Pattern-matching Unit for Medical Application (PUMA)

Report (Nov 2019 - May 2020)

Outline

- Intro: MRI data, dictionary specification
- Principal Component Analysis and an iterative method
- Simulating the Associative Memory
- Using the viewsharing signal
- Next steps

Description of input data from MRI

- The MRI images typically have 200x200x200 pixels (8M pixels), sampled in 800 time points
  - "aliasing" effects, due to the reconstruction process
  - Images suffer also from noise
  - These two effects are source of differences between the dictionary and pixel components
- Singular value decomposition (https://en.wikipedia.org/wiki/Singular_value_decomposition) is used to reduce the 800 time points of each pixel to a small number of SVD complex values. For usage with the AM we started with using a SVD with 8 complex values

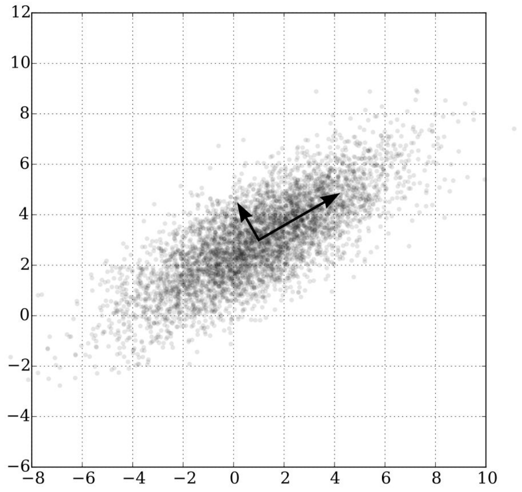
Description of input data from MRF (cont.)

- All values in the acquired images (and in the dictionary) are normalized and dephased
    - The normalization of the SVD vector has no physical meaning, so we normalize all the data to a norm=1
    - The overall phase of the SVD vector has no physical meaning, so we change the overall phase of the vector to make the first entry real
- After dephasing, the dictionary data has only the real component. (The imaginary components are all 0)
- Since we are comparing the signal images with dictionary data, we take the real part of the signal images and we discard the imaginary part.

Description of the used dictionaries

- We used a dictionary provided by Imago7
- 2 dictionaries: normal, viewsharing
- First one generated with the following parameters:
  - $t_{1\_min} = 0.02$, $t_{1\_max} = 4$, $t_{2\_min} = 0.001$, $t_{2\_max} = 0.4$, $step_{t1} = 0.01$, $step_{t2} = 0.001$
  - ~156k entries
- Second one generated with the following parameters:
  - $t_{1\_min} = 0.1$, $t_{1\_max} = 3$, $t_{2\_min} = 0.01$, $t_{2\_max} = 0.6$, $step_{t1} = 0.01$, $step_{t2} = 0.001$
  - ~156k entries
- Right now, only 2 parameters (t_1, t_2)
  - the goal of the project is to be able to use much larger dictionaries based on a larger number of parameters

Principal component analysis



Look at the correlation in data, determine eigen vectors and eigen values of the correlation matrix….

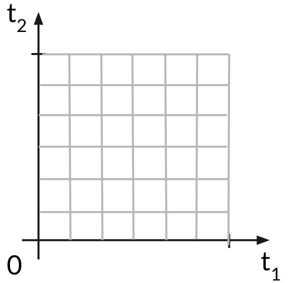
We focus on the determination of parameters t_1 and t_2 with linear approximation.

$$[t'_1 \quad t'_2] = (x - \overline{D}) \times M$$

Where $\boldsymbol{x}$ is the vector of input data SVD values, $\boldsymbol{D}$ and $\boldsymbol{M}$ are a vector and a matrix determined from the training phase. They are determined as the values that minimize the mean square deviation between t_1' and t_1 in the dictionary (respectively, between t_2' and t_2).

