Line2D at 0x7f4d8883f898>,
 <matplotlib.lines.Line2D at 0x7f4d8883fbe0>,
 <matplotlib.lines.Line2D at 0x7f4d888377f0>,
 <matplotlib.lines.Line2D at 0x7f4d887c62b0>,
 <matplotlib.lines.Line2D at 0x7f4d887c65f8>,
 <matplotlib.lines.Line2D at 0x7f4d887c6940>,
 <matplotlib.lines.Line2D at 0x7f4d887c6c88>,
 <matplotlib.lines.Line2D at 0x7f4d887c6fd0>,
 <matplotlib.lines.Line2D at 0x7f4d887cd358>,
 <matplotlib.lines.Line2D at 0x7f4d8883ff28>,
 <matplotlib.lines.Line2D at 0x7f4d887cd9e8>,
 <matplotlib.lines.Line2D at 0x7f4d887cdd30>,
 <matplotlib.lines.Line2D at 0x7f4d887d50b8>,
 <matplotlib.lines.Line2D at 0x7f4d887d5400>,
 <matplotlib.lines.Line2D at 0x7f4d887d5748>,
 <matplotlib.lines.Line2D at 0x7f4d887d5a90>,
 <matplotlib.lines.Line2D at 0x7f4d887cd6a0>,
 <matplotlib.lines.Line2D at 0x7f4d887dc160>]
```



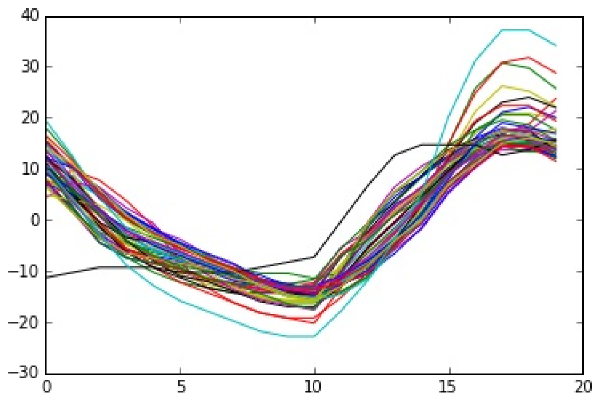
## d. Preprocessing Series data

A `Series` objects has some methods which can be useful in an eventual preprocessing phase.

For example,`center` subtracts the mean, `normalize` subtracts and divides by a baseline (either the mean, or a percentile).

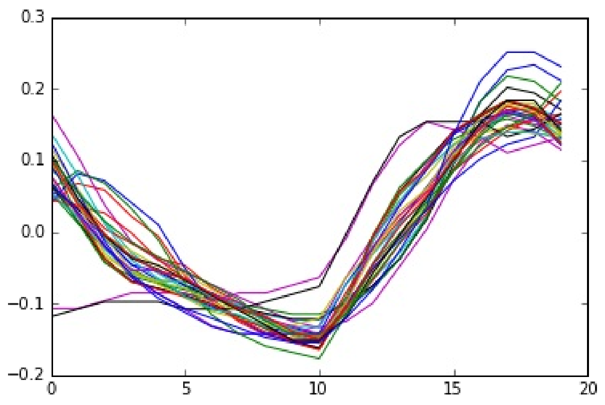
```
In [17]:
```

```
examples = series_data.center().filter(lambda x: x.std() >= 10).sample(50)
plt.plot(np.array(examples).T)
plt.show()
```



In [18]:

```
normalizedRDD = series_data.normalize(method='mean').filter(lambda x: x.std() >= 0.1).sample(50)
plt.plot(np.array(normalizedRDD).T)
plt.show()
```

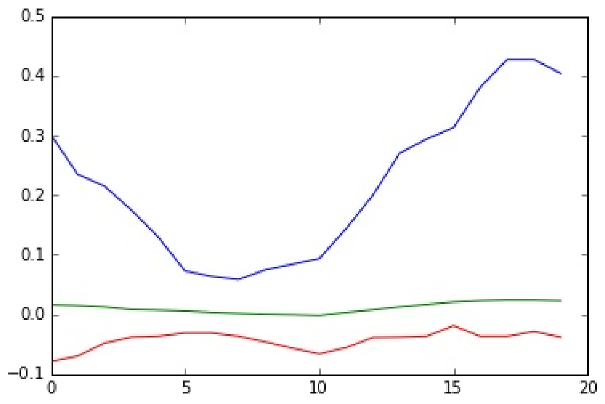


## e. Computing statistics about Series data

A `Series` can be summarized with statistics both within and across images. To summarize **across records** (the statistic of all voxels at each timestamp), we can do the following:

In [19]:

```
plt.plot(series_data.normalize().max());
plt.plot(series_data.normalize().mean());
plt.plot(series_data.normalize().min());
```



To summarize **within records**, we can use the `map` method:

In [20]:

```
means = series_data.map(lambda x: x.mean())
flat_means = means.flatten().toarray()
flat_stdevs = stdevs = series_data.map(lambda x: x.std()).flatten().toarray()
print("means:", flat_means)
print("length of means:", len(flat_means))
print("mean of the first series:", flat_means[0])
print("standard deviation of the first series:", flat_stdevs[0])
```

```
means: [ 25.8   25.85  25.7  ...,  26.    26.    26.  ]
length of means: 13224
mean of the first series: 25.8
standard deviation of the first series: 0.4
```

`means` is now a `Series` object, where the value of each record is the mean across the time series for that voxel.

Note that in the source code above, we use function `toarray` to return all records to the driver as a numpy array.

For this `Series`, since the keys correspond to spatial coordinates, we can `pack` the results back into a local array in **driver node**.

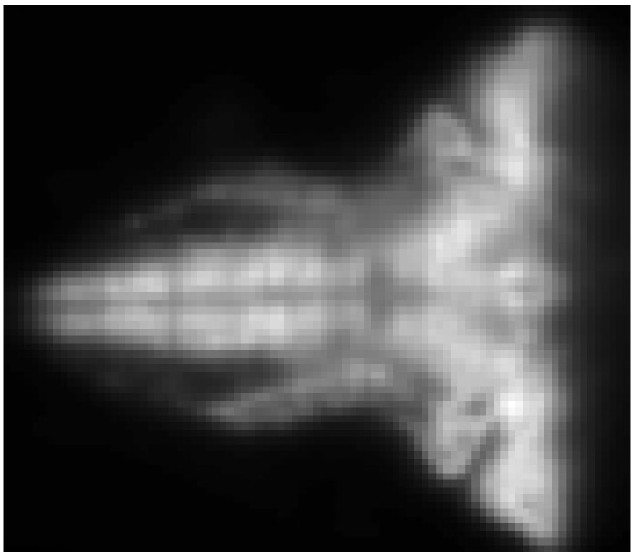
To look at this array as an image, we can use function `draw_image` as before.

In [21]:

```
# we should recover the shape of means before plotting
# draw the stdandard deviations of series that belong to the first layer
draw_image(flat_means.reshape((2, 76, 87)) [0,:,:])
```

Out[21]:

```
<matplotlib.image.AxesImage at 0x7f4d89497978>
```



Note that `toarray` is an example of a local operation, meaning that all the data involved will be sent to the Spark driver node. In this case, packing the mean is no problem because its size is quite small. But for larger data sets, this can be **very problematic**. So, it's a good idea to downsample, subselect, or otherwise reduce the size of your data before attempting to pack large image data sets!

### f. Identifying correlations

In several problem domains, it may also be beneficial to assess the similarity between a designated signal (time series) and another signal of interest by measuring their correlation. For example, say we have two time series corresponding to the consumption of Coca Cola and Pepsi, it would perhaps be interesting to verify whether behavioural patterns are similar for both brands over time.

Simply as a proof of concept, we shall compare our data to a random signal and we expect that, for a random signal, the correlation should be low. The signal can be stored as a numpy array or a MAT file containing the signal as a variable. Note that the size of the signal must be equal to the size of each `Series` element.

In [22]:

