CSCI 8360 Data Science Practicum
Department of Computer Science
University of Georgia

Project 3: Neuron Finding

DUE: Thursday, March 8 by 11:59:59pm

Out February 15, 2018

1 Overview

As they say in Monty Python–and now for something completely different!

Hopefully by now, you've got the handle of the fundamentals: programming in distributed environments, implementing scalable analysis pipelines, and building preprocessors and classifiers that work on unstructured text. You're going to need these skills for the latest challenge: finding neurons in very large, time-series image datasets. Here are the details:

http://neurofinder.codeneuro.org/

This is the trickiest project thus far, at least in terms of what you're trying to learn. This can best be described as object-finding or image segmentation, where your goal is to design a model whose output is the coordinates to regions of interest in an image. There's no discrete label; rather, your model needs to learn segments in a continuous two-dimensional plane; relevant information to learning these segments, however, may be strewn over a third dimension of time. This makes for a very high-dimensional, large-scale problem: the data are height-by-width-by-time, and your model needs to learn a height-by-width mapping of pixels, where each pixel is either part of a neuron, or isn't.

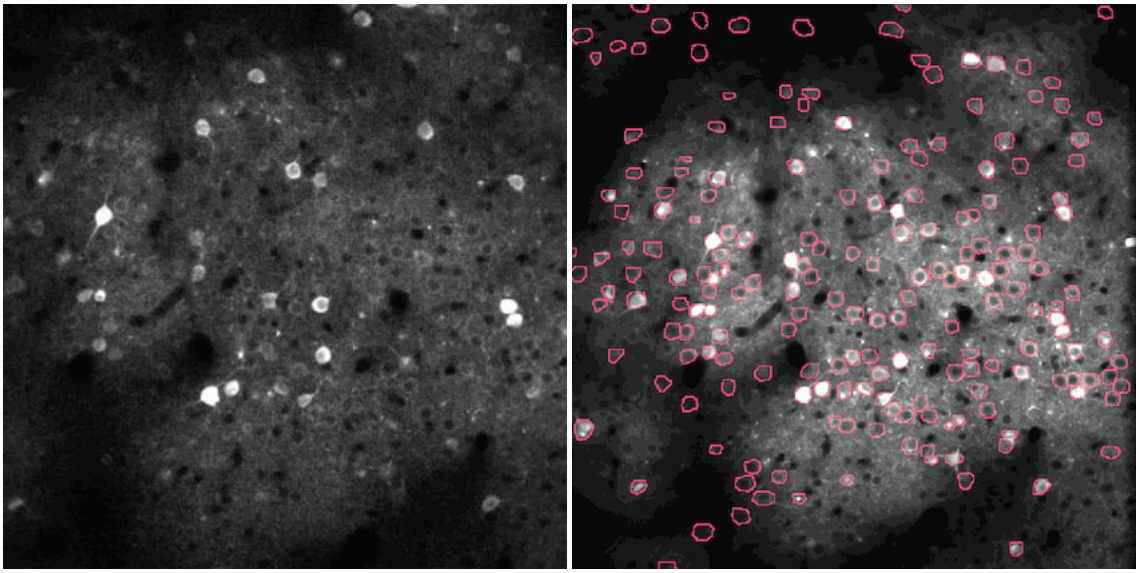
Each folder of training and testing images is a single plane, and the images are numbered according to their temporal ordering. The neurons in the images will "flicker" on and off, as calcium (Ca^{2+}) is added, activating the action potential gates. You'll have to use this information in order to locate the neurons and segment them out from the surrounding

image.

2 Data

Each file is a TIFF image, separated into folders, where each folder is a single sample. There are 19 training samples, and 9 testing samples.

Each folder contains a variable number of images; sample 00.00 contains 3,024 images, while sample 00.01 contains 3,048. The image files themselves are numbered, e.g. `image00000.tiff`, but all the images in a single folder represent the same sample, just taken at different times with different calcium levels. The training labels exist at the sample level, so you'll use *all the images in a single folder* to learn the locations of the neurons. Each folder will have a unique sample with unique numbers and positions of neurons. However, while time is a dimension to the data, you may not need to explicitly model it; you're just interested in finding the active neurons in space.



The image on the left represents more or less what you'll receive in the training and testing data. The image on the right is the goal of your learner: draw circles around the regions that contain neurons. The data are accessible at the CodeNeuro website linked above.

