NeuroScopeIO

R Interaction for NeuroScope Products

Josh Taylor

05-July-2020

Contents

Preface

The NeuroScopeIO package provides input/output functionality to read Neuro-Scope learning products into an R working environment, and partial functionality to write such products back to file. We will use a 10-dimensional SHGR cube and its associated SOM products to demonstrate this functionality. We consider "NeuroScope" products to comprise the following list:

- a data cube, in Khoros viff format
- an include file (.incl)
- an nna file (.nna)
- a weight cube (.wgtcub) file, in Khoros viff format

- a nunr file (.nunr)
- a ctab file
- a .cadj file, in Khoros viff format. Note: `ann-SOMconsc` recall produces the .cadj file in Khoros KDF format. This must be converted to viff format manually in order to import with NeuroScopeIO.

This document demonstrates common uses of functions in the NeuroScopeIO package, but does not exercise every function in the package. Complete documentation is available (both in stanard man pdf format, as well as interactive `?<function_name>` format), so any of the package functions can be explored further via these help options.

1 Collect Product Paths

To make this process ".parm-file-like", collect the paths of each product to start:

```
data_file = "~/Google Drive/Datasets/SGRW/ds/100d/SGRW-100d_11class_cov100-500.viff"
include_file = "~/Google Drive/Datasets/SGRW/ds/SGRW_11class.incl"
nna_file = "~/Google Drive/Datasets/SGRW/ds/100d/SGRW-100d.nna"
wgtcub_file = "~/Google Drive/Datasets/SGRW/dp/100d/recall/SGRW-100d_11class_cov100-500_20x2
nunr_file = "~/Google Drive/Datasets/SGRW/dp/100d/recall/SGRW-100d_11class_cov100-500_20x20.
ctab_file = "~/Google Drive/Datasets/SGRW/ds/SGRW.ctab"
cadj_file = "~/Google Drive/Datasets/SGRW/dp/100d/recall/SGRW-100d_11class_cov100-500_20x20.
```

2 Importing Data Products

2.1 Data Cubes

The function `NeuroScopeIO::read_khoros_viff` function accepts two arguments: `filename`, which is a path to the viff cube to import, and `verbose`, which is boolean. If `verbose=T` then the pixel spectra of the four "corners" of the image cube are printed to the R console window, which can be helpful for spot checking your import. The return value is a 3d data cube (array) in R, whose (1,1) element contains the spectral from pixel (1,1) in the image, and so on.

```
X = NeuroScopeIO::read_khoros_viff(filename = data_file, verbose = T)
## NW corner of cube:
## 462.905417 467.139347 451.875788 470.143694 471.996632 470.979375 467.936955 465.556848 4
## NE corner of cube:
## 804.353130 821.655741 816.815530 822.657375 822.450648 805.601848 822.200685 830.811183 8
## SW corner of cube:
## 280.030129 260.716222 258.803367 221.690254 241.651300 246.235956 251.543549 233.403980 2
## SE corner of cube:
## 694.161150 706.606234 717.338467 717.277809 735.129885 753.215351 747.655152 761.917667 7
str(X)
```

```
##  num [1:128, 1:128, 1:100] 463 466 466 499 485 ...
```

Note that the function `read_khoros_viff` is internally calling another function of the NeuroScopeIO library to read and interpret the Khoros viff header: `read_khoros_header`. It is rarely necessary to invoke this function by itself.

We can strip out and save the cube dimensions for later use:

```
img_x = ncol(X)
img_x
## [1] 128
img_y = nrow(X)
img_y
## [1] 128
img_z = dim(X)[3]
img_z
## [1] 100
```

Most base R functionality is built around **data matrices**, not arrays, so we need to convert the 3d cube we just imported to a data matrix. Use the command:

```
X = NeuroScopeIO::convert_datcub_to_datmat(datcub = X)
str(X)
##  num [1:16384, 1:100] 463 489 490 467 470 ...
```

Data matrices have nrows = img_x*img_y and ncols = img_z. The data matrix is populated in band-sequantial order (top to bottom, left to right), so that row 1 corresponds to the spectra in pixel (x=1,y=1), row 2 corresponds to pixel (x=2,y=1), and so on.

If desired at some point, any data matrix can be converted back into a data cube via

```
Xcube = NeuroScopeIO::convert_datmat_to_datcub(datmat = X, img_x = img_x, img_y = img_y)
str(Xcube)
##  num [1:128, 1:128, 1:100] 463 466 466 499 485 ...
```

You must supply the image x-y dimensions for this conversion.