```
from numpy import random
signal = random.randn(len(first_series))
print("The correlation of the first element with random signal:" , series_data.correlate(signal).first())

first_element = series_data.first()
corr = series_data.correlate(np.array(first_element)).first()
print("The correlation of the first element with itselft:", corr)
```

```
The correlation of the first element with random signal: [ 0.18487991]
The correlation of the first element with itselft: [ 1.]
```

## 3. Usecase

## 3.1. Context

Neurons have a variety of biochemical and anatomical properties. Classification methods are thus needed for clarification of the functions of neural circuits as well as for regenerative medicine. In this usecase, we want to categorize the neurons in a fish brain, based on their behavior. The behavior of a neuron can be expressed by the change of its states. The activies of the brains are captured over time into images.

Neurons have a variety of biochemical and anatomical properties. Classification methods are thus needed for clarification of the functions of neural circuits as well as for regenerative medicine. In this usecase, we want to categorize the neurons in a fish brain, based on their behavior. The behavior of a neuron can be expressed by the change of its states. The activies of the brains are captured over time into images.

In this notebook, we use K-Means, a well known clustering algorithm which is also familiar to you, as it was introduced during the last lecture on anomaly detection.

## 3.2 Data

The dataset we will use is the time series data which we played with in the previous section. Refer to section 2 if you want to duplicate the code to load such data.

## 3.3. Building model

### a. Importing required modules

In [23]:

```
%matplotlib inline
import matplotlib.pyplot as plt
from pyspark.mllib.clustering import KMeans, KMeansModel
from matplotlib.colors import ListedColormap
```

### b. Loading & inspecting the data

**Question 7**

Load example series data from `fish`, then normalize and cache it to speed up repeated queries. Print the dimensional information of the loaded data.

In [24]:

```
####!@SOLUTION@!####
# we must normalize it to get best clustering
data = td.images.fromexample('fish', engine=sc).toseries().normalize()
data.repartition(12)

# cache it to speed up related queries
data.cache()

# check the dimensions of data
print (data.shape)
```

```
(2, 76, 87, 20)
```

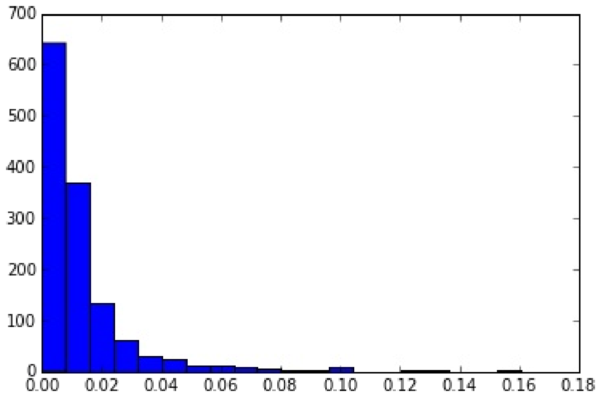
**Question 8**

When studying the properties of large data set, we often take a small fraction of it. We have many strategies to select this subset, such as selecting randomly, selecting elements that has the standard deviation bigger than a threshold, or mixing the conditions. In this notebook, we will use the second method as a small demonstration.
In order to choose a good value for the threshold of standard deviation, we should compute the stddev of each series and plot a histogram of a 10% sample of the values.

Complete the source code below to compute the standard deviation of series in data. Plot the histogram of it and discuss it in details. In your opinion, what should be the best value for the threshold ?

In [25]:

```
# calculate the standard deviation of each series
# then select randomly 10% of value to plot the histogram
stddevs = (data.map(lambda x: x.std()).sample(int(data.count()/10)).flatten().toarray())



# plot the histogram of 20 bins
plt.hist(stddevs, bins=20)
plt.show()
```



We can notice that a lot of series standard deviation values are included in the subset [0.00,0.005], which is logical because a lot of points belong to the background of the image. The rest, i.e the points having higher standard deviation, we assume that there are some of them that although they belong to the brain, they are less relevant for the study as their variation of activity is very low. Instead, there are some with high standard deviation and where the activity changes the most, and we assume that those points are the interesting ones. We assumed that a good threshold for our case can be 0.01

In [26]:

```
data_count = data.count()
print(data_count)
print("data over the threshold:",data.filter(lambda x: x.std() >= 0.01).count()/ data_count *100,'%')
```

```
13224
data over the threshold: 44.60072595281307 %
```

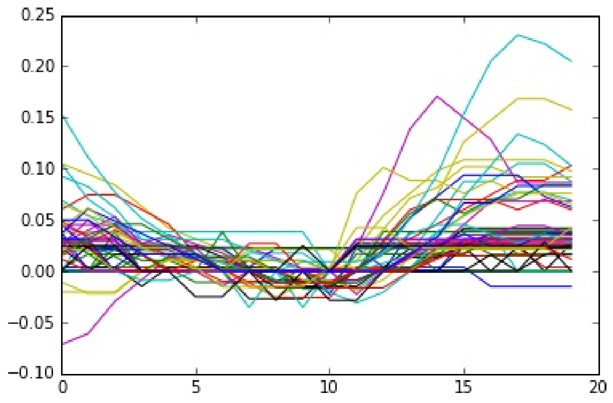
Note that we can notice that this choice is somehow accurate as we saw in the visualization of the first layer of the first image and the standard deviations of this layer across all images in questions 1 and 2. In fact, we can see that around ~ 45% of voxels belong to the brain and thus have significant variations, while no variation affects the remaining ~ 55% (belonging to the background).

**Question 9**

Extract some samples just to look at the typical structure of the time series data. The objects are selected randomly, and has the standard deviation bigger than the threshold which you picked in question 8. Plot the samples and discuss your obtained figure.

In [27]:

```
# sample 50 objects of the data randomly base on the standard deviation
threshold=0.01
examples = data.filter(lambda x: x.std() >= threshold).sample(50)


# plot the sample data
plt.plot(np.array(examples).T)
plt.show()
```



The obtained figure shows that some of the most active neurons have the same allure of their activity over the time. Particularly, some curves (for instance the red one) have the allure of a sinusoid.

## c. Clustering series

In this section, we will use K-means to cluster the series. In other words, we cluster the voxels based on the their behavior. Currently, we have no clue about how many groups K of neural behavior. To this end, instead of choosing a single value K, we use multiple values, build model with each K and compare the resulting error values. After that, we can choose the best value of K.

**Question 10**

Complete the source below to build multiple models coresponding to multiple values of `K` using algorithm KMeans of Thunder. a) Comment the structure of the code. Precisely, focus on the `for` loop, and state what is parallel and what is not. b) Can you modify the structure of the code such that you use the most of the parallelization capabilities of Spark?

In [35]:

```
from pyspark.mllib.clustering import KMeans #######################################
# declare the possible values of K
ks = [5, 10, 15, 20, 30, 50, 100, 200]


# convert series data to rdd of values
training_data = data.tordd().map(lambda x: np.array(x[1]) ).cache()

def buildModels(data):
    # declare the collection of models
    models = []

    # build model for each K and append to models
    for k in ks:
        models.append(KMeans.train(data, k))
    return models
ddata = data.tordd().map(lambda x: np.array(x[1]) ).cache() ################################
models = buildModels(ddata)
```

**a)** We think that in the for loop the parallelisation takes place only within the Kmeans algorithm already programmed in the MLLIB, so it uses the parallelization capabilities of spark; while the iterations of this loop are executed in series and not in parallel.

In [33]:

```
ks = sc.parallelize([5, 10, 15, 20, 30, 50, 100, 200])
# convert series data to rdd of values
training_data = data.tordd().map(lambda x: np.array(x[1]) ).cache()

def buildModelsParallel(data):
    # declare the collection of models
    bData = sc.broadcast(data.collect())

    models = ks.map(lambda k: KMeans(k).fit(bData.value))

    return models
#     return models
```

In [32]:

```
import time

timeInit = time.time()
models = buildModels(training_data)
print('time without using most of spark parallelization capabilities ',time.time() - timeInit, "seconds")
```

