Bell Jar: A Semi-Automated Registration and Cell Counting Tool for Mouse Neurohistology Analysis

Alec Soronow

March 27, 2022

Abstract

In this paper, I introduce *Bell Jar*, a multi-platform tool that aids users in aligning their tissue to a reference atlas and can perform cell detection and quantification. The problem of aligning and registering reference atlases to neurohistology data has been a critical effort in contemporary neuroinformatics. The ability to accurately register experimental tissue to an atlas enables automated determination of experimental cells areas and layers for rapid quantification of experiments. Despite widespread efforts to solve this problem, most available tools are paid products or complicated compilations of scripts tuned for only one kind of data. *Bell Jar* provides these tools in one easy-to-navigate application that manages its dependencies, supports various data types, and requires no programming expertise.

1 Introduction

An essential advancement in neuroscience research has been the widespread use and development of intersectional viral tracing in transgenic animals. These experiments allow researchers to precisely target substructures of the brain to help better understand their function and connectivity. While some labs have access to precise imaging methods like two-photon tomography, most brains from these experiments are processed as coronal sections of tissue. The varying section angle of this tissue and its low spatial resolution means identifying labeled cells in experimental brain tissue is challenging with this histology. To assign anatomical areas to the labeled cells in these sections, an individual using a reference atlas [6, 9] (a mapping of the 3D locations of the brain onto 2D sections and agreed upon as the standard) will make manual determinations of area boundaries. Delineating areas in a coronal section by hand is highly subjective, with varying biases depending on the person making the determinations. Ensuring the accuracy of these assignments is crucial to the validity of the conclusions drawn from the experiments.

Some groups have developed suites of tools to automate parts or all of the quantification process to improve accuracy and validity [10, 11]. These methods have shown promising results, however, they have caveats that dissuade widespread use in the community. The first barrier to adoption is the required technical experience to use these methods. While the creators have provided excellent interfaces for their tools, the lack of one centralized application that is cross-platform and dependency agnostic leads to issues for users who do not have the technical knowledge to troubleshoot the platform-specific problems. Without a centralized application, the collection of scripts can become a challenge to manage, and the process may feel clunky to users that expect straightforward input/output interfaces. The second barrier is that these tools are either strictly automated or manual. There is much required preprocessing and fine-tuning for each experiment in the automated case, causing issues for labs with various experiments with different tissue types and conditions. In

the manual case, while there are some inbuilt algorithms to assist the user, much of the work that can be automated is repeated by hand for all experiments.

To address these barriers and improve the state of the art, I present *Bell Jar*. This single cross-platform application combines robust machine learning automation and manual fine-tuning to provide a more user-friendly and accurate quantifying method for neurohistology. The application helps users align and quantify brains in a directed workflow that minimizes user input. In the first step, images of experimental tissue stained with either DAPI or Nissl are automatically preprocessed to isolate just the section with no background objects or debris. The user confirms the selected tissue sections and proceeds or manually adjusts the filter. The next step uses a convolutional autoencoder to create embeddings of the experimental images. These are compared to precomputed embeddings of the Allen Brain Reference Atlas [9] over a selected range of sectioning angles to find the best angle for the experiment and the most likely matching atlas sections. The user then fine-tunes these suggestions for a high-fidelity alignment. Finally, the alignment is warped onto the experimental tissue. Users can either export this mapping to use with custom scripts or proceed with the built-in cell detection tool. The result of the pipeline is a precise and accurate mapping of atlas areas onto the experimental tissue.

2 Materials and Methods

2.1 Cross-Platform Application Architecture

One of the primary goals of *Bell Jar* is to be functional on any platform. To accomplish this, I built the program's user interface using electron, which I've diagramed the architecture of in Figure 1. The Electron framework is based on NodeJS and chromium, so it can run on any machine which supports those frameworks. Electron also allows bundling to native application packages on Windows, OSX, and Linux so a user can download the tool instead of building it from the source. As shown in Figure 1, I took a compartmentalized approach to the architecture. In image analysis and machine learning, two of the essential tools for this project, the best libraries for rapid prototyping and usability, are in python. While python scripts execute the app's functionality, the Electron front-end provides a clean, easy-to-understand GUI for users to interact through. So I found a node package that enables spawning python child processes with specific environmental targets.

