Neuron Detection in Calcium Imaging Data

Guy Teichman & Moran Neuhof, Winter 2019

Neuron Detection

To detect the neurons properly, we tried several methods. In this short paper we will describe the main steps we took, and the main methods we used. For brevity, we will only describe the major advances and combination of methods which resulted in the biggest improvements.

Method #1 (Boolean mask):

The first method we tried was creating a Boolean mask from the Field of View video (FOV) which we summed over time, using a manually picked threshold. Since neurons that fire should be brighter than their background, thresholding can help us separate the neurons from the background. The main issue in this version appeared to be that many of the neurons are not highly visible in the FOV, and the high and inconsistent background fluorescence hinders our ability to successfully threshold out the neurons.

```
summed = np.float64(imgs.sum(axis=0))
msk = summed > (0.25 * np.max(summed))
```

Method #2 (Band pass filter):

To address the problem of the noisy background, we attempted to use the Fourier transform to filter our particularly high and low frequency data from the image (band pass filter). The goal of filtering high frequencies was to remove the high-frequency noise in the image; and the goal of filtering low frequencies was to remove the unequal background fluorescence, which appears to change in a consistent low frequency. After inverse-transforming the filtered image we once again created a mask from the image (see **Table 1** for scores). This method has significantly increased our precision while not severely hurting the recall. However, the filter has left behind a ringing effect which interfered with neuron detection.

```
def bandPassFilter(img, radIn=50, radOut=10000, plot=False):
    """Receive image, inner radius size, outer radius size.
    Return an image filtered with a disk, using FFT."""

    # FFT
    fft=np.fft.fftshift(np.fft.fft2(img))
    # shape
    x,y=np.shape(fft)
    xg, yg = np.ogrid[-x//2:x//2, -y//2:y//2]

    # define  filter disk
    inner_circle_pixels = xg**2 + yg**2 <= radIn^2
```

```
    outer_circle_pixels=xg**2 + yg**2 <= radOut^2
    filter_disk = np.ones_like(inner_circle_pixels)
    filter_disk[np.invert(outer_circle_pixels)] = 0
    filter_disk[inner_circle_pixels] = 0
    # filter
    fftfilt = fft * filter_disk

    ifft = np.fft.ifft2(fftfilt)
    img_filt = np.abs(ifft)
    return img_filt

f_summed = bandPassFilter(summed)
msk = f_summed > (0.1 * np.max(f_summed))  # create mask from the filtered image
```

Method	Recall	Precision	Combined	Inclusion	Exclusion
Boolean Mask	0.2061	0.203	0.2045	0.3338	0.8003
Band Pass filter	0.1455	0.5275	0.228	0.4588	0.855
Band Pass filter, watershed	0.1424	0.3264	0.1983	0.6608	0.7648
Background equalization, watershed	0.1424	0.3431	0.2013	0.6658	0.7753
Background eq, watershed, size selection	0.1333	0.557	0.2152	0.7087	0.759
Band Pass filter, watershed, size selection	0.1364	0.5056	0.2148	0.6983	0.7459
Smoothing, correlation, background eq., bandpass, watershed, size filter	0.2758	0.474	0.3487	0.723	0.6528
Smoothing, correlation, background eq, band pass, k-means, watershed, size filter, hit or miss	0.4121	0.3579	0.3831	0.6832	0.6227
Grid Search – best recall score	0.5545	0.2596	0.3536	0.6037	0.693
Grid Search – best combined score	0.4182	0.4182	0.4182	0.685	0.6465

Table 1: Neurofinder scores for the different methods

Method #3 (Watershed):

As our inclusion scores were rather low, we used the watershed algorithm to expand the size of the neurons detected. We used the Boolean masks produced by the various methods as seeds provided to the watershed algorithm. We then applied watershed to pipelines and increased our inclusion scores significantly (see **Table 1**).

```
def watershed(image,mask, filename, dims, dial_rad=9, max_neuron_size=21,
min_neuron_size=4, coef=2):
    """
    Receives an image,
    a boolean mask containing the seeds for the watershed algorithm,
    the json filename to save,
    the dimensions of the image
    and an optional dialation radius
    and max/min neuron sizes for size selection,
    as well as the length/width ratio (coef).
    """
    # sure foreground
    sure_fg = np.uint8(mask)
    sure_bg = np.uint8(morphology.binary_dilation(mask,morphology.disk(dial_rad)))

    # initialize
    img = np.zeros([512,512,3], dtype='uint8')
    img[:,:,0] = np.ndarray.astype(norm_data(image)*255,'uint8')

    # unknown area
    unknown = cv.subtract(sure_bg, sure_fg)

    # check connected components and label
    ret, markers = cv.connectedComponents(sure_fg)
    del ret
    # make backgorund 1, and unknown 0
    markers += 1
    markers[unknown==1] = 0
    # run watershed
    markers_after_WS = cv.watershed(img,markers)

    size_selected_markers = size_selection(markers_after_WS, max_neuron_size,
min_neuron_size, coef)
    if size_selected_markers is not None:  # if size selection kept anything
        mask_to_json(size_selected_markers, filename, True)
        return size_selected_markers
    else:
        return None
```