A PCA-like iterative method

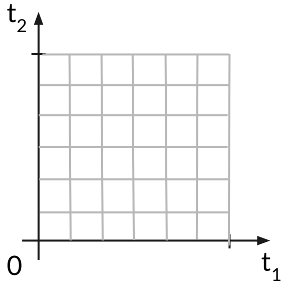
- Just "getting our feet wet"...
- Partition the (t_1, t_2)-space into n^2 portions
  - Values of (t_1, t_2) taken from the dictionary LUT
- For each portion *i*:
  - Let T_i be the portion of the LUT
  - Let D_i be the corresponding portion of the dictionary
  - Calculate M_i as shown in the formula
- Calculate the "global" matrix M_{glob} starting from the whole LUT *T* and dictionary *D*



$$M_i = \frac{(T_i - \overline{T_i}) \times (D_i - \overline{D_i})}{D_i^2}$$

$$M_{glob} = \frac{(T - \overline{T}) \times (D - \overline{D})}{D^2}$$

A PCA-like iterative method



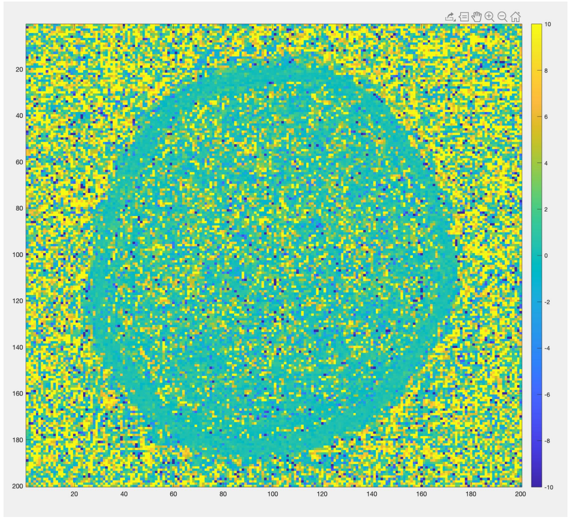
- Take a vector of *m* SVD components *x* (i.e. a pixel of the signal)
- Find a first rough estimate of the corresponding (t_1, t_2) values using M_{glob}

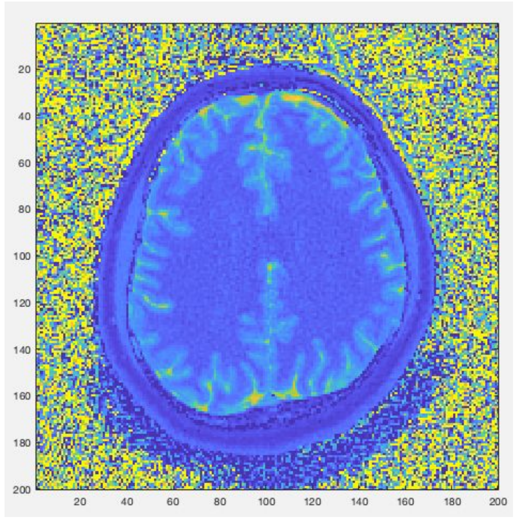
$$[t'_1 \quad t'_2] = (x - \overline{D}) \times M_{glob}$$

- The pair (t'_1, t'_2) identifies a point in the plane. This point is inside the *j*-th portion
- Find a second estimate $[t''_1 \quad t''_2] = (x - \overline{D_j}) \times M_j$
- Two things can happen:
  - (t''_1, t''_2) is inside portion *j* again → STOP!
  - (t''_1, t''_2) is inside a neighboring portion *k* → repeat procedure using D_k and M_k ...
  - Define other stop conditions to avoid cycling indefinitely
- The advantage w.r.t. dot-product match: fewer calculations
  - pre-calculate the n^2 matrices and vectors
  - calculations for each pixel are usually less than 10 steps (except when cycles happen)

Results

- ...not that good (see images below, left: PCA-like iterative method, right: dot-product match)
- Nevertheless, we started gaining some confidence with the subject