```
time without using most of spark parallelization capabilities  25.875746965408325 seconds
```

In [34]:

```
import time

timeInit = time.time()
models = buildModelsParallel(training_data)
print('time  using  spark parallelization capabilities ',time.time() - timeInit, "seconds")
```

```
time  using  spark parallelization capabilities  0.5337924957275391 seconds
```

**b)** We modified the code so as to make building the models for each k in ks happen in parallel with the others. We note an improvement in the time of running between buildModels and buildModelsParallel.

## d. Testing models & choosing the best one

Next, we evaluate the quality of each model. We use two different error metrics on each of the clusterings.

- The first is the sum across all time series of the Euclidean distance from the time series to their cluster centroids.
- The second is a built-in metric of the `KMeansModel` object.

**Question 11**

a) Write function `model_error_1` to calculate the sum of Squared Euclidean Distance from the Series objects to their clusters centroids. b) Comment the choice of the error function we use here. Is it a good error definition?

In [36]:

```
from scipy.spatial.distance import cdist

# calculate the Euclidean distance
# from each time series to its cluster center
# and sum all distances
def model_error_1(data, model):
    a=data.map(lambda x: cdist([x], [model.centers[model.predict(x)]], metric='euclidean')).sum()[0][0]
    return a
```

In [37]:

```
model_error_1(training_data, models[0])
```

Out[37]:

```
518.99645789549038
```

**b)** We think that the choice of sum of squared euclidean distance is not accurate. Actually, using this error function will imply that the error will necessarly decrease while increasing the k (because the clusters will contain fewer elements, and increasingly nearer to their cluster's centroid), so this will make it difficult to find the optimal k.

**Question 12**

a) Write function `model_error_2` to calculate the total of similarity of `Series` objects based on how well they match the cluster they belong to, and then calculate the error by inverse the total similarity. b) Similarly to the previous question, comment the choice of the similarity function.

In [38]:

```
# calculate the total of similarity of the model on timeseries objects
# and calculate the error by inverse the total similarity

# Estimate similarity between a data point and the cluster it belongs to.
def similarity(centers, p):
    if np.std(p) == 0:
        return 0
    return np.corrcoef(centers[np.argmin(cdist(centers, np.array([p])))], p)[0, 1]


def model_error_2(data, model):
    return 1.0 /np.array(data.map(lambda p: similarity(model.centers, p)).reduce(lambda a,b: a+b))
```

In [39]:

```
model_error_2(training_data, models[0])
```

Out[39]:

```
0.00022501829084323909
```

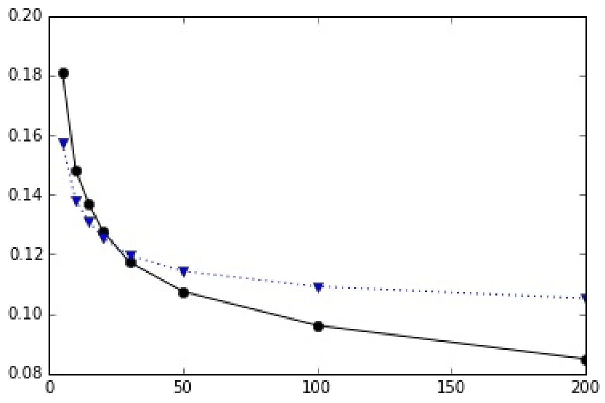
**b)** The similarity function is more accurate for our problem. Actually, it allows us to see the model that have the lower error (which means higher similarity), and that's the model where most of the points were classified well. Moreover, the similarity functions remove the voxels where no activity variation happens, which are mainly the voxels that are a part of the background and not the fish brain, and this is good for our problem.

**Question 13**

Plot the error of the models along with the different values of K in term of different error metrics above. From the figure, in your opinion, what is the best value for `K` ? Why ?

In [40]:

```
def testAndPlotTheResult(data, models):
    # compute the error metrics for the different resulting clusterings

    # errors of models when using function Sum Square Distance Error
    errors_1 = np.asarray([model_error_1(data, model) for model in models])

    # error of models when using similarity
    errors_2 = np.asarray([model_error_2(data, model) for model in models])

    # plot the errors with each value of K
    plt.plot(
        ks, errors_1 / errors_1.sum(), 'k-o',
        ks, errors_2 / errors_2.sum(), 'b:v')
    plt.show()

testAndPlotTheResult(training_data, models)
```



We've said above that the similarity function is more significant for our problem, so based on the similarity function curve we think that the best k to choose is around k=30 because the error remains more or less at the same level after it.

Determining the optimal $k$ is particularly troublesome for the $k$-Means algorithm because error measures based on distance decrease monotonically as $k$ increases. This arises because when $k$ is increased, each cluster is decomposed into more and more clusters, such that each point becomes closer to its cluster mean. In fact, in the extreme case where $k = N$, each point will be assigned to its own cluster, and all distances are reduced to nil. Cross-validation or using holdout data is also unlikely to be particularly effective in this case.

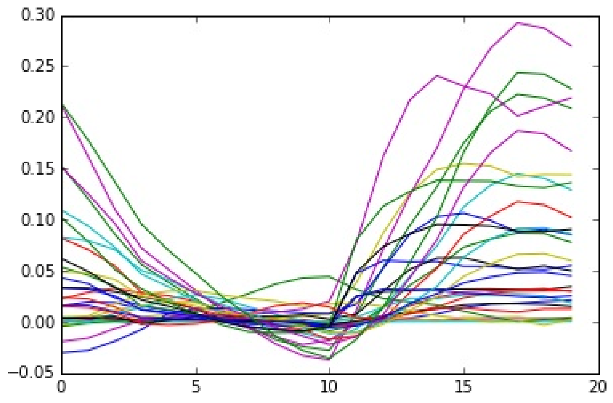
To this end, it is often worth assessing a model by measuring its impact on the overall aim of carrying out the clustering. For example, if we are carrying out $k$-means for grouping customers having similar taste and purchase history with the ultimate intent of making recommendations to customers, our objective function should measure how effective the recommendations are (perhaps using holdout data). An appealing aspect of using such a metric is that it is no longer guaranteed to behave monotonically with respect to $k$. We shall investigate this further in Question 20.

**Question 14**

Plot the centroids of the best model. Do you think that the result is good ?

In [41]:

```
# plot the best performing model
# k=30 is in the position that has the index 4 in the list
bestModel = models[4]
plt.plot(np.asarray(bestModel.centers).T)
plt.show()
```



There are two main things to notice : The first is that although the centroids curves have different allures, some of them have the same allure, yet with a difference in the amplitude. So we think that the result is more or less good, but it still need to be optimized for a better performance. Actually, we want the voxels with more or less the same activity variation to be together in the same cluster, which means reducing the number of clusters, but this is an issue because this will increase the error again.

## e. Visualizing the result

We can also plot an image of labels of neurons, such that we can visualize the group of each neuron.

**Question 15**

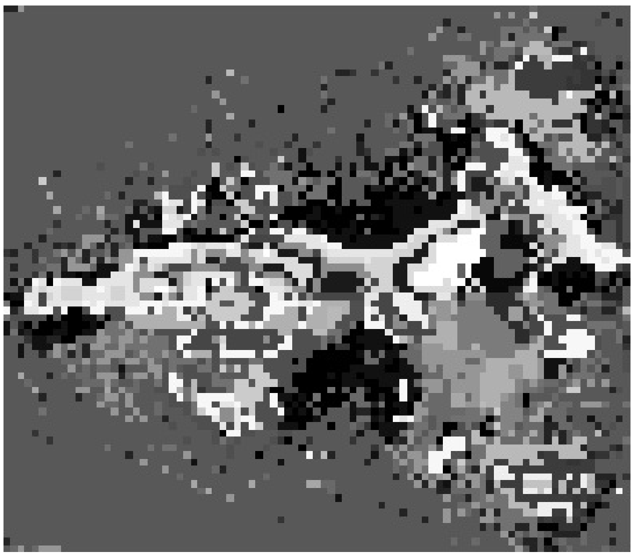
Complete the source code below to visualize the result of clustering.

In [42]:

```
# predict the nearest cluster id for each voxel in Series
labels = bestModel.predict(training_data)

# collect data to the driver
imgLabels = np.asarray(labels.collect()).reshape((2, 76, 87))

# consider the voxel of the first layers

draw_image(imgLabels[0,:,:])
```

Out[42]:

```
<matplotlib.image.AxesImage at 0x7f4d8a6a4ac8>
```



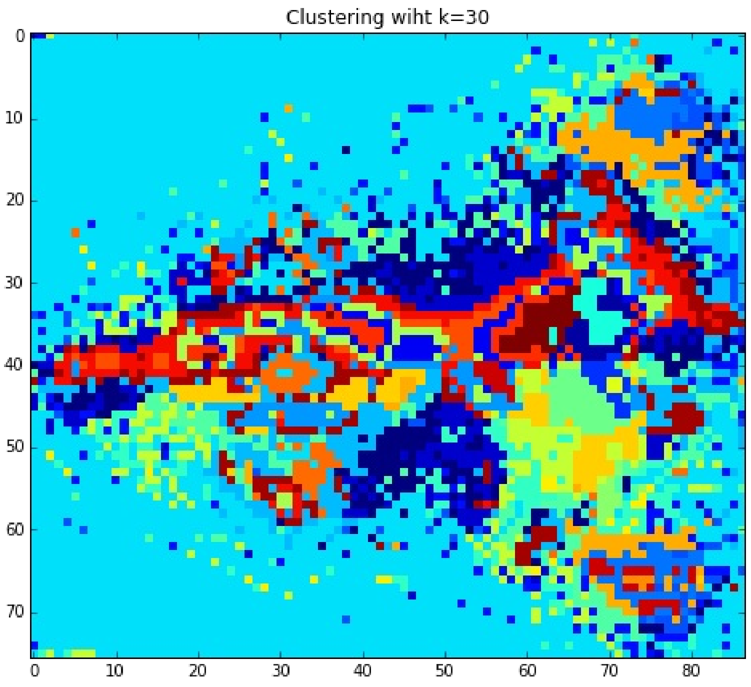
If we want to see the clustering in colours :

In [43]:

```
# predict the nearest cluster id for each voxel in Series
labels = bestModel.predict(training_data)

# collect data to the driver
imgLabels = np.asarray(labels.collect()).reshape((2, 76, 87))

# consider the voxel of the first layers
plt.figure(figsize=(8,8))
plt.imshow(imgLabels[0,:,:], interpolation='nearest', aspect='equal')
plt.title("Clustering wiht k=30 ")
```

Out[43]:

```
<matplotlib.text.Text at 0x7f4d885714e0>
```

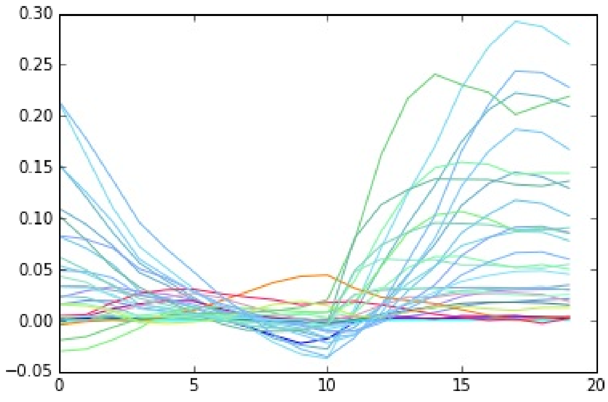


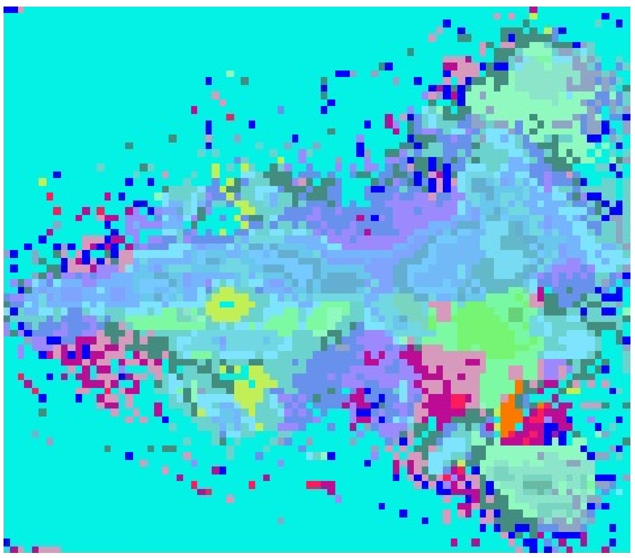
With the default color scheme, this figure is quite difficult to understand and to distinguish the groups according to their similar colors. So, we should have a smater color selection. The fact is, when we do clustering, it is often the case that some centers are more similar to one another, and it can be easier to interpret the results if the colors are choosen based on these relative similarities. The method `optimize` tries to find a set of colors such that similaries among colors match similarities among an input array (in this case, the cluster centers). The optimization is non-unique, so you can run multiple times to generate different color schemes.

In [44]:

```
from numpy import arctan2, sqrt, pi, abs, dstack, clip, transpose, inf, \
    random, zeros, ones, asarray, corrcoef, allclose, maximum, add, multiply, \
    nan_to_num, copy, ndarray, around, ceil, rollaxis

# these functions below are inspired mainly from Thunder-Project source code, v.0.6
# url: https://raw.githubusercontent.com/thunder-project/thunder/branch-0.6/thunder/viz/colorize.py

# Optimal colors based on array data similarity.
def optimize_color(mat):
        mat = np.asarray(mat)

        if mat.ndim < 2:
            raise Exception('Input array must be two-dimensional')

        nclrs = mat.shape[0]

        from scipy.spatial.distance import pdist, squareform
        from scipy.optimize import minimize

        distMat = squareform(pdist(mat, metric='cosine')).flatten()

        optFunc = lambda x: 1 - np.corrcoef(distMat, squareform(pdist(x.reshape(nclrs, 3), 'cosine')).flatte
n())[0, 1]
        init = random.rand(nclrs*3)
        bounds = [(0, 1) for _ in range(0, nclrs * 3)]
        res = minimize(optFunc, init, bounds=bounds, method='L-BFGS-B')
        newClrs = res.x.reshape(nclrs, 3).tolist()
        from matplotlib.colors import ListedColormap


        newClrs = ListedColormap(newClrs, name='from_list')

        return newClrs

# Blend two images together using the specified operator.
def blend(img, mask, op=add):
        if mask.ndim == 3:
            for i in range(0, 3):
                img[:, :, :, i] = op(img[:, :, :, i], mask)
        else:
            for i in range(0, 3):
                img[:, :, i] = op(img[:, :, i], mask)
        return img

def _prepareMask(mask):
        mask = asarray(mask)
        mask = clip(mask, 0, inf)

        return mask / mask.max()

# Colorize numerical image data.
def transform(cmap, img, mask=None, mixing=1.0):
        from matplotlib.cm import get_cmap
        from matplotlib.colors import ListedColormap, LinearSegmentedColormap, hsv_to_rgb, Normalize

        img = asarray(img)
        dims = img.shape

        if cmap not in ['polar', 'angle']:

            if cmap in ['rgb', 'hv', 'hsv', 'indexed']:
                img = copy(img)
                for i, im in enumerate(img):
                    norm = Normalize(vmin=None, vmax=None, clip=True)
                    img[i] = norm(im)

            if isinstance(cmap, ListedColormap) or isinstance(cmap, str):
                norm = Normalize(vmin=None, vmax=None, clip=True)
                img = norm(copy(img))

        if mask is not None:
            mask = _prepareMask(mask)

        if isinstance(cmap, ListedColormap):
            if img.ndim == 3:
                out = cmap(img)
                out = out[:, :, :, 0:3]
            if img.ndim == 2:
                out = cmap(img)
                out = out[:, :, 0:3]
        else:
            raise Exception('Colorization method not understood')

        out = clip(out, 0, 1)

        if mask is not None:
            out = blend(out, mask, multiply)

        return clip(out, 0, 1)