Method #4 (Background equalization):

Using the previous methods, we were unable to remove the image's background. The image background turned out to be dark in certain areas, while bright in others, thus preventing us from detecting neurons with varying levels of activity. We hypothesized that if we divide each pixel by the relative brightness of its area, the image's brightness would become more uniform. Therefore, we created a highly blurred version of the

summed FOV, by filtering it with a gaussian filter with a large kernel. We then divided the summed FOV by its blurred version and ended up with a similar image which has an equalized background brightness (see **Fig. 1**). We then used our previous pipeline (with watershed) to analyze this equalized image. Since this image's brightness was more uniform, thresholding was easier and more accurate. As the inclusion, exclusion and precision scores of this method increased, the recall score did not improve (see Table 1). While simple thresholding keeps a relatively higher recall score, it also results in a higher false positives rate (lower precision).

```
background = ndimage.gaussian_filter(summed, sigma=15)
summed_eq = summed / background
```

Method #5 (Size selection):

We then focused on the neuron morphology, which is resembles a round shape with a diameter of about 4 to 20 pixels. Given these morphological characteristics, we can iterate over the neuron-candidates we have identified and remove any neurons that appear to be too small (diameter under 4-5 pixels), too big (diameter over 19-24 pixels), or not round (the length/width ratio is higher than 2, or smaller than 0.5). When we applied this size selection, we observed that it allows us to use much more sensitive thresholding, so we can capture more true neurons, while not compromising our precision. In many cases this method allowed us to remove over two thirds of the neurons we detected, dramatically increasing our precision (see **Table 1**).

```
def size_selection(labeled, max_neuron_size=21, min_neuron_size=4, coef=2, verbose=False):
    """
    labeled: image with label of each neuron
    max_neuron_size: the maximal neuron size accepted
    min_neuron_size: the minimal neuron size accepted
    coef: the x/y or y/x maximal ratio
    verbose: True writes down the number of kept/removed neurons.
    """
    new_labeled = labeled.copy()
    n = np.max(new_labeled)  # number of neurons
    counter = 0
    for elem in range(1, n+1):  # iterate over neurons
        xelem, yelem = np.nonzero(new_labeled == elem)  # keep the positions of the element
        try:
            neuron_max_x_y = (np.max(xelem) - np.min(xelem), np.max(yelem) - np.min(yelem))
   # ( size x, size y)
            # if neuron is too big or too small
            if np.max(neuron_max_x_y) > max_neuron_size or np.min(neuron_max_x_y) <
min_neuron_size or is_not_round(neuron_max_x_y, coef):  # or the neuron shape is not round
                # TODO: add shape selection here (is_round)
                new_labeled[new_labeled == elem] = 0
                counter += 1
        except ValueError:
            continue

    kept = n - counter
    if verbose:
        print(f"removed {counter} neurons. Kept {kept} neurons.")
```

```
    new_labeled[new_labeled == -1] = 0

    if kept == 0 and verbose:
        print("No neurons kept! ignoring it.")
        return None
    else:
        return new_labeled


def is_not_round(neuron_max_x_y, coef=2):
    """Receives the length/width of the object, and returns True if the object is not
round.
    coef: the x/y or y/x maximal ratio."""
    x_len, y_len = neuron_max_x_y
    if (x_len/y_len) > coef or (y_len/x_len) > coef:
        return True
    else:
        return False
```

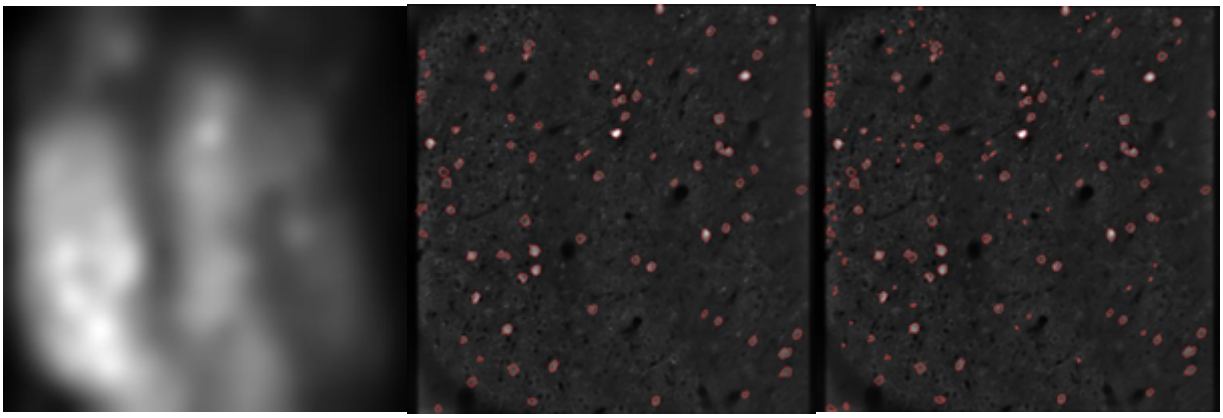


Figure 1: Neurons identified using various methods. From left to right: Background produced using gaussian filter; Summed image after watershed; Summed image after watershed on equalized background.