In addition to the images themselves, the training sets also contain ground-truth regions of interest (ROI) coordinates in the `regions` subfolder, formatted in a JSON file. Each ROI is represented using only two fields: an integer `"id"`, and an $N \times 2$ array of pixel coordinates named `"coordinates"`. This array lists the (i, j) coordinates in the image of the contour entirely surrounding the neuron (i.e. the pink borders in the right-hand

image above).

Your goal is to develop an image segmentation pipeline that identifies as many of the neurons present as possible, as accurately as possible. The ground-truth labels for the 9 test samples are stored in AutoLab and will be used to grade your submissions.

Like on the CodeNeuro website, we're using more than simple classification accuracy this time around to rank submissions. For the 9 testing sets, you'll get a **combined score** from precision and recall, which is posted on AutoLab for each of the 9 test sets. Each metric measures something slightly different:

1. **Recall**: Number of matched regions divided by the number of ground-truth regions (i.e., ratio of your correct predictions to the number of actual neurons)
2. **Precision**: Number of matched regions divided by the number of *your* regions (i.e., ratio of your correct predictions to the total number of neurons you predicted)
3. **Combined**: $2 * (\texttt{recall} * \texttt{precision})/(\texttt{recall} + \texttt{precision})$
4. **Inclusion**: Number of intersecting pixels divided by the number of total pixels in the ground-truth regions *(not posted on AutoLab leaderboard)*
5. **Exclusion**: Number of intersecting pixels divided by the number of total pixels in *your* regions *(not posted on AutoLab leaderboard)*

Due to the relatively limited nature of AutoLab's leaderboard compared to that on CodeNeuro, only the combined score for each test set will be posted following a submission. However, if you'd like a more detailed breakdown including all five metrics mentioned here, let me know; if there's a critical mass requesting this, I'll see what I can do (e.g. install something to email you with the details following a submission to AutoLab).

3 Theory Guidelines

We're solidly in imaging territory, which means Convolutional Neural Networks (CNNs) are the Big Thing$^{\text{TM}}$. However, before you reach for a deep learning framework and start piecing together some absurd 123-layer densely-connected network, I'd like to point you to a slightly less-shiny but still robust and reliable method: non-negative matrix factorization, or NMF.

NMF is in the same family of algorithms as Principal Components Analysis, or PCA; it's a dimensionality reduction algorithm by way of matrix factorization. It attempts to take some matrix, X, and factorize it into constituent basis vectors (e.g., principal components) and factors (e.g., eigenvalues), subject to some constraints. In PCA, those

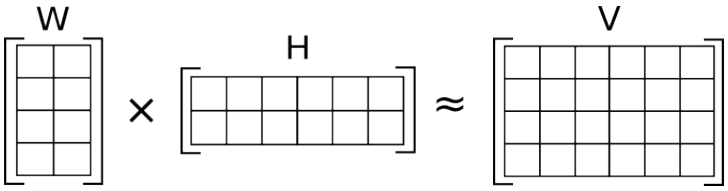


Figure 3.1: From Wikipedia's NMF page. In this example, V is the data matrix, and W and H are the basis vectors and factors, respectively.

constraints maximize "explained variance" in the data. In NMF, the goal is still to minimize reconstruction error (as PCA also does), but instead of explicitly also maximizing explained variance, NMF instead requires the elements of its factor matrices to be *non-negative* (hence, the name).

If $X \in \mathbb{R}^{n \times m}$ is our data, then the goal of NMF is find W and H such that

$$\min_{W,H} ||X - WH||_F,$$

subject to $W \geq 0$, $H \geq 0$. The dimensionality reduction comes into play with the choice of p, the number of basis dimensions: $W \in \mathbb{R}^{n \times p}, H \in \mathbb{R}^{p \times m}$, where $p <<< n, m$.

The non-negative constraint makes a lot of sense for applications where you specifically want "weights" on the dimensions of your basis vectors; in fact, you can directly interpret this as clustering.

Which brings us to why this method is useful for image segmentation: like most dimensionality reduction strategies, it can also be interpreted as a feature extractor, learning features that are somehow "important" (*a la* the objective function); however, NMF specifically is useful here because it can essentially perform unsupervised identification of features of interest in your image, i.e., pixels that change (blink) over time. Here is a brief example of NMF in action on this very dataset.