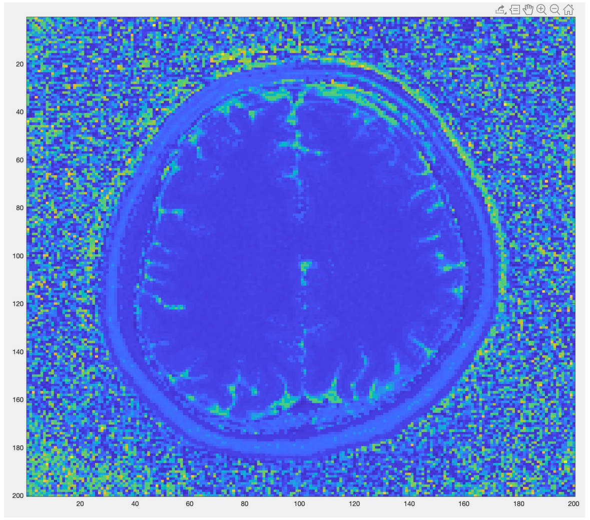


How can we improve?

- Take a page from machine learning!
- Discard the dictionary, use (a subset of) the signal as training data
- We took 5 slices of the signal (50, 75, 100, 125, 150) and built a training dataset from them
  - Using Imago7's dot-product match values as truth
- Try to reconstruct slice 110

Results

- Difference vs. dot product:
  - min diff $t_1 = 0$ $t_2 = 0$
  - max diff $t_1 = 3.61$ $t_2 = 0.38$
  - mean diff $t_1 = -0.071$
    $t_2 = -0.0077$
  - standard deviation diff
    $t_1 = 0.1933$ $t_2 = 0.0295$



- MUCH better!
- Surely has its own problems (e.g. overfitting)
- Highlights something that we encountered repeatedly:

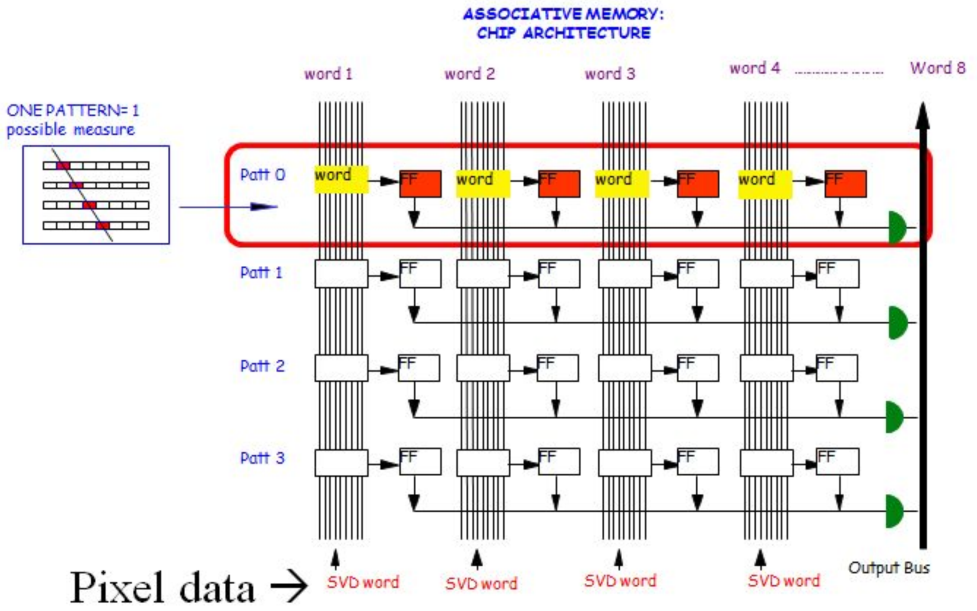
Dictionary and signal "don't speak the same language"!