Method #6 (Filtering over time):

As our summed FOV image is a result of the change in neuron fluorescence over time. To remove noise in each pixel, we used 1d gaussian filter over the frames for each pixel. The filtering resulted in a cleaner, smoother signal, and less noise (**Fig. 2**). We integrated this smooth image in the methods discussed below.

```
def filter_timeseries(imgs, filter_methods):
    """Receive a 3d matrix, filter over time using a filter method."""

    # examples:
    # ndimage.gaussian_filter1d, 1
    # ndimage.gaussian_filter1d, 4
    # gaussian_laplace, 3
    # maximum_filter1d
```

```
    # initialize a zeros matrix for each of the filtering methods
    filtered_imgs = [[np.zeros(imgs.shape), filter_methods[i][0], filter_methods[i][1]] for
i in range(len(filter_methods))]
    print(f"Running {len(filter_methods)} timelapse filtering methods.")

    for method_num, (filter_method, param) in enumerate(filter_methods):  # run each of the
methods
        filtered_imgs[method_num][0] = filter_method(imgs, param, axis=0)

    return filtered_imgs  # [img, method, param]


# Creating a list of filters and their parameters
filter_methods = [(ndimage.gaussian_filter1d, 4)]
# Filtering each pixel's change in fluorescence over time
ts_filtered_imgs = filter_timeseries(np.float64(imgf), filter_methods)
```

Method #7 (Neighbor correlation):

As our recall scores dropped, we estimated that we had a problem in detecting correct seeds of the location of the neurons (later used by the watershed algorithm). Therefore, we tried generating a new "base image" to form the analysis on, instead of the summed FOV. We hypothesized that if we examine the change over time of each pixel, neighboring pixels that belong to the same neuron should “spike” together, and therefore their change over time will be correlated, while noise pixels will not be correlated neither with neuron pixels nor with other noise pixels. Hence, if we give each pixel a value based on its correlation with its neighboring pixels over time, we will end up giving high values to neuron pixels and low values to noise pixels (see **Fig. 2**). We also performed the background equalization described above for the result of the correlation (summed over the time dimension). Moreover, using this method we were able to detect neurons that were previously very difficult to detect using any of the other methods, including neurons that were not included in the ground truth (see **Table 1**).

```
mem_d_corrcoef = {}  # initialize
def corrcoef_val_per_couple(i, j, r, c, data, d=mem_d_corrcoef):
    """Calculate the correlation value per each two positions.
    Saves it to dict d for memoization"""
    if (i,j,r,c) in d:
        return d[(i,j,r,c)]
    else:
        val = np.corrcoef(data[:,i,j], data[:,r,c])[0,1]
        # keep one of the corr matrix
        d[(i,j,r,c)] = val  #
        return val


def correlate_neighbors(data):
    """Calculate correlation matrix between adjacent neurons over time."""
    ## 8 neighbors
    counter = 0  # intialize for progress
    neighborCorrcoef = np.zeros([512,512],dtype='float64')  # the matrix we write into
```

```
    n = 512*512  # total pixels
    for i in range(512):    # going over all the rows
        for j in range(512):    # going over all the columns
            cnt = 0  #how many neighbors did we successfully test (for regular pixel is 8,
for corner is 3, for edge is 5)
            newval = 0  # the sum of all of the correlations, that we will use at the value
of the specific pixel we test
            # build neighbor list
            neighbor_pairs = [(i+a,j+b) for a,b in ((-1, 0), (1, 0), (0, -1), (0, 1),
(1,1), (-1,1), (-1,-1), (1,-1))]

            # check neighbors
            for pair in neighbor_pairs:  #going over each neighbor's coordinate (row-
column) pair
                r,c = pair # unpacking the row-column values from thte pair
                if r>=0 and c>=0: #make sure no negative indices
                    try:
                        newval += corrcoef_val_per_couple(i, j, r, c, data)
                        # add to newval the sum of the multiplication of the two times
series of the original pixel and it's neighbor, divided by the multiplication of the
average values of them both
                        cnt += 1  #if we reached this we successfully tested the neighbor
                    except IndexError:  #this is for corners and edges
                        continue
            # normalize value (for edges)
            neighborCorrcoef[i,j] = newval / cnt  # divide the total value by the number of
neighbors that were tested and summed, and then write into the final matrix
            counter += 1  # increment for progress
            if counter % 10000 == 0:  # progress print
                print(f"Progress: {counter}/{n} ({counter/(n) * 100:.2f}%)")
    return neighborCorrcoef
```