NMF does a great job picking up non-zero fluctuations in the data from the background. Problem is, it's pretty terrible at picking out neurons that *don't* light up, or which are otherwise closer to the background. Note on Neurofinder's own leaderboard that the "local-nmf" implementation–the one linked in the previous paragraph–is only about halfway up the list. This is to highlight that **NMF alone won't be enough**. At the very least, you'll need to increase its sensitivity to pull out the darker neurons.

4 Engineering Guidelines

Code reviews by teammates are now REQUIRED. This means everyone is **REQUIRED** to lock your `master` branch against direct pushes; instead of pushing directly

to `master`, everyone will now create their own branches for any modifications, commit them, submit a pull request, and have their code reviewed by at least one other teammate before it can be merged into the `master` branch. You can set this up in two steps: first, protect the master branch; second, enable required reviews for pull requests (go through Step 5, and then Step 9-10).

The gloves are off; anything is fair game. Use whatever framework you want, distributed or not. If you're not sure where to start, I would highly recommend looking at the CodeNeuro website; existing submissions' code repositories are linked from the website. You are more than welcome to use these approaches as *guidelines*; just know that I will be **strictly enforcing Academic Honesty policies**. If I find code that has been substantially copied from others, or ideas used without attribution, there will be problems.

Create a GitHub repository for you and your team to work on the project. You can make it private, and you can create a GitHub team within the DSP-UGA org, but the development must take place under the GitHub organization as I will be using it and the activity you record on it as part of the grading process. Name the repository with the scheme of `team-name` to help differentiate from other teams and future projects. **Seriously, please name it correctly**.

Exercise good software design principles. This means: adopt a consistent coding style, document your code, use the GitHub ticketing system to identify milestones and flag bugs, and provide informative and frequent `git commit` comments. Also, be sure you include an informative `README`, an accurate `CONTRIBUTORS`, a `LICENSE`, and are organizing your code in a clean and logical way.

Shut down your GCP clusters when you are finished testing. GCP clusters incur costs for every hour they are turned on, even if they are not actively running any jobs. That said, it?s also expensive to spin them back up if you?re running a lot of jobs, so I would encourage you to turn them on, run a bunch of tests, and only when you?re out of tests to run, you shut them down until you can get more in the pipeline.

5 Hints and Reminders

These datasets are *huge*; almost half a terabyte in total. In Project 1 you could kinda fudge your program using operations that weren't terribly scalable. That was by design, since you were probably still getting the hang of Spark. Here, though, those strategies will no longer work: you and your team need to determine a design that *scales*. Ideally your Project 1 codebase will be a good starting point, but if it's not, *you need to get to work immediately*.

Read over the CodeNeuro website! It has lots of great tips and pointers for how

to get started, including a "random algorithm" submission and a "terrible algorithm" submission that serve as useful templates for how to create and format the output of your model. All the associated GitHub repositories are linked from this website. http://neurofinder.codeneuro.org/

If you run into problems, work with your teammates. Open up GitHub tickets, discuss the problems, find workarounds. If you still run into problems, query the Slack channel. I'll be regularly checking Slack and interjecting where I can. Whatever you do, please don't turn into Lone Riders and try to solve the whole thing yourself.

Last year's top scorer obtained a 3.29359 combined score over the test sets–an average precision of 0.93366, average recall of 0.99291, average inclusion of 0.60408, and average exclusion of 0.76294. By comparison, the lowest combined score was 2.89987. Not a big margin of difference! You'll have to use every ounce of planning and strategy to obtain higher scores!

6 Submissions

All submissions will go to **AutoLab**, our brand-new autograder and leaderboard system.

One teammate should be designated for submitting on behalf of the team; they should submit to Autolab *using the team name* so I can easily match submissions with teams. If two people from the same team submit, I will pick one at random and grade that submission, so be sure you're communicating clearly.

When submitting, you will submit one file: **the JSON file containing the predictions of your trained model on the test datasets.** The format of the submitted JSON file is **a list of dictionaries**. Each dictionary has all the regions for a single dataset (e.g. 00.00, 04.00, etc), so you'll append a list of these together if you're submitting results for multiple datasets at once. *You don't have to submit for every dataset every time*; if a dataset dictionary is missing in your submission, the autograder can handle that gracefully. However, the overall structure must remain the same: even if you're submitting results for just 1 dataset, you should still have the outer structure be a list of 1 element (the lone dictionary for that one dataset).