- i.e. SVD components of the dictionary entries are not in the same ballpark as the corresponding SVD components of the signal pixels
- This can be a problem also for the associative memory

The Associative Memory

- an ASIC
- each pattern represents one or more dictionary entries
- input data is compared in parallel with all patterns on the chip

- currently: 128k pattern/chip, 100 MHz
    - 10^{14} comparisons per second
- in the future: 384k pattern/chip, 250 MHz



Using AM in the PUMA project

- After normalization, the *m* SVD components are in the range [-1, 1]. Now we are using m=8.
  - Except the first component, which is in the range [0, 1]
- The AM uses integer numbers, SVD components are floating-point numbers
- Need a mapping between float and int → **binning** each component and use the index of the bin
  - 10 bins for component 1
    - any entry having value *0 ≤ x < 0.1* gets in bin 1
    - any entry having value *0.1 ≤ x < 0.2* gets in bin 2
    - …
  - 10 bins for component 2-8
    - any entry having value *-1 ≤ x < -0.8* gets in bin 1
    - any entry having value *-0.8 ≤ x < -0.6* gets in bin 2
    - …
- Each entry of the dictionary can therefore be represented as a string of *m* integers → the **pattern**
  - Each integer represents the bin that holds the entry's value for that component
- Different entries can be represented by the same pattern

Using AM in the PUMA project

- Represent each pixel of the signal in the same way: *m* integers corresponding to the bins
- Finding a match is now simply a “string” comparison
    - where a “string” in this context is a vector of 8 integers
- Can choose to accept a match only if all *m* components are in the same bins OR allow partial correspondence, e.g. 7/8 or 6/8

Simulating the AM

- A simulator exists for the "original" version of the AM chip (used for ATLAS)
- Written in C++
- Slightly cumbersome to use in this "exploration" phase
  - must convert dictionary entries into a *pattfile*
  - must convert signal into a specific format
  - output is not an image (or Matlab matrix) → comparison w/ Imago7's results is difficult
- Will certainly be used in the laboratory tests phase

Simulating the AM

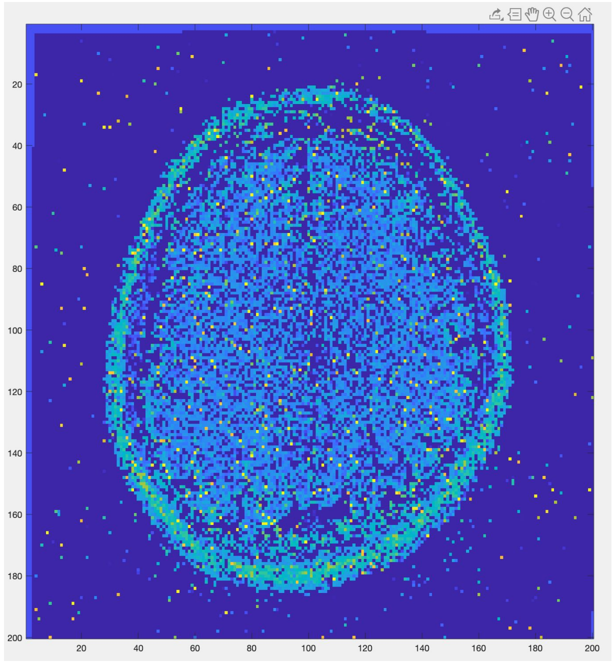
- I wrote our version of the C++ simulator
- Input and output are MATLAB matfiles
- Easy to do back-and-forth between the two environments
- Can extract whichever data we want from the pattern generation and matching processes
- In current version, each pattern is associated with a value of t_1, t_2
  - Obtained by averaging the values of t_1, t_2 corresponding to the dictionary entries that generated the pattern
  - Just to have an idea if what we're doing goes in the correct direction
- In final version: each pattern associated to the list of dictionary entries that generated it
  - Dot-product match to find the closest entry
- Not optimized yet (slow)