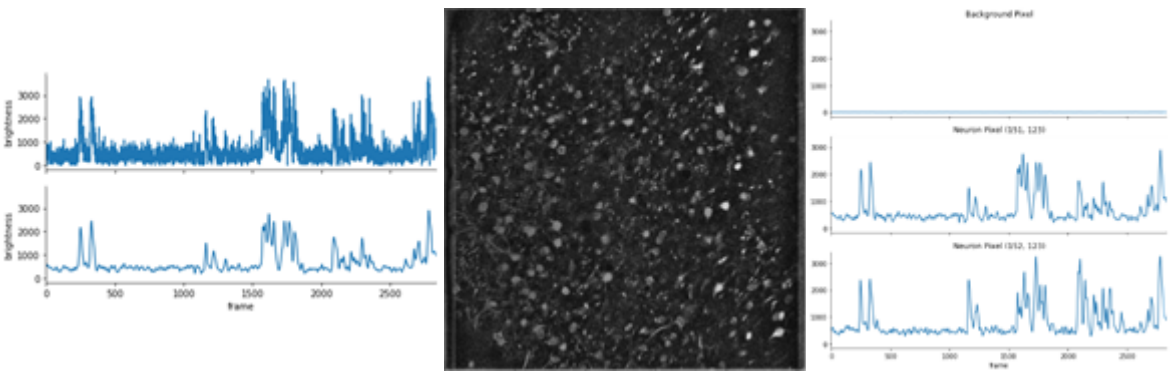


Figure 4: Pixels value over time. From left to right: Unfiltered and filtered neuron, over time; Pearson correlation score for each pixel with its neighbors, summed over time; Correlation between pixels: noise vs. adjacent pixels.

Method #8 (K-means):

To determine the right threshold for separating the background from foreground, we used the K-means algorithm as implemented in Exercise 7. We applied the K-means to the results of the correlated pixels.

```
### This library performs Kmeans and separates different values of brightness

def create_threshold(data, K):
    """Initialize thresholds based on value range"""
    min_val = np.min(data)
    max_val = np.max(data)
    bins = np.linspace(min_val, max_val, num=K, endpoint=False)  # value histogram with
equal spacing
    bins=bins[1:]  # setting up bin threshold K-1
    return bins


def compute_segment_values(data, thresholds):
    """Return a list of all values of the image pixels, divided into segments (based on
their thresholds)"""
    segments = []
    # first segment:
    segment = data[(0 <= data) & (data < thresholds[0])]
    segments.append(segment)

    # all thresholds
    for i in range(1, len(thresholds)):
        segment = data[(thresholds[i-1] <= data) & (data < thresholds[i])]
        segments.append(segment)

    # last_segment
    segment = data[(thresholds[-1] <= data) & (data <= 1)]
    segments.append(segment)

    return segments


def compute_centroids(segments, thresholds):
    """Compute the centroids of K segments"""
    centroids = []  # initialize list
    thresholds = [0] + [thresh for thresh in thresholds] + [1]
    for i,segment in enumerate(segments):
        if len(segment) == 0:  # segment is empty
            centroid = (thresholds[i+1] - thresholds[i]) / 2  # mean between two thresholds
            centroids.append(centroid)
        else:
            centroids.append(np.mean(segment))
    return centroids


def compute_new_thresholds(centroids):
    # we now have K centroids.
    # The thresholds should be midway between each two centroids
    thresholds = []
    for i in range(1, len(centroids)):
```

```
        threshold = centroids[i-1] + ((centroids[i] - centroids[i-1])/ 2)
        thresholds.append(threshold)

    return thresholds


def centroids_stable(new_centroids, old_centroids, diff_thresh = 1e-5):
    """Returns True of the centroids are stable."""
    diff_centroid_arr = np.abs(np.array(new_centroids) - np.array(old_centroids))
    for cent_diff in diff_centroid_arr:
        if cent_diff > diff_thresh:
            # too much of a difference
            return False
    return True  # stayed stagnant


def segment_image(data, thresholds, centroids):
    """Segment the image for presentation"""
    segmented_data = data.copy()  # new version
    # first segment:
    segmented_data[(0 <= data) & (data < thresholds[0])] = centroids[0]
    # all thresholds
    for i in range(1, len(thresholds)):
        segmented_data[(thresholds[i-1] <= data) & (data < thresholds[i])] = centroids[i]
    # last_segment
    segmented_data[(thresholds[-1] <= data) & (data <= 1)] = centroids[-1]
    return segmented_data


def kmeans(data, K):  # deprecated
    "Implement K-means with K clusters"
    # initialize
    thresholds = create_threshold(data, K)  # K-1 thresholds
    segments = compute_segment_values(data, thresholds)  # should be K segments
    centroids = compute_centroids(segments, thresholds)  # should be K centroids

    # new centroids
    old_centroids = [0] * K

    # iterate
    while not centroids_stable(centroids, old_centroids):  # as long as the centroids are
not stable
        # reset thresholds to be midway between cluster centers
        thresholds = compute_new_thresholds(centroids)
        segments = compute_segment_values(data, thresholds)  # should be K segments
        old_centroids = centroids
        centroids = compute_centroids(segments, thresholds)  # should be K centroids


    # color image based on centroid values:

    segmented_data = segment_image(data, thresholds, centroids)
    return segmented_data
```

```
def kmeans_with_centroids(data, K):
    "Implement K-means with K clusters, return centroids"
    # initialize
    thresholds = create_threshold(data, K)  # K-1 thresholds
    segments = compute_segment_values(data, thresholds)  # should be K segments
    centroids = compute_centroids(segments, thresholds)  # should be K centroids

    # new centroids
    old_centroids = [0] * K

    # iterate
    while not centroids_stable(centroids, old_centroids):  # as long as the centroids are
not stable
        # reset thresholds to be midway between cluster centers
        thresholds = compute_new_thresholds(centroids)
        segments = compute_segment_values(data, thresholds)  # should be K segments
        old_centroids = centroids
        centroids = compute_centroids(segments, thresholds)  # should be K centroids


    # color image based on centroid values:

    segmented_data = segment_image(data, thresholds, centroids)
    return segmented_data, thresholds, centroids

def run_and_plot_kmeans(data, last_K=16):
    """Run K means algorithm and plot it, with various Ks"""
    plt.subplots(2,3, figsize=(16,16))

    plt.subplot(231)
    plt.imshow(data, cmap='gray')
    plt.title(f"Before segmentation")

    plt.subplot(232)
    K=2
    segmented_data, thresholds, centroids = kmeans_with_centroids(data, K)
    plt.imshow(segmented_data, cmap='gray')
    plt.title(f"After segmentation, Kmeans with K={K}")

    plt.subplot(233)
    K=3
    segmented_data, thresholds, centroids = kmeans_with_centroids(data, K)
    plt.imshow(segmented_data, cmap='gray')
    plt.title(f"After segmentation, Kmeans with K={K}")

    plt.subplot(234)
    K=6
    segmented_data, thresholds, centroids = kmeans_with_centroids(data, K)
    plt.imshow(segmented_data, cmap='gray')
    plt.title(f"After segmentation, Kmeans with K={K}")
```

```
    plt.subplot(235)
    K=10
    segmented_data, thresholds, centroids = kmeans_with_centroids(data, K)
    plt.imshow(segmented_data, cmap='gray')
    plt.title(f"After segmentation, Kmeans with K={K}")

    plt.subplot(236)
    K = last_K
    segmented_data, thresholds, centroids = kmeans_with_centroids(data, K)
    plt.imshow(segmented_data, cmap='gray')
    plt.title(f"After segmentation, Kmeans with K={K}")

    plt.tight_layout()

    return segmented_data, thresholds, K
```