# generate the better color scheme
newClrs = optimize_color(bestModel.centers)
plt.gca().set_color_cycle(newClrs.colors)
plt.plot(np.array(bestModel.centers).T);

# draw image with the new color scheme
brainmap = transform(newClrs, imgLabels[0,:,:])
draw_image(brainmap)
```

```
Out[44]:
<matplotlib.image.AxesImage at 0x7f4da0e8d358>
```





## f. Improving the result by removing noise

One problem with what we've done so far is that clustering was performed on all time-series without data pre-processing. Many of time-series objects were purely noise (e.g. those outside the brain), and some of the resulting clusters capture these noise signals. A simple trick is to perform clustering after subselecting pixels based on the standard deviation of their time series. First, let's look at a map of the standard deviation, to find a reasonable threshold that preserves most of the relavant signal, but ignores the noise.

**Question 16**

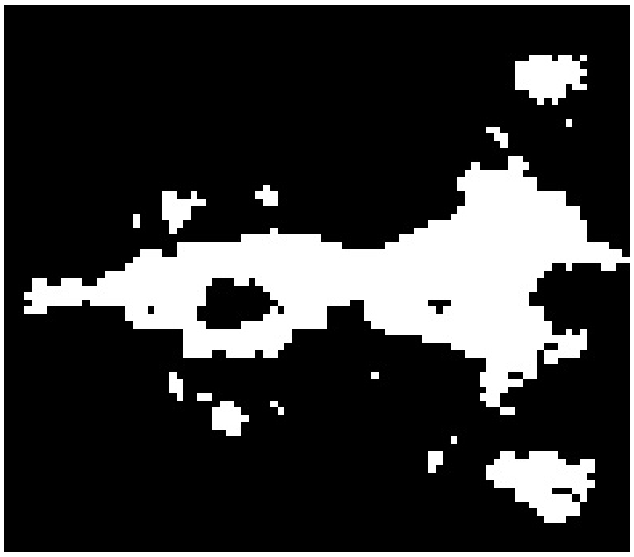
Try with different threshold of standard deviation to filter the noise. What is the "best value" that preserves most of the relavant signal, but ignores the noise ? Why ?

In [71]:

```
# calculate the standard deviation of each voxel
# then collect to the driver
stdMap = data.map(lambda x: x.std()).toarray()

# here we should try with many different values of threshold
# and choosing the best one
# visualize the map of the standard deviation after filtering
draw_image(stdMap[0,:,:] > 0.02)
```

Out[71]:

```
<matplotlib.image.AxesImage at 0x7f4d879cb7f0>
```



We tried different values of the threshold, and we think that 0.02 seems to be accurate because it reproduces most of the areas where there is relevant variation of activity, as shown in the visualization of standard deviation of the first layer in question 2.

**Question 17**

Filter your data such that we only keep the voxels that have the standard deviation bigger than the threshold in question 16.

In [46]:

```
####!@SOLUTION@!####
from numpy import std
# remove series object that has the standard deviation bigger than a threshold
filtered = data.filter(lambda x: x.std() >0.02).cache()
print(filtered.shape)
```

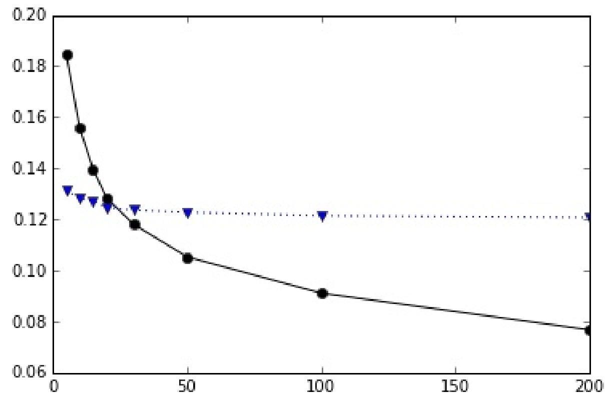
```
(2299, 20)
```

**Question 18**

Re-train and choose the best models with different values of `K` on the new data.

In [47]:

```
filterdata=filtered.tordd().map(lambda x: np.array(x[1]) ).cache()
models = buildModels(filterdata)

testAndPlotTheResult(filterdata, models)
```



We think that a good value of k can be here **k=10** , as the curve remains almost stable after it (the similarity function curve).

**Question 19**

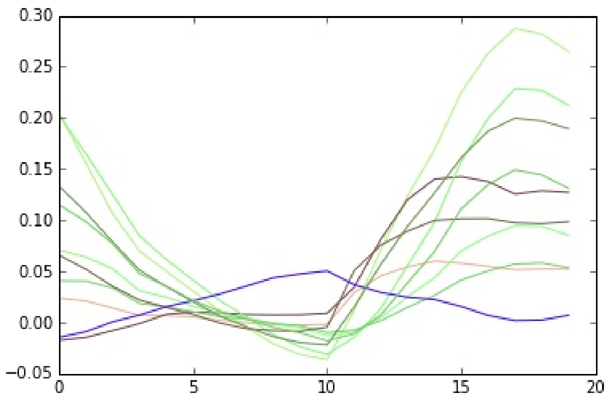
a) Plot the centroids of the best model with a smart color selection. b) Plot the result of the clustering algorithm by a color map of voxels. c) Comment about your figures.

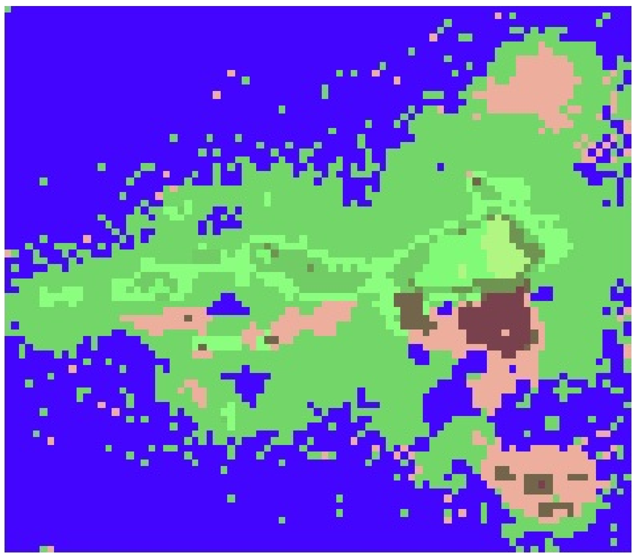
In [48]:

```
bestModel = models[1]
newClrs = optimize_color(bestModel.centers)
plt.gca().set_color_cycle(newClrs.colors)
plt.plot(np.array(bestModel.centers).T)




# predict the nearest cluster id for each voxel in Series
labels = data.map(lambda x: bestModel.predict(x))

# collect data to the driver
imgLabels = labels.toarray().reshape(2, 76, 87)

brainmap = transform(newClrs, imgLabels[0, :, :])
draw_image(brainmap)
```

Out[48]:

```
<matplotlib.image.AxesImage at 0x7f4da0429908>
```





We got more or less a similar result as previously for the centroids curves, but obviously with less curves because we have diminished the number of clusters.

For the clustering, it looks more homogenous as for example , we can see that almost all the voxels where there is no activity (background of the picture or voxels of the brain but in which the standard deviation is null) were more or less classified together.

## g. Improve the visualization by adding similarity

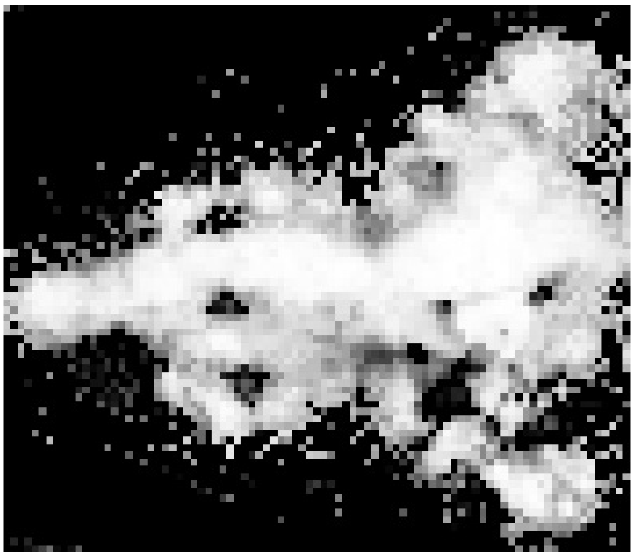
These maps are slightly odd because pixels that did not survive our threshold still end up colored as something. A useful trick is masking pixels based on how well they match the cluster they belong to. We can compute this using the `similarity` method of `KMeansModel`.

In [49]:

```
sim = data.map(lambda x: similarity(bestModel.centers, x))

imgSim = sim.toarray()

# draw the mask
draw_image(imgSim[0,:,:], cmap='gray', clim=(0,1))
```

Out[49]:

```
<matplotlib.image.AxesImage at 0x7f4da0cae940>
```



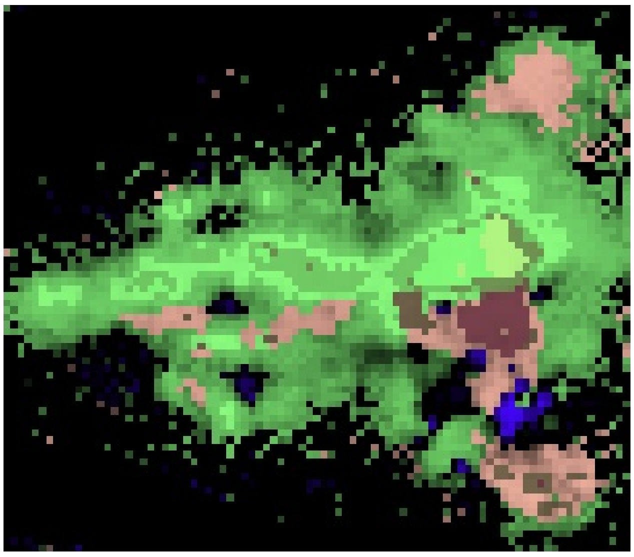
And, it can be used as a linear mask on the colorization output

In [50]:

```
brainmap = transform(newClrs, imgLabels[0,:,:], mask=imgSim[0,:,:])
draw_image(brainmap)
```

Out[50]:

```
<matplotlib.image.AxesImage at 0x7f4da0d1c400>
```



**Question 20**

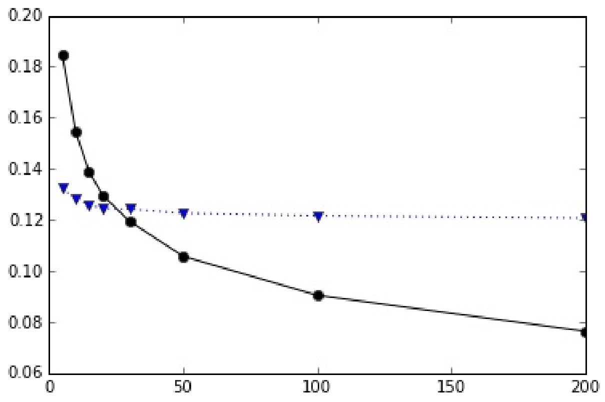
Since in the usecase we build and test the model from the same data, it can lead to overfitting problems. To avoid that, we can divide the data into training set and testing set. Note that each neuron occurs only one time in the data. So, we can not divide the data by dividing the neurons. Instead, we can divide the states of neurons into two different sets. Let's try with this approach and show the result.

In [51]:

```
arr = np.arange(20)
np.random.shuffle(arr)
# decompose randomly the states indexes between a training set indexes of 15 states and a test set indexes o
f 5 states

training_data_index = arr[:15]
test_data_index = arr[5:]

# filter the data based on our threshold

data_filtered = data.tordd().map(lambda x: np.array(x[1])).filter(lambda x: x.std() > 0.02).cache()

# split the data into training set and test set

training_data = data_filtered.map(lambda x: np.array([itm for ind, itm in enumerate(x) if ind in training_da
ta_index])).cache()

test_data = data_filtered.map(lambda x: np.array([itm for ind, itm in enumerate(x) if ind in test_data_index
])).cache()

models = buildModels(training_data)
```

In [52]:

```
models = buildModels(training_data)
testAndPlotTheResult(training_data, models)
```



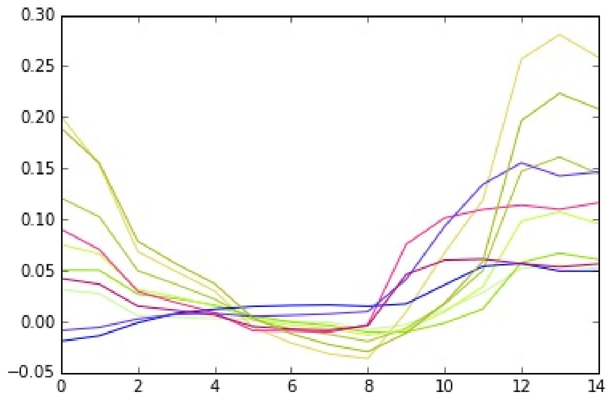
Again we take **k=10** .

In [53]:

```
bestModel = models[1]
newClrs = optimize_color(bestModel.centers)
plt.gca().set_color_cycle(newClrs.colors)
plt.plot(np.array(bestModel.centers).T)
```

Out[53]:

```
[<matplotlib.lines.Line2D at 0x7f4da0ebcba8>,
 <matplotlib.lines.Line2D at 0x7f4d879a9b00>,
 <matplotlib.lines.Line2D at 0x7f4d879a90b8>,
 <matplotlib.lines.Line2D at 0x7f4d879a9518>,
 <matplotlib.lines.Line2D at 0x7f4d879a95c0>,
 <matplotlib.lines.Line2D at 0x7f4d879a96d8>,
 <matplotlib.lines.Line2D at 0x7f4d86fb0710>,
 <matplotlib.lines.Line2D at 0x7f4d86fb0080>,
 <matplotlib.lines.Line2D at 0x7f4d86fdfba8>,
 <matplotlib.lines.Line2D at 0x7f4d86fdf6d8>]
```



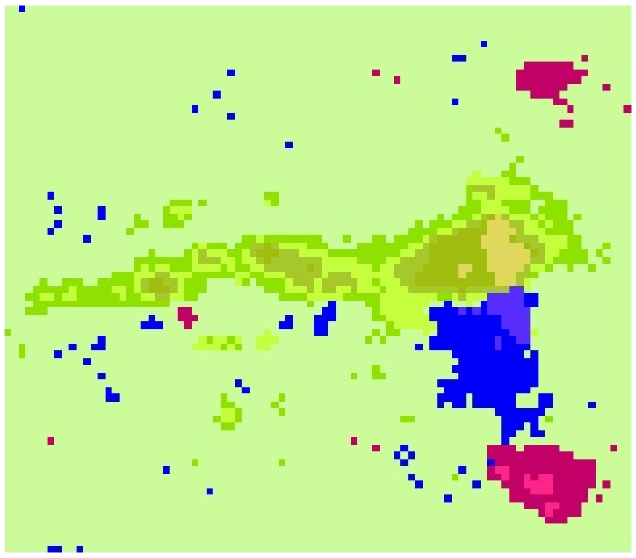
Let's visualize the result of the clustering on the training set using the smarter colour selection:

In [64]:

```
training_data = (data.map(lambda x: np.array([itm for ind, itm in enumerate(x) if ind in training_data_index
]))
                 .cache())

labels = training_data.map(lambda x: bestModel.predict(x))

# collect data to the driver
imgLabels = labels.toarray().reshape(2, 76, 87)

# draw image with the new color scheme
brainmap = transform(newClrs, imgLabels[0, :, :])
draw_image(brainmap)
```

Out[64]:

```
<matplotlib.image.AxesImage at 0x7f4da0310128>
```



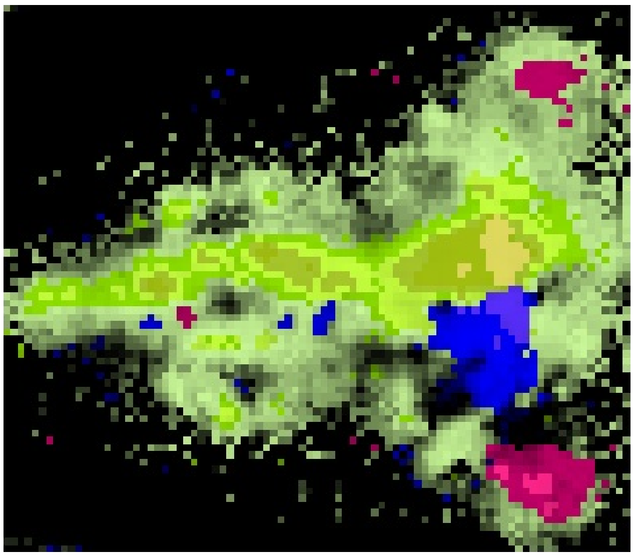
and let's now improve the visualization by adding similarity :

In [65]:

```
sim = training_data.map(lambda x: similarity(bestModel.centers, x))

imgSim = sim.toarray()


brainmap = transform(newClrs, imgLabels[0,:,:], mask=imgSim[0,:,:])
draw_image(brainmap)
```

Out[65]:

```
<matplotlib.image.AxesImage at 0x7f4d885c73c8>
```

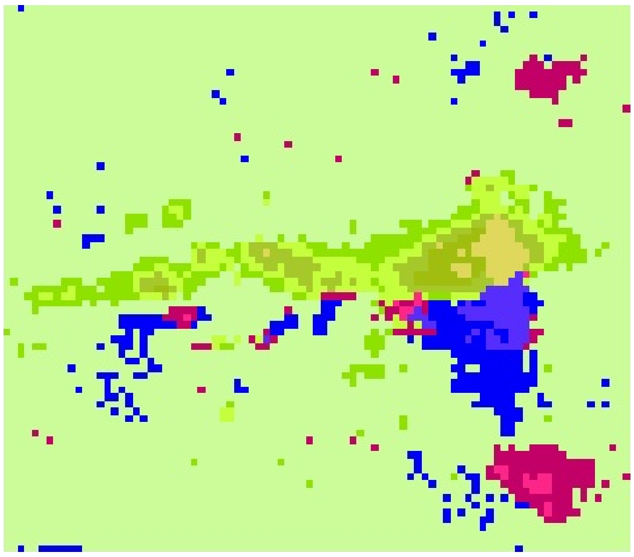


Let's again visualize the result of the clustering but now **on the test set** using the smarter colours selection:

```
In [67]:
```

```
test_data = (data.map(lambda x: np.array([itm for ind, itm in enumerate(x) if ind in test_data_index]))
                 .cache())

labels = test_data.map(lambda x: bestModel.predict(x))

# collect data to the driver
imgLabels = labels.toarray().reshape(2, 76, 87)

# draw image with the new color scheme
brainmap = transform(newClrs, imgLabels[0, :, :])
draw_image(brainmap)
```

```
Out[67]:

<matplotlib.image.AxesImage at 0x7f4d889dce10>
```



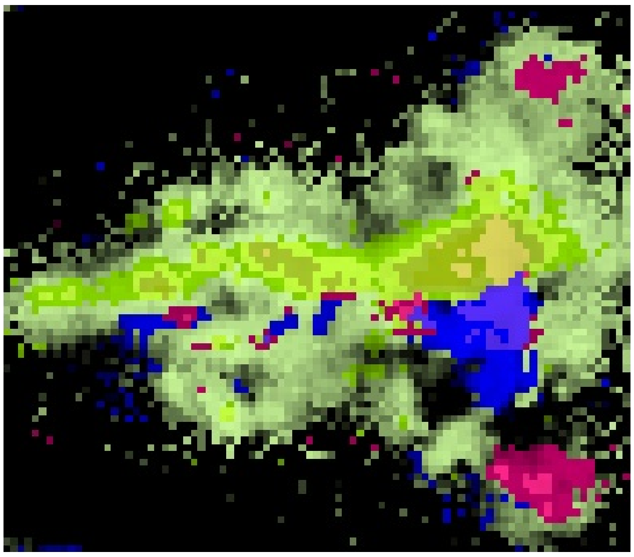
and let's again improve the visualization by adding similarity :

In [68]:

```
sim = test_data.map(lambda x: similarity(bestModel.centers, x))

imgSim = sim.toarray()


brainmap = transform(newClrs, imgLabels[0,:,:], mask=imgSim[0,:,:])
draw_image(brainmap)
```

Out[68]:

```
<matplotlib.image.AxesImage at 0x7f4d878b2e10>
```



We think that the model have more or less a good performance as it gives a similar clustering for both, training data and test data. So we think that overfitting has been avoided.

**Question 21**

Is using K-Means the best choice for a clustering algorithm? Comment the choice and suggest alternatives. For example, look at [Mixture Models](https://en.wikipedia.org/wiki/Mixture_model) and, if you have time, propose an alternative clustering technique.

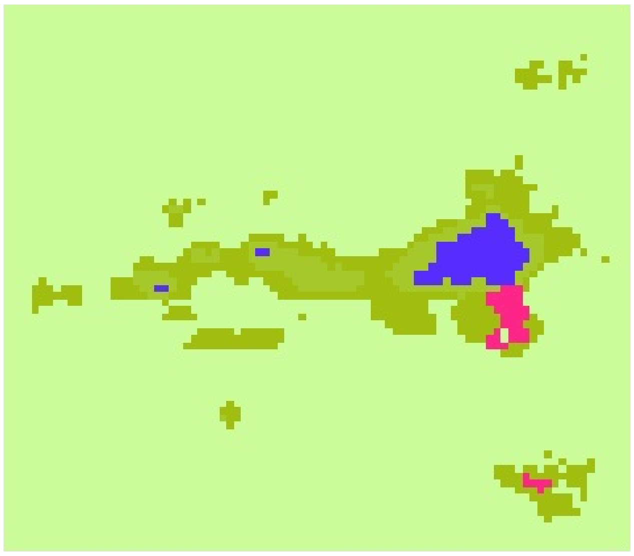
**NOTE** Mixture models will be covered in the ASI course in greater detail.

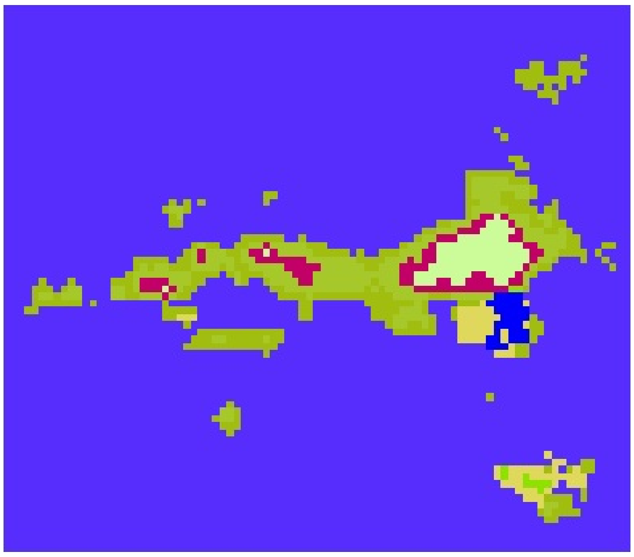
**The K-means** algorithm is performant as the computations are done in an efficient and fast way. However, one of the disadvantages of this approach (and that we have faced during the whole lab) is that the K-means requires prior specification of k (the number of clusters), while we don't have any scientific basis thanks to which we can determin exactly the optimal number of clusters, so we managed during the lab to choose a good number of k by trying to minimize some error function while trying to have the less possible number of clusters.

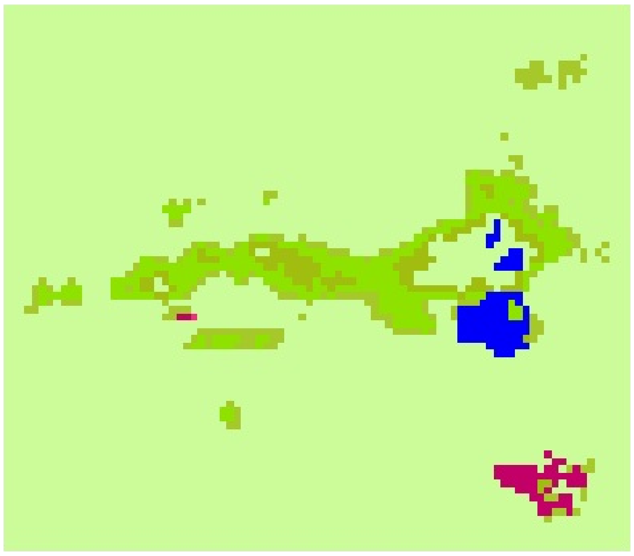
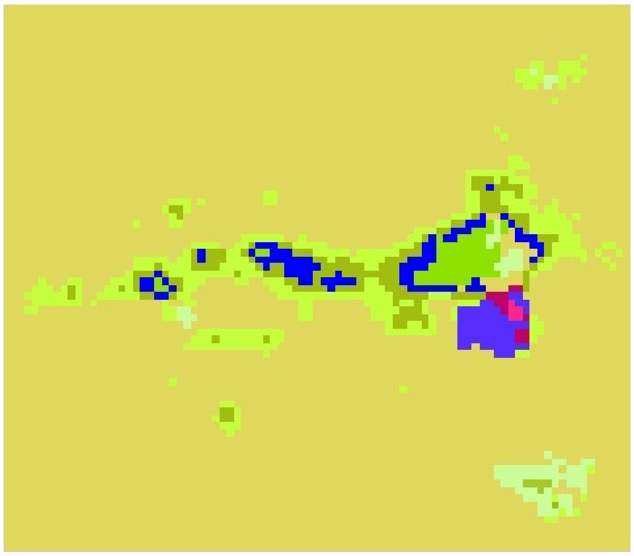
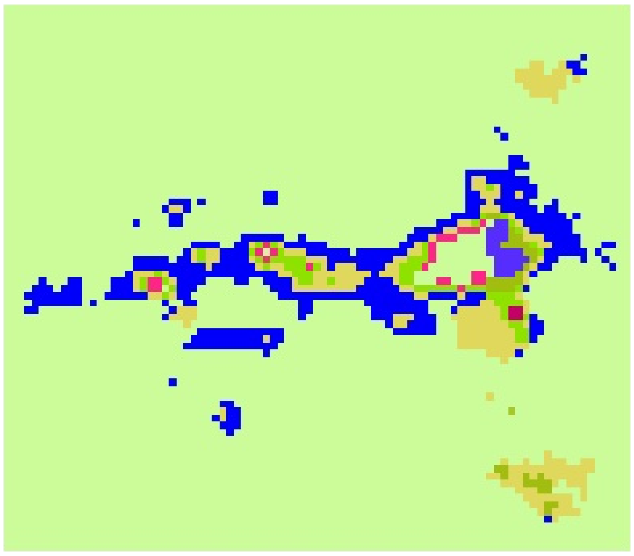
Let's try some mixture models ; for instance the gaussion mixture model :

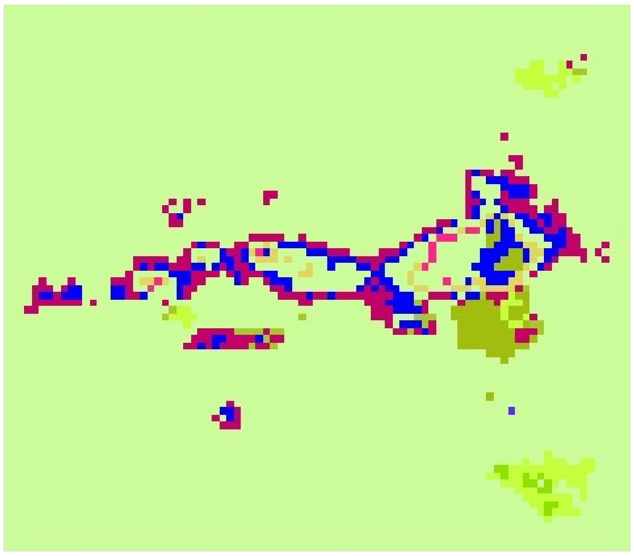
In [89]:

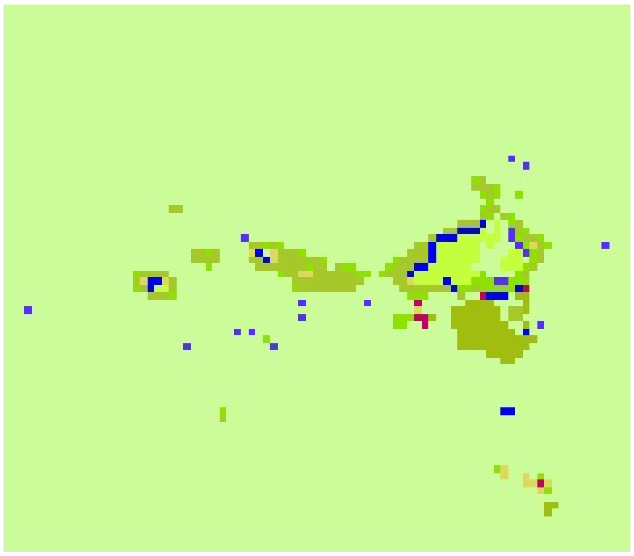
```
from sklearn import mixture
for k in ks :
    model = mixture.GMM(n_components=k)
    model.fit(training_data.flatten().toarray())
    labels = model.predict(training_data.flatten().toarray()).reshape(2,76,87)

    # draw image with the new color scheme

    brainmap = transform(newClrs, labels[0, :, :])

    draw_image(brainmap)
```

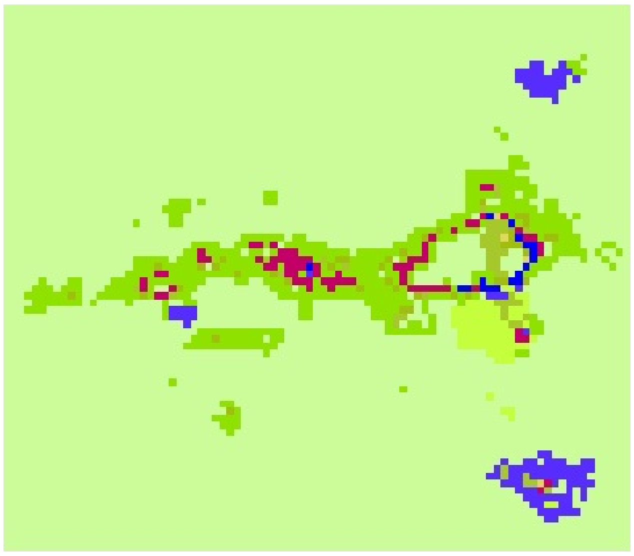










We just tried the gaussian mixture model with several values for the number of components, it seems to work better when this number is between 10 and 30.

## 4. Summary

We studied Thunder and its important methods to work with images, such as `Image`, `Series` and how to apply them to a use case. In the use case, we used the K-Means algorithm to cluster the neurons without prior knowledge of what a good choice of K could be. Subsequently, we introduced some techniques for improving the initially obtained results, such as removing noise and considering similarity.

## References

Some of the examples in this notebook are inspired from the documentation of Thunder (http://docs.thunder-project.org/).

In [ ]:

```
```

In [ ]:

```
```