So, I created a python virtual environment with Conda and included its strict dependency list with my app distribution. In this manner, I have accounted for the specific packages without needing the user to install each dependency independently. Additionally, this also means the only dependency the user must install for this application is Conda. The user points to the install in the setup phase, and the environment builds with known working version libraries. Another key feature of the architecture is the room for multithreading. While none of the currently included scripts use this, future scripts can use the output controller to concatenate data between spawned python processes.

2.2 Convolutional Autoencoder Image Similarity

Instead of a hand-crafted metric, I took a machine learning approach to determine the cut angle and the matching sections. A convolutional autoencoder transforms the input images into vector embeddings which I can compare to embeddings of the atlas sections to identify the best matching section. I iterate this process over the experimental tissue and take the majority cutting angle as the truth angle (the angle to search for proper matches). Then I check once more for matches but

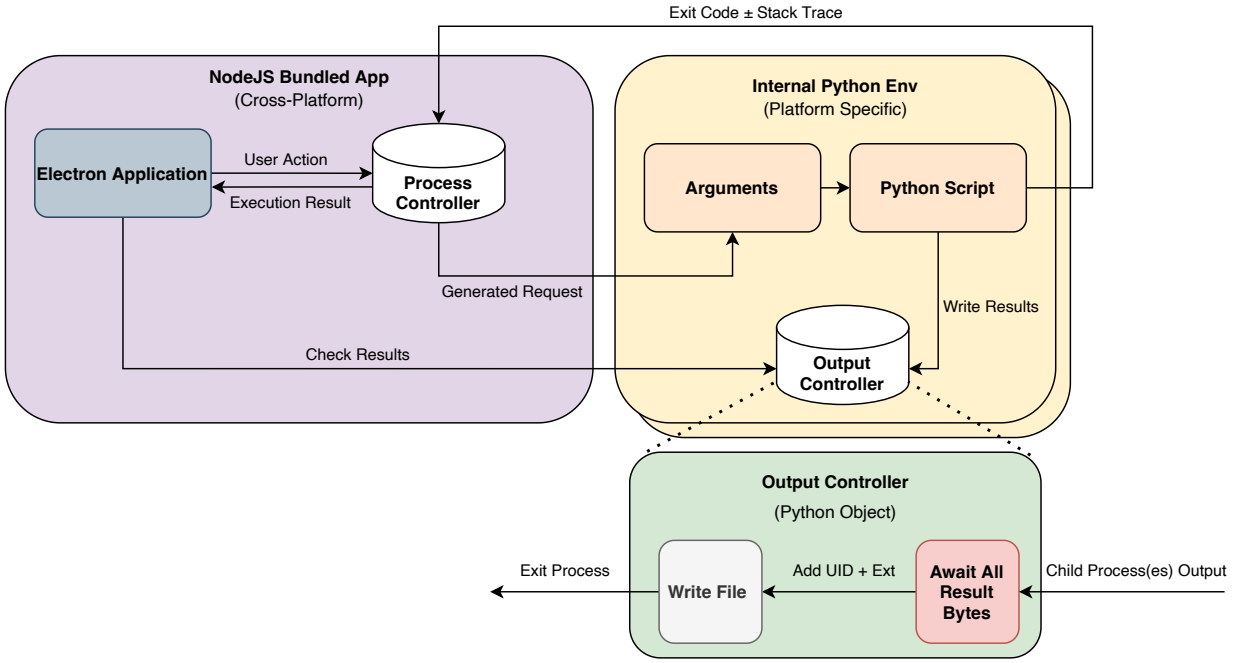

Figure 1: Drawn here is a map of *Bell Jar's* architecture. The purple box contains the processes within the electron framework, and the yellow box details the pathway within the python environment. When a user selects an action in the program (i.e., detect cells), the relevant directory paths and arguments are first collected. These I pass as arguments in a generated request to the python process running the script for the selected task; the request contains additional information such as output directory, a timestamp, and a unique id (to accommodate multiple actions at once on capable machines). When the script finishes, it writes the output to its output controller, or for multithreading, a single output controller can concatenate data between script children. Figure drawn with *Diagrams.net* [5].