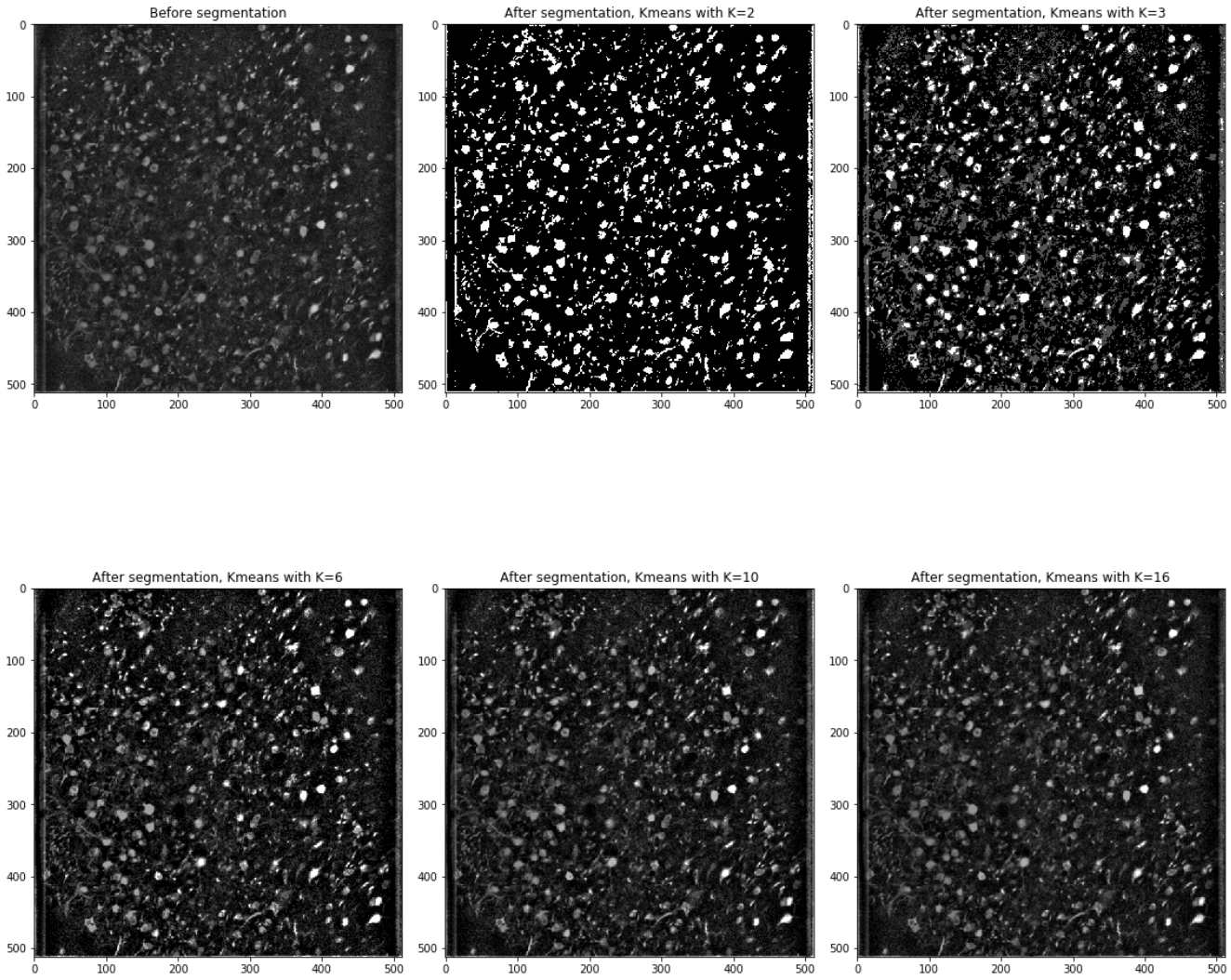


Figure 2: The results of K-means brightness segmentation. Shown are different Ks, each layer represented by the average brightness of the layer. As the value of K gets higher, the more the image resembles the original image.