2.2 Include Files

The *.incl file associated with each data cube contains pertinent information about pixel labels and masking (that is, whether each pixel is "masked" from the training / recall process). We import this information into R with the function `read_incl`, which takes a path to a .incl file, along with the image x-y dimensions as inputs:

```
incl = NeuroScopeIO::read_incl(filename = include_file, img_x = img_x, img_y = img_y)
str(incl)
```

```
## List of 2
##  $ masked: logi [1:128, 1:128] FALSE FALSE FALSE FALSE FALSE FALSE ...
##  $ class : chr [1:128, 1:128] "A" "A" "A" "A" ...
```

The return value of `read_incl` is an R list with components:

- **masked**, which is a matrix (nrows = img_y, ncols = img_x) whose (i,j) entry = TRUE if pixel (i,j) is maksed from processing, or FALSE otherwise
- **class**, which is a matrix (nrows = img_y, ncols = img_x) whose (i,j) entry = the class label of pixel (i,j). If no class labels are found in the .incl file, all entries of this matrix are set = "?".
  If an application requires matching a label or mask designation to each row of the data matrix X, we can convert the incl matrices (which are just cubes which have a 3rd dimension size = 1) to `convert_datcub_to_datmat`:

```
vec_mask = NeuroScopeIO::convert_datcub_to_datmat(datcub = incl$masked)
str(vec_mask)
##  logi [1:16384, 1] FALSE FALSE FALSE FALSE FALSE FALSE ...
vec_class = NeuroScopeIO::convert_datcub_to_datmat(datcub = incl$class)
str(vec_class)
##  chr [1:16384, 1] "A" "A" "A" "A" "A" "A" "A" "A" "A" "A" "A" "A" "A" "A" ...
```

Note: Since R follows column-major order, DO NOT just vectorize the matrices (i.e., with `c(incl$class)`). This would result in a vector of labels in pixel order (x=1,y=1), (x=1,y=2) (i.e, along rows). You could achieve correct behavior with `c(t(incl$class))`, but I suggest using the built-in conversion functions for clarity.

2.3 nna Files

The scaling information (from network input range to network output range) utilized during training and recall is contained in .nna files, which we load into R with

```
nna = NeuroScopeIO::read_nna(filename = nna_file, img_z = img_z)
str(nna)
## List of 4
##  $ input_min : num [1:100] -200 -200 -200 -200 -200 -200 -200 -200 -200 -200 ...
##  $ input_max : num [1:100] 1400 1400 1400 1400 1400 1400 1400 1400 1400 1400 ...
##  $ output_min: num 0
##  $ output_max: num 1
```

The return value of `read_nna` is again a list with components:

- **input_min** the min of the network input range, by dimension (a vector of length=img_z)

- **input__max** the max of the network input range, by dimension (a vector of length=img_z)
- **output__min** the min of the network output range (a single number, usually = 0)
- **output__max** the max of the network output range (a single number, usually = 1)

Importing .nna information is required for replicating a recall (performing BMU selection, building a CADJ matrix, etc) in R.

3 ann-SOMconsc Products

3.1 Weight Cubes

Since weight cubes (*.wgtcub files) are just Khoros cubes in viff format, we can import them with the same function used to import the data cubes:

```
W = NeuroScopeIO::read_khoros_viff(filename = wgtcub_file, verbose = F)
str(W)
##  num [1:20, 1:20, 1:100] -0.404 -0.403 -0.406 -0.411 -0.33 ...
```

Note that, as imported, W is in internal network range, so if any comparison to the data is desired it must be mapped to data range first. The weight cube has dimensions width = som__x and height = som__y, which we can save for later use:

```
som_x = ncol(W)
som_x
## [1] 20
som_y = nrow(W)
som_y
## [1] 20
```

Ultimately we need to convert the imported array W to a data matrix as well, but there is a crucial intermediate step. The (1,1) entry of W corresponds to the prototype vector attached to the top-left neuron of the SOM lattice. However, NeuroScope convention for other SOM products (nunr, CADJ) begins neuron/prototype ordering at the **bottom left** corner of the lattice. Thus, we need to flip the rows of this cube before converting to a data matrix using the function:

```
W = NeuroScopeIO::flip_datcub(datcub = W, flipX = F, flipY = T, flipZ = F)
```