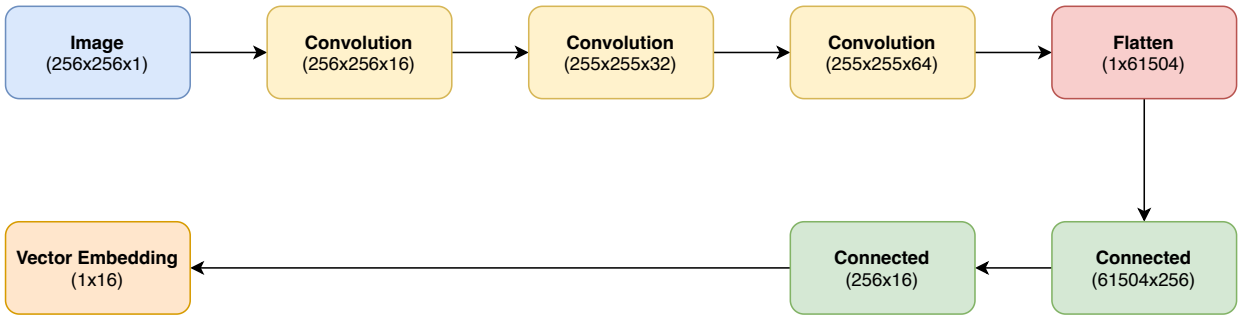

Figure 2: Drawn here is the architecture of the convolutional autoencoder. It takes a 256x256 grayscale image (channel dimension of 1) and convolves it with a stride of (2x2), a kernel of (3x3), and a padding of 1 over three deepening layers. I then flatten the large tensor to a long vector that I pass to fully connected layers. These layers distill the feature information into a smaller embedding that represents the key features in the image. Figure drawn with *Diagrams.net* [5].

restrict to the true angle and preserve the best matches for each section. If the scores are below a threshold, then the best match is taken, and using the user-provided spacing, the remaining sections I extrapolate as follows. If the best match was the section j at $1000\mu m$ and the spacing is $50\mu m$, then the first and last k to j and j to n sections will be $\forall k \ldots j : (1000\mu m - 50\mu m \times j) + 50\mu m$ and $\forall j \ldots n : (1000\mu m + 50\mu m \times n - j) - 50\mu m$.

The architecture of the encoder is drawn in Figure 2 and is a simple but effective network that is small enough to train efficiently on a consumer PC [3]. I generated a synthetic dataset of approximately $30,000$ atlas images at various section angles by rotating and slicing the digital atlas volume to train the network and create the atlas embeddings. I made a decoder with the same architecture but revered to test and train the encoder.

2.3 Warping, Registration, and Cell Detection

The warping of the matching section onto the experimental section is done with SIFT in OpenCV [4, 8]. First, I calculate up to one million features from each image using the SIFT detector [4]. Next, the best one thousand features are selected based on the hamming distance between the two images' features. The homography of the best matches is found and the warp is applied to the atlas image and its annotation. Final resulting registration is now available to query for atlas locations. To get locations of cells in the tissue, I used YoloV5 [7]. I trained the object detector to recognize neurons in the experimental tissue, and then I took the centroids of the bounding boxes as my cell locations. However, the raw images of the cells are of too low quality to be used outright. So I first apply a maximum intensity projection to the images to remove as much blur and get the best brightness possible. Next, I apply a tophat filter to even the illumination and remove the beveled features of the neuronal cell bodies. Finally, the detector is trained on these processed neurons so that I can detect cells in the sections. Additionally, the final detection step uses SAHI (Slicing Aided Hyper Inference) [1, 2] in combination with the YoloV5 model to better detect the small neuronal cell bodies in the large images.