First results from the AM simulator

- Using the "plain" dictionary (no noise added) and the standard reconstructed signal
- Pretty poor…
    - we know of two key differences in acquired images
        i. The images include noise. The real noise is noise in the k-space measurements.
        ii. The images in the real space include aliasing effects
- Over a whole 200x200 slice, we matched **40** pixels → 0.1% of pixels were matched...
- We then tried allowing partial correspondence: 7/8 and 6/8
- Way better, but still insufficient (5199 matched pixels, 13%)

Adding Gaussian noise

- Next step: adding some noise to the dictionary entries
- Goal is always to bring entries and signal components "closer"
- We started by adding 5% of gaussian noise
    - Procedure for adding *n%* noise to *m* SVD components:
        - Generate a vector of *m* values according to the normal (Gaussian) distribution
        - multiply this vector by the noise percentage
        - add the noise component-by-component
- A little better: ~12000 matched pixels (30%)

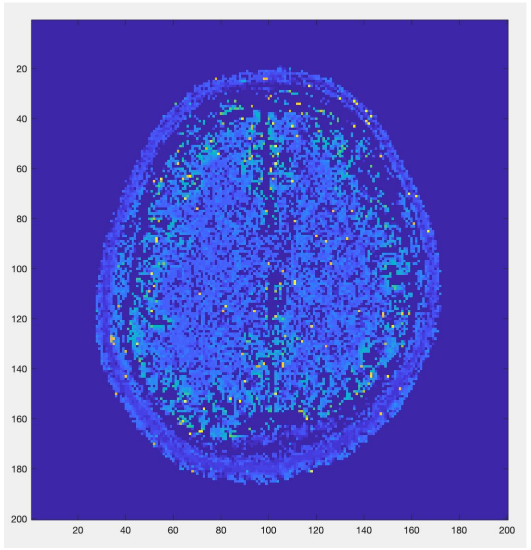


A breakthrough: viewsharing

- Despite our best efforts, we still couldn't match even half of the signal's pixels
  - The number of generated patterns from the dictionary is xx starting from yy dictionary entries. This means that dictionary entries often fall on the same patterns, in other word the pattern bank has good efficiency for the dictionary
- As we said before: problem is that dictionary and signal components "do not speak the same language"
- Up until now, we used a signal reconstructed in the "classical" way (Imago7's Example 8 brain)
- Matteo and Luca proposed to use a signal reconstructed using the ***viewsharing*** technique
  - Also suggested to use a **mask** to restrict calculations only on "meaningful" pixels
  - In slice 110 (our "testing" slice) there are only 18829 meaningful pixels (instead of 40000 for the whole slice)
  - Efficiency will thus be calculated as (n° of matched pixels)/18829

Viewsharing signal

- MUCH better results!
- Dictionary without noise + partial correspondence: **11727** matched pixels over 18829!



Viewsharing signal: refinements

- We added noise to the dictionary entries
- Tested different levels of gaussian noise (3%, 10%, 30%)
- To increase entropy, we did the following:
    - Instead of only adding a flat %, we also added *X* intermediate levels of noise
        - E.g. instead of adding 10% noise, we added 1%, 2%, ... , 10%
    - We repeated the noise application process *Y* times
- Start from dictionary of size $n \rightarrow$ end up with a dictionary of size $n * X * Y$
- Logged: nº of generated patterns, nº of matched pixels, deltas between our match and dot-product match

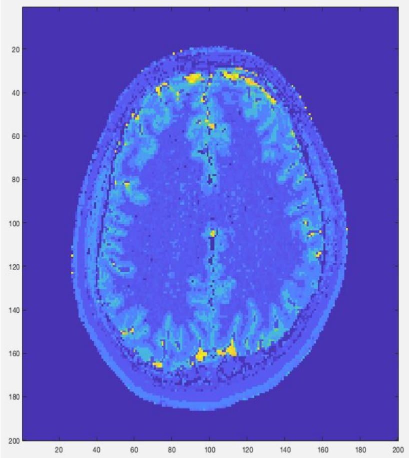
Viewsharing signal: results

- Pattern generation details:
  - 10 runs x 10 levels of 10% maximum noise
  - size of the bins (from lowest component to highest): 10, 10, 9, 9, 8, 8, 7, 7
  - started from a dictionary with 171981 entries
  - total number of unique patterns generated: **1032**
- Image processing details:
  - slice 110 of viewsharing signal
  - usual mask (18829 pixel to match)
  - we accept matches over 8, 7 or 6 components