Note the arguments flipX, flipY, and flipZ all default to F (to flip rows of a data cube, we set flipY = TRUE). Now we can convert W to a data matrix as above:

```
W = NeuroScopeIO::convert_datcub_to_datmat(datcub = W)
str(W)
##  num [1:400, 1:100] 0.098 0.0974 0.0982 0.098 0.1027 ...
```

3.2 NUNR Mapping Info

The actual mapping formed during SOM learning is contained in the associated .nunr file, which we load into R via

```
nunr = NeuroScopeIO::read_nunr(filename = nunr_file, som_x = som_x, som_y = som_y, img_x = i
str(nunr)
## List of 4
##  $ density    : num [1:400] 86 77 78 72 83 68 66 71 69 63 ...
##  $ class      : chr [1:400] "B" "B" "B" "B" ...
##  $ mapping    : int [1:128, 1:128] 272 334 273 292 247 230 356 248 337 345 ...
##  $ som_indexmap: int [1:20, 1:20] 381 361 341 321 301 281 261 241 221 201 ...
```

Note that the function `read_nunr` requires not only the path of the nunr file itself, but also the dimensions of the image cube and SOM. This information is contained in the header of .nunr files, but I have required it here as a check. Again, the nunr information is returned to R as a list with components:

- **density** a vector (length = # of PEs, or som_x*som_y) containing the size of each prototype's receptive field
- **class** a vector (length = # of PEs) containing the plurality data label of pixels mapped to each prototype
- **mapping** a matrix (nrows = img_y, ncols = img_x) whose (i,j) entry contains the index of the PEto which pixel (i,j) is mapped. For this purpose, PE indices are treated as linear indices (not array indices), and range from 1 to som_x*som_y.

- **som_indexmap** a matrix(nrows = som_y, ncols = som_x) whose (i,j) entry contains the linear index of the (i,j) neuron on the SOM lattice. The information in this matrix can be merged with that in the mapping matrix to determing the (x,y) neuron location to which a data vector is mapped, if desired.

3.3 Color Tables

Color table information is contained in .ctab files, which can be loaded with

```
ctab = NeuroScopeIO::read_ctab(filename = ctab_file)
str(ctab)
## 'data.frame':    20 obs. of  4 variables:
##  $ class: chr  "A" "B" "C" "D" ...
##  $ R    : num  15 255 76 237 182 255 255 23 136 140 ...
##  $ G    : num  82 116 187 41 96 242 111 190 128 86 ...
##  $ B    : num  186 23 23 57 205 0 255 207 123 75 ...
```

The color table is returned to R as a data frame with columns:

- **class** the set of distinct labels for pixels in the data cube
- **R**, **G**, **B** which are the RGB color components associated with each label.

Note that no assumption is made to the completeness of the ctab (i.e., whether it is missing color specifications for labels found in incl$class). The above function merely reads the information as it exists in a .ctab file.

3.4 CADJ

CADJ matrices are stored in .cadj files which are Khoros KDF formatted. **For use in R, .cadj files must be converted to viff format** with kformats. Once this conversion is complete, we can read the CADJ with the viff reader:

```
CADJ = NeuroScopeIO::read_khoros_viff(filename = cadj_file, verbose = F)
str(CADJ)
##  int [1:400, 1:400, 1] 0 34 0 0 0 0 0 1 0 0 ...
```

Note that the above imported CADJ as a 3d cube, whose 3rd dimension has size = 1. It is generally easier to work with matrices in R, so we can drop the third dimension

```
CADJ = CADJ[,,1]
```

Since the CONN matrix is also of frequent use, I suggest building it from the CADJ import immediately:

```
CONN = CADJ + t(CADJ)
```