3 Results

While still not finished, the final pipeline has shown promising time savings over other methods I currently use or in combination with them. As seen in the diagram of our standard processing pipeline in Figure 3, the total time to completely quantify an experimental brain can be as little as one hour. An example of how this workflow is applied is demonstrated in Figure 4, where a single section is used to illustrate. The major time savings from the application so far has been the advances in cell detection using YoloV5, and SAHI [1, 2, 7]. Previous methods relied on image segmentation to process their cell images and obtain the locations of neurons. Common tools used in segmentation take significantly more processing power and memory than processing images in the *Bell Jar* pipeline. For example, to properly segment our images in the QUINT workflow [11] we downsampled our images 30% or more to fit in memory during segmentation. Downsampling images lead to a loss in precision of cell location and clumping of neighboring cells. With the object detection framework herein, these are not issues since inference is run on the original resolution images and does not rely on intensity thresholding to separate cells.

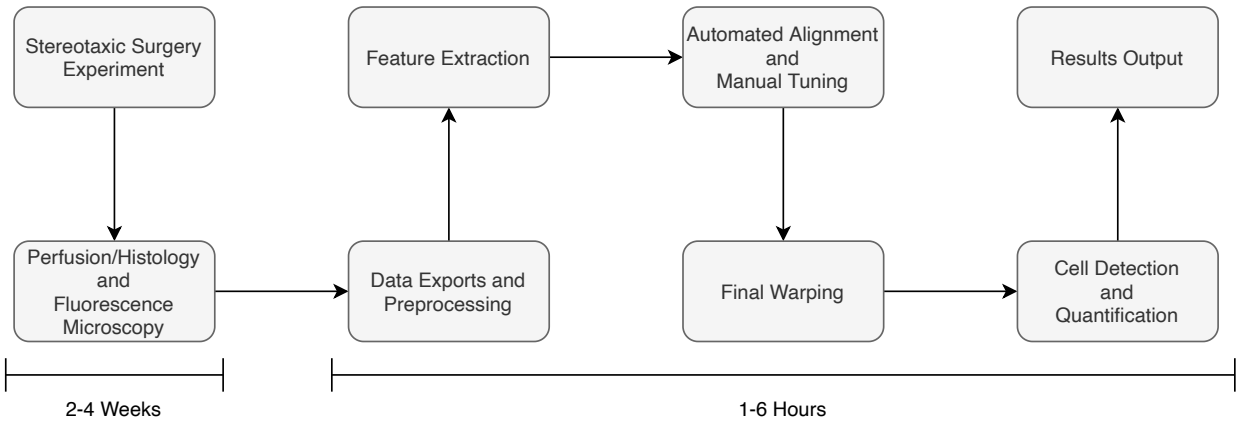

Figure 3: Drawn here is the typical workflow of an experiment. The initial surgery and imaging take weeks due to incubation times and scheduling. Therefore, it is crucial that the analysis be completed quickly but with integrity. To do this, I split the image data into each of its channels. The DAPI images (nuclear stain) I used to align and warp the experiment to the Allen Brain Atlas [9]. Afterward, the rabies images (neuronal cell bodies) are processed, and the cells are detected with a YoloV5 model using SAHI [1, 2, 7]. I transformed the coordinates of the detected cells into regions of the warped atlas for the finals results. Figure drawn with *Diagrams.net* [5].