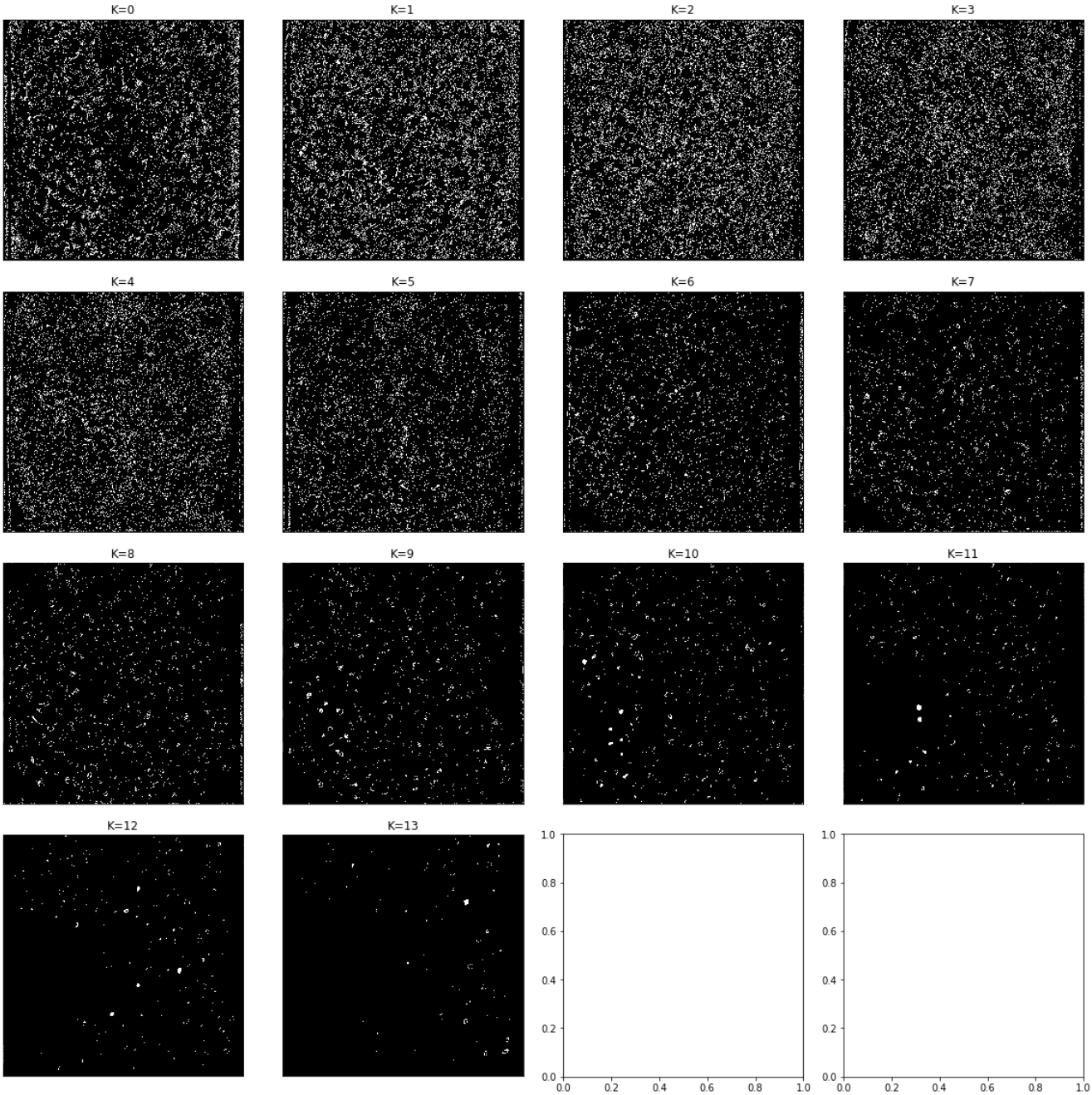


Figure 3: The results of K-means brightness segmentation. Shown are different layers produced by K-means. Each layer is between two thresholds, where the higher values of K represent the brighter values of the image.