For example, to submit two predicted regions for one dataset, the JSON file would look something like this (square brackets denote lists or arrays, and curly braces denote dictionaries or hashmaps or associative arrays):

```
[
    {"dataset": "00.01.test",
    "regions":
        [
            {"coordinates": [ [0, 0], [0, 1] ]},
```

```
            {"coordinates": [ [10, 12], [14, 17] ]}
        ]
    }
]
```

Let's parse this structure:

- The outermost data structure is a **list**. This list will contain up to 9 elements, one for each of the test datasets, but it can contain as few as 1 element for submitting on only a single test dataset.
- A single element of this list is an **associative array**, or dictionary, or hashmap. It should have two keys: `dataset` and `regions`. These keys and corresponding values are required.
- The value for the `dataset` key is just another string: the name of the test dataset (e.g. `00.00.test`, or `04.00.test`, and so on). The value for `regions`, however, is yet another **list**.
- Each element of the list is an **associative array** / dictionary / hashmap. These maps have 1 required key: `coordinates`.
- The value for `coordinates` is a **list of (x, y) integer pairs**. These pairs identify the (x, y) coordinates of each pixel in the dataset that corresponds to a single cohesive region.
- Multiple distinct regions in the same dataset, therefore, are denoted through multiple dictionaries in the `regions` list, each with the `coordinates` key and corresponding list of integer pairs.

If this is still not clear, **please read the CodeNeuro website.** There are numerous examples and excellent clarifying documentation for you, including this example of how to create the JSON submission file.

Your classification accuracy will be tabulated and compared against the true labels (hidden on the server) and should appear on the leaderboard.

If you think you can do better than what shows up–particularly to beat out everyone else–make another submission! There's no penalty for additional submissions, but keep in mind that swamping the server at the last minute may result in your submission being missed; AutoLab is programmed to close submissions *promptly* at 11:59pm on March 8, so give yourself plenty of time!

7 Grading

Given that this is a practicum, and that you're all talented graduate students, the grading for the projects is a little different, operating on a sliding scale with some stationary points that require extra effort to move beyond.

Everyone's starting grade is a **B** (85%). If you follow the guidelines in this handout, implementing the basic algorithm as described, and everything functions properly, and your documentation is reasonably ok, and there were no major issues with your team, then you also finish with a **B**. If you're happy with that, great! If not, read on:

Getting an **A** requires an investment of additional work, both in terms of functionality and algorithm performance, as well as on the software engineering side. You must demonstrate the extra customizations you made to your code and how those tweaks resulted in an improved score on the final testing set (getting the top score, for instance, would certainly qualify–though I feel I should point out that this sort of thing is relative, in that if you get a new top score that is only 0.000001% better than the previous one, it won't translate to an entire additional letter grade. #SorryNotSorry). If your pipeline is multi-staged, clearly showing that a lot of thought went into the theoretical design–perhaps even using a current method (that is *properly* and *prominently* cited)–it will help boost your grade.

Furthermore, you must show exemplary software engineering techniques: your code style should be clean, logical, and well-documented, and you should provide a clear README with instructions to new users on installing and deploying your document classifier. Your `git commit` comments and GitHub tickets should provide a "paper trail" of documentation to show the route your team took in developing the software. Unit testing, continuous integration tools, consistent and thorough use of branches and merges, and adherence to coding style standards (e.g. Python's PEP8) will also help push your grade above the default.

It should go without saying, but violations of standard coding practices or errors in the theory will drop your grade; you don't get a **B** for simply turning in an empty assignment.

This combination of **algorithmic novelty** and **engineering best-practices** will earn you and your team that well-deserved **A**.

8 Summary

In summary, these are the things I will look over when determining your team grade:

1. **Code**: Organization, structure, style, and clarity. Strategy and theory will also be a big component (what did you implement? was it implemented correctly? does the theory follow logically from the problem you're trying to solve?).

2. **Documentation**: This includes comments in the code, but also in the repository itself (GitHub wiki, `README` instructions). How easy would it be for someone to get up and running with your code, or to submit a bugfix?

3. **Accuracy**: Testing accuracy as submitted on AutoLab, and how far you are from the very top.

4. **Team**: Like the code, I'm looking for structure and organization, as well as a "paper trail" to get a feel for your team's working dynamics. How were project efforts divided among teammates? What do the git logs reveal about who was contributing to the project? How does this align with the `CONTRIBUTORS` content? Were code reviews conducted properly?

5. **Extras**: These include things like continuous integration tools, project websites, theoretical novelty (within the constraints of the project), unit tests with good coverage, among others. Use your imagination!