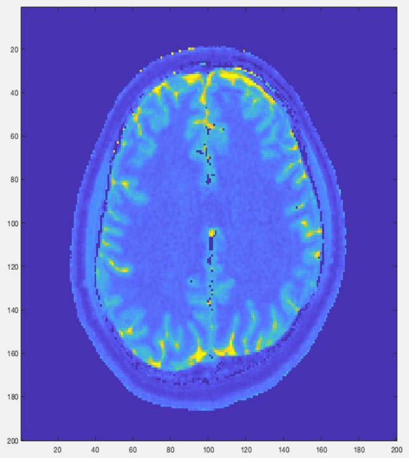
Viewsharing signal: results

- **18823** pixels matched → 99.96% efficiency!
- Comparison with dot-product result → delta between our matched matrix and dot-product matrix
  - Mean values of the delta: t_1 = -0.1026, t_2 = 0.0003
  - Standard deviations: t_1 = 0.2936, t_2 = 0.0719

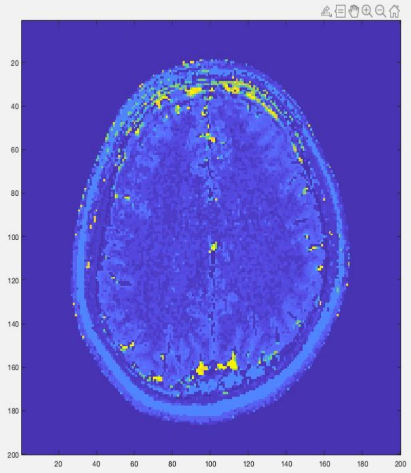
Viewsharing signal: results



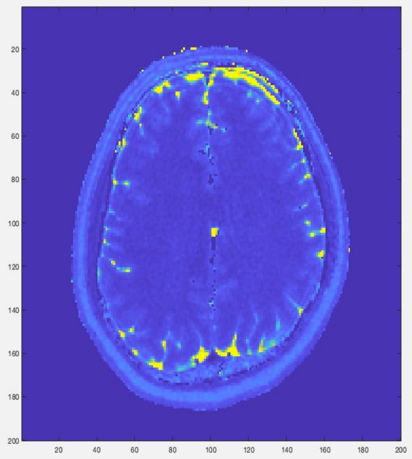
t_1 (AM sim)



t_1 (dot pr)



t_2 (AM sim)



t_2 (dot pr)

A possible approach to be explored

- First use AM to match all points within the mask
- For the ~0.04% points without match (AM inefficiency) use the standard scalar product approach
    - negligible computational cost
- Refine AM based t_1, t_2 estimates using PCA, or using the scalar product among the dictionary entries that correspond to the pattern

- Open points
    - check resolutions with AM+PCA
    - check scalability to more than 2 parameters
    - further optimize the AM comparison

Objectives and next steps

- Objectives:
  - Find out if AM can be used to accelerate t_1, t_2 extraction from a MRI signal
  - Find the best way to populate the AM's banks (i.e. add noise/other transformations to dictionary entries)
- Next steps:
  - Fix bugs in the simulator and perform some optimizations
  - Start to test the AM match with the existing AM boards
  - Study alternative approaches to include noise directly in the patterns

Questions

- Is the viewsharing calculation computationally intensive or light?
- Is the “mask” computationally intensive or light?
    - How many pixel to be matched are spared when using the mask?
- What is the input needed to make the mask? Can it be done before the match with the dictionary is performed?