4 Importing a Complete Product Suite

The above examples demonstrate import of NeuroScope products individually, but often we would like the entire product suite brought into an R environment. This can be accomplished with the helper function:

```
SHGRSOM = NeuroScopeIO::load_SOM_products(data_file = data_file, include_file = include_file
## Loading NeuroScope products ...
## ++ data cube ... done
## ++ include file ... done
## ++ nna file ... done
## ++ weight cube ... done
## ++ weight cube scaling ... done
## ++ nunr file ... done
## ++ lattice coords ... done
## ++ ctab file ... done
## ++ cadj file ... done
str(SHGRSOM)
## List of 14
##  $ X             : num [1:16384, 1:100] 463 489 490 467 470 ...
##  $ img_x         : int 128
##  $ img_y         : int 128
##  $ img_z         : int 100
##  $ incl          :List of 2
##   ..$ masked: logi [1:128, 1:128] FALSE FALSE FALSE FALSE FALSE FALSE ...
##   ..$ class : chr [1:128, 1:128] "A" "A" "A" "A" ...
##  $ nna           :List of 4
##   ..$ input_min : num [1:100] -200 -200 -200 -200 -200 -200 -200 -200 -200 -200 ...
##   ..$ input_max : num [1:100] 1400 1400 1400 1400 1400 1400 1400 1400 1400 1400 ...
##   ..$ output_min: num 0
##   ..$ output_max: num 1
##  $ W             : num [1:400, 1:100] -43.2 -44.1 -42.8 -43.2 -35.7 ...
##  $ som_x         : int 20
##  $ som_y         : int 20
##  $ nunr          :List of 4
##   ..$ density     : num [1:400] 86 77 78 72 83 68 66 71 69 63 ...
##   ..$ class       : chr [1:400] "B" "B" "B" "B" ...
##   ..$ mapping     : int [1:128, 1:128] 272 334 273 292 247 230 356 248 337 345 ...
##   ..$ som_indexmap: int [1:20, 1:20] 381 361 341 321 301 281 261 241 221 201 ...
##  $ lattice_coords: int [1:400, 1:2] 1 2 3 4 5 6 7 8 9 10 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:2] "x" "y"
##  $ ctab          :'data.frame':  20 obs. of  4 variables:
##   ..$ class: chr [1:20] "A" "B" "C" "D" ...
##   ..$ R    : num [1:20] 15 255 76 237 182 255 255 23 136 140 ...
##   ..$ G    : num [1:20] 82 116 187 41 96 242 111 190 128 86 ...
##   ..$ B    : num [1:20] 186 23 23 57 205 0 255 207 123 75 ...
##  $ CADJ          : int [1:400, 1:400] 0 34 0 0 0 0 0 1 0 0 ...
##  $ CONN          : int [1:400, 1:400] 0 72 0 0 0 0 0 2 0 0 ...
```

The function `load_SOM_products` is just a wrapper which calls each step of the above under the hood. Its return value is an R list with component names as above. The conversion of both the data and weight cubes to their respective matrices is done internally (along with the row reordering of W discussed above). List elements can be accessed via the `$`, e.g., to return the data matrix

```
str(SHGRSOM$X)
##  num [1:16384, 1:100] 463 489 490 467 470 ...
```

One caveat here is that, if nna_file is given, the prototype matrix is mapped from network range to input range.

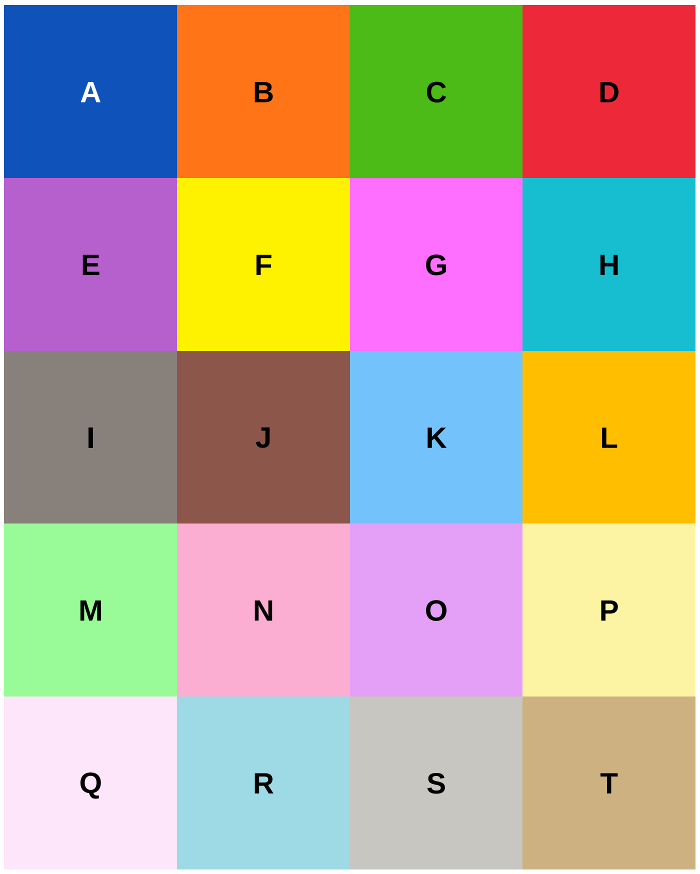
Note that not all inputs to `load_SOM_products` are required. If, for example, you only wish to import the data cube, you can specify the data_file path and leave the others blank

```
str(NeuroScopeIO::load_SOM_products(data_file = data_file))
## Loading NeuroScope products ...
## ++ data cube ... done
## ++ include file ... skipped (given as null)
## ++ nna file ... skipped (given as null)
## ++ weight cube ... skipped (given as null)
## ++ nunr file ... skipped (given as null)
## ++ lattice coords ... skipped (must supply wgtcub_file to compute lattice coords)
## ++ ctab file ... skipped (given as null)
## ++ cadj file ... skipped (given as null)
## List of 4
##  $ X    : num [1:16384, 1:100] 463 489 490 467 470 ...
##  $ img_x: int 128
##  $ img_y: int 128
##  $ img_z: int 100
```