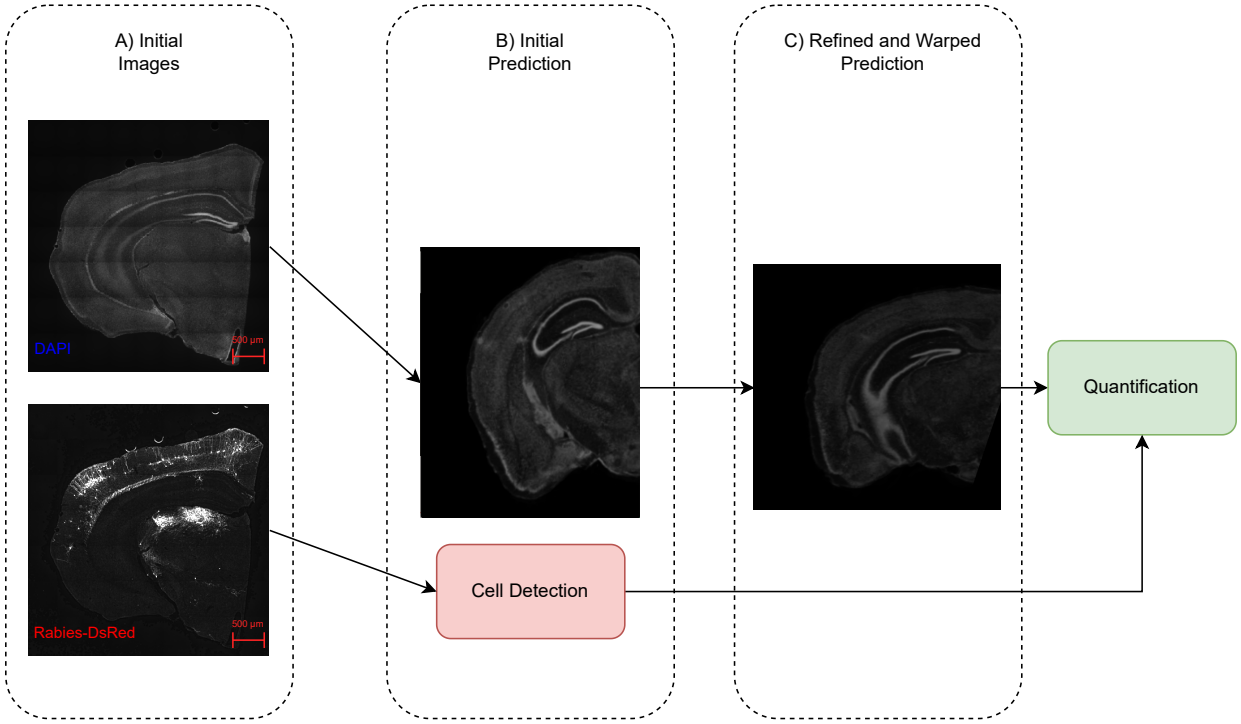

Figure 4: Shown here is the Bell Jar pipeline illustrated with a single section of mouse brain tissue. A) The multi-channel fluorescence microscopy images are separated into DAPI (4',6-diamidino-2-phenylindole) to label the cell nuclei and traced Rabies-dsRed+ cell bodies standalone images. The traced cell bodies have been further processed with tophat filtering to make them easier to detect. B)The encoded embedding of the DAPI image is compared to the mouse brain atlas embeddings, and I display the initial match. At this point, the traced cell body image is run through the YoloV5 [7] detection model with SAHI [1, 2] to obtain the locations of the cell bodies in the image as pixel coordinates. C) The suggested section I manually refined to better match the tissue and then warped to fit. Finally, the quantification can be done with the refined prediction and cell locations' inferred atlas locations. Figure draw with *Diagrams.net* [5].

4 Discussion

Bell Jar is still under heavy development. I have completed several key portions of it outlined in this paper. Notably, the cell detection methods and convolutional autoencoder network are operational. The front-end user experience has not seen much work since the start of the project and is the next focus for development after I can achieve reliable counts. In the meantime, I have used the cell detection methods to greatly speedup the QUINT workflow [11] with our large data sets yielding significant time savings already.