Method #9 (Hit or Miss):

Using the neighbor correlation data has allowed us to accurately detect the population of "active" neurons, or the neurons that spike more often during the video. However, we appear to be missing a second population of neurons, which doesn't fire often during the video, and is therefore harder to detect. When examining the ground truth data, we observed that some of the harder-to-detect neurons look like bright rings with a dark interior. We hypothesized that we could detect them by using a hit or miss-like operation, where we would convolve the image with a structuring element made up of a positive part and a negative part (“donut”, see **Fig. 4**). And indeed, this convolution has produced an image in which the centers of "passive" neurons were marked by a bright center, allowing us to isolate them by thresholding.

We then combined the seeds of those isolated neurons with the seeds we previously isolated from the neighbor correlation data, as well as a binary erosion, and applied watershed to the combined seeds. This pipeline improved our recall score to 0.4121 (see **Table 1**).

```
def hitormiss_donut(img, rad_inner=5, width=2):
    """
    Hit or miss- donut shaped, for less active neurons.
    rad_inner: radius of negative values
    width: radius of positive values (around the edges)
    """
    r1 = rad_inner  #radius of hit-or-miss-circle (negative values)
    r2 = rad_inner + width  # radius of positive values in hit-or-miss-circle (to emphasize
edges)

    disk = np.int8(morphology.disk(r1))
    frame_with_disk = np.int8(np.zeros([r2*2+ 1, r2*2+1]))  #  frame
    frame_with_disk[(r2-r1):r1*2 + (1+r2-r1), (r2-r1):r1*2 + (1+r2-r1)] = disk  # disk in
the center of frame

    # outer circle
    larger_disk = np.int8(morphology.disk(r2))  # larger disk, in the shape of frame
    donut = larger_disk -2 * frame_with_disk

    # convolve the disk with the image
    conv = signal.convolve2d((norm_data(img) - 0.5), donut, mode='full')
    (cx,cy) = np.shape(conv)
    conv = conv[(cx-512)//2:cx-(cx-512)//2, (cy-512)//2:cx-(cy-512)//2]  # resize
    return conv
```

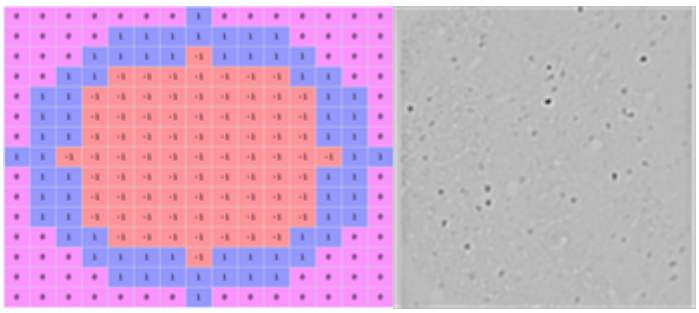


Figure 5: Donut hit-or-miss convolution. From left to right: Donut shaped morphology filter; Hit-or-miss convolution applied to the summed, equalized FOV.

Method #10 (Grid search):