5 Visualizations

Minor visualization capabilities exist within the package, mostly for checking the integrity of the imported information. For example, once imported, a color table can be viewed:

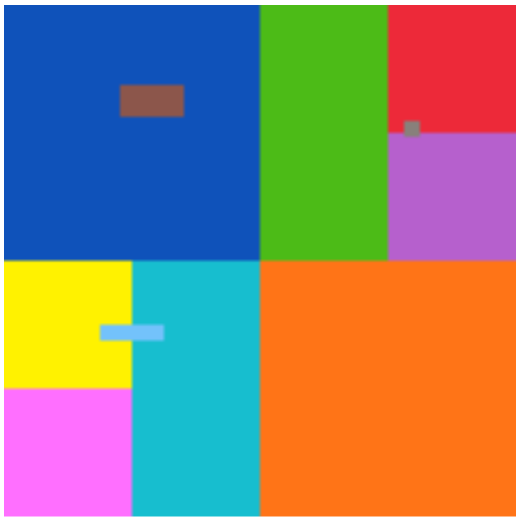
```
NeuroScopeIO::vis_ctab(ctab = ctab, label.cex = 0.9, label.font = 2, nrows_in_display = NULL
```



The label.cex argument controls the size of each label as plotted in its square, and the label.font controls whether the label is printed in bold (=2) or regular face (=1). nrows_in_display allows manually setting the number of rows in the visualized table. If not given, this defaults to NULL, which is a flag for the function to pick its own visualized table dimensions.

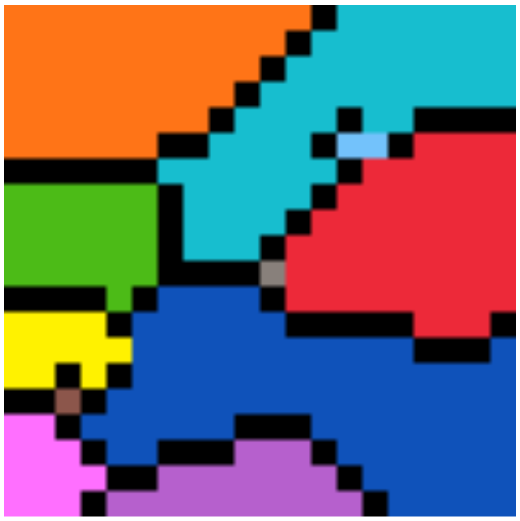
For labeled data, class maps can be visualized with

```
NeuroScopeIO::vis_classmap(class_matrix = incl$class, ctab = ctab)
## Warning in as.cimg.array(RGBarray): Assuming third dimension corresponds to
## colour
```

 The argument class_matrix must contain class labels for each pixel (and be in “image” format, i.e., with dimensions = image dimension). This obviously requires import of the include and ctab information.

PE plurality labels of the SOM can be visualized in the same manner by converting the nunr class information back to a cube:

```
som_labels = NeuroScopeIO::convert_datmat_to_datcub(datmat=nunr$class, img_x = som_x, img_y
NeuroScopeIO::vis_classmap(class_matrix = som_labels, ctab = ctab, pixel_expansion_factor =
## Warning in as.cimg.array(RGBarray): Assuming third dimension corresponds to
## colour
```



The additional argument pixel_expansion_factor enlarges each pixel defined in the class_matrix by its value. This is helpful for visualizing SOM labels, as

SOM lattices are typically small (and the resulting plot hard to see).