Future features of *Bell Jar* after the alpha development will be the ability to extend the program with a python scripting library. Scripting would allow labs to program their tools into the framework, which is vital since not everyone will need cell body detection and may want other methods. I have already made the code for this project open-source, and it is freely available at https://github.com/asoronow/belljar for anyone to contribute. There are no releases in the limited repo now, and it must be built on your machine. You must also locate the python dependencies, which are changing rapidly at this stage. I hope that with community effort, the program can be extended beyond mouse work to other animal brain atlases and help increase the speed of neuroscience research.

5 Acknowledgements

I want to Acknowledge Matt Jacobs and Richard Dickson for their numerous contributions of ideas for the program, which helped get it to the breadth I've approached today. I'd also like to acknowledge Euiseok Kim for advising me on this project and providing the encouragement to persevere. Additionally, I'd like to recognize the Koret Foundation for providing the funding for me to conduct this research with their scholarship. Lastly, I'd like to acknowledge Kevin Karplus, who provided the template for this report, and all his wisdom on tackling large technical projects.

References

[1] Fatih Cagatay Akyon, Sinan Onur Altinuc, and Alptekin Temizel. "Slicing Aided Hyper Inference and Fine-tuning for Small Object Detection." In: *arXiv preprint arXiv:2202.06934* (2022).

[2] Fatih Cagatay Akyon et al. *SAHI: A lightweight vision library for performing large scale object detection and instance segmentation*. Nov. 2021. DOI: 10.5281/zenodo.5718950. URL: https://doi.org/10.5281/zenodo.5718950.

[3] Eugenia Anello. *Convolutional Autoencoder in Pytorch on Mnist Dataset*. Jan. 2022. URL: https://medium.com/dataseries/convolutional-autoencoder-in-pytorch-on-mnist-dataset-d65145c132ac.

[4] G. Bradski. "The OpenCV Library." In: *Dr. Dobb's Journal of Software Tools* (2000).

[5] *Diagrams.net*. URL: https://www.diagrams.net (visited on 03/16/2022).

[6] Keith B. J. Franklin and George Paxinos. *The mouse brain in stereotaxic coordinates*. eng. Compact 3. ed. Amsterdam Heidelberg: Elsevier Academic Press, 2008.

[7] Glenn Jocher et al. *ultralytics/yolov5: v6.1 - TensorRT, TensorFlow Edge TPU and OpenVINO Export and Inference*. Version v6.1. Feb. 2022. DOI: 10.5281/zenodo.6222936. URL: https://doi.org/10.5281/zenodo.6222936.

[8] David G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints." en. In: *International Journal of Computer Vision* 60.2 (Nov. 2004), pp. 91–110. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000029664.99615.94. URL: http://link.springer.com/10.1023/B:VISI.0000029664.99615.94 (visited on 03/17/2022).

[9] Quanxin Wang et al. "The Allen Mouse Brain Common Coordinate Framework: A 3D Reference Atlas." en. In: *Cell* 181.4 (May 2020), 936–953.e20. ISSN: 00928674. DOI: 10.1016/j.cell.2020.04.007. URL: https://linkinghub.elsevier.com/retrieve/pii/S0092867420304025 (visited on 03/13/2022).

[10] Jing Xiong et al. "Mapping Histological Slice Sequences to the Allen Mouse Brain Atlas Without 3D Reconstruction." In: *Frontiers in Neuroinformatics* 12 (Dec. 2018), p. 93. ISSN: 1662-5196. DOI: 10.3389/fninf.2018.00093. URL: https://www.frontiersin.org/article/10.3389/fninf.2018.00093/full (visited on 03/13/2022).

[11] Sharon C. Yates et al. "QUINT: Workflow for Quantification and Spatial Analysis of Features in Histological Images From Rodent Brain." In: *Frontiers in Neuroinformatics* 13 (Dec. 2019), p. 75. ISSN: 1662-5196. DOI: 10.3389/fninf.2019.00075. URL: https://www.frontiersin.org/article/10.3389/fninf.2019.00075/full (visited on 03/17/2022).