To optimize the parameters of the methods above, we implemented a grid search over the different parameters used in the pipeline, including brightness thresholds, size threshold, filter sizes and parameters, etc. The grid search code appears below, although we do not recommend running it (it might take a while). This method found parameters which yielded better results (**Table 1**).

```
# Grid search on the following:
# show(hitormiss_donut(new_summed))
```

```
grid_search_res = []
bool_img_disk_range1 = range(1,2)
rad_inner_range = range(4,7)
width_range = range(2,4)
hms_max_size_range = range(8, 13)
# thresholds
hms_labeling_threshold_min_range = range(73, 76)
hms_labeling_threshold_max_range = range(86, 91)
bool_img_disk_range2 = range(1,2)

max_size_range = range(21, 24)
min_size_range = range(4, 7)
dial_rad_range = range(6, 10)

img_tuple = (corrcoef_norm_eq_no_frame, summed_eq)

# combinations
total_counts = count_combs([bool_img_disk_range1, rad_inner_range, width_range,
hms_max_size_range, hms_labeling_threshold_min_range, hms_labeling_threshold_max_range,
bool_img_disk_range2, max_size_range, min_size_range, dial_rad_range,img_tuple])
print(f"Trying {total_counts} combinations")
start_time = time.time()
print(f"Started at {time.strftime('%H:%M:%S', time.gmtime(start_time))}")
dict_for_bad_eval = {x: 0 for x in ('combined', 'inclusion', 'precision', 'recall',
'exclusion')}  # if eval fails

# grid search
counter = 0
for bool_img_disk_size1 in bool_img_disk_range1:
    bool_img = morphology.erosion(seg_no_frame, morphology.disk(bool_img_disk_size1)) > 0
    # hit or miss
    for rad_inner in rad_inner_range:  # positive radius
        for width in width_range:  # negative radius
            if rad_inner + width >= 12:  # bigger than the neuron radius
                continue  # ignore
            hms = norm_data(hitormiss_donut(summed_eq, rad_inner, width))
            # show(draw_circles(hms, cont))
            # hms_with_cont = draw_circles(hms, cont)
            for min_threshold in hms_labeling_threshold_min_range:
                min_threshold /= 100
                for max_threshold in hms_labeling_threshold_max_range:
                    max_threshold /= 100
                    hms_labeled = ndimage.label(((min_threshold < hms) & (hms <
max_threshold)))[0]  # between thresholds
                    # show(hms_labeled)
                    for hms_max_size in hms_max_size_range:
                        hms_selected = size_selection(hms_labeled, hms_max_size, 0)  #
selecting smaller neurons
                        hms_selected_bool = hms_selected > 0
                        for bool_img_disk_size2 in bool_img_disk_range2:
                            bool_img2 = morphology.erosion(hms_selected_bool,
morphology.disk(bool_img_disk_size2))
                            comb_bool_img = bool_img | bool_img2
```

```
                            for dial_rad in dial_rad_range:
                                for max_neuron_size in max_size_range:
                                    for min_neuron_size in min_size_range:
                                        for i, img in enumerate(img_tuple):
                                            json_fname = "grid_search_json"
                                            size_selected_after_ws = watershed(img,
comb_bool_img, json_fname, dims, dial_rad=dial_rad, max_neuron_size=max_neuron_size,
min_neuron_size=min_neuron_size)
                                            if size_selected_after_ws is None:  # if size
selection didn't keep anything
                                                d = {"bool_img_disk_size1":
bool_img_disk_size1,
                                                 "rad_inner": rad_inner,
                                                 "width": width,
                                                 "min_threshold": min_threshold,
                                                 "max_threshold": max_threshold,
                                                 "hms_max_size": hms_max_size,
                                                 "bool_img_disk_size2":
bool_img_disk_size2,
                                                 "dial_rad": dial_rad,
                                                 "max_neuron_size": max_neuron_size,
                                                 "min_neuron_size": min_neuron_size,
                                                 "img": i,
                                                 "evaluation": dict_for_bad_eval
                                                }
                                            else:
                                                # img_w_contours =
draw_circles(corrcoef_norm_eq_no_frame, find_contours(size_selected_after_ws))
                                                # show(img_w_contours)
                                                evaluation_res = evaluation(json_fname)

                                                d = {"bool_img_disk_size1":
bool_img_disk_size1,
                                                     "rad_inner": rad_inner,
                                                     "width": width,
                                                     "min_threshold": min_threshold,
                                                     "max_threshold": max_threshold,
                                                     "hms_max_size": hms_max_size,
                                                     "bool_img_disk_size2":
bool_img_disk_size2,
                                                     "dial_rad": dial_rad,
                                                     "max_neuron_size": max_neuron_size,
                                                     "min_neuron_size": min_neuron_size,
                                                     "img": i,
                                                     "evaluation": evaluation_res
                                                    }
                                            grid_search_res.append(d)
                                            os.remove(fix_json_fname(json_fname))
                                            # counter
                                            counter += 1
                                            if counter % 100 == 0:
                                                elapsed_time = time.time() - start_time
```

```
                                                print(f"Progress: {counter}/{total_counts}
({counter/total_counts * 100:.2f}%). Elapsed: {time.strftime('%H:%M:%S', elapsed_time)}")
                                            # once in 100:
                                            if counter % 1000 == 0:
                                                # with open('res_last.pickle', 'wb') as f:
# used to be
                                                with open('res_last_2.pickle', 'wb') as f:
                                                    # Pickle the 'data' dictionary using
the highest protocol available.
                                                    pickle.dump(grid_search_res, f,
pickle.HIGHEST_PROTOCOL)
                                                    print("Results pickled")

print(f"Finished at {time.strftime('%H:%M:%S', time.gmtime(start_time))}")
with open('res_last_2.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(grid_search_res, f, pickle.HIGHEST_PROTOCOL)
    print("Results pickled")

# print results:
print("Index for each category:")
for k in grid_search_res[0]['evaluation'].keys():
    print(f"Best {k} score:")
    ind = np.argmax([x['evaluation'][k] for x in (x for x in grid_search_res)])
    print(f"Index: {ind}")
    print(f"Evaluation scores:")
    print(grid_search_res[ind]['evaluation'])
    print("Values:")
    print(grid_search_res[ind])
    print("**************")

# running the pipeline with the best recall and combined scores:
def pipeline_with_params(params_dict, fname, img_tuple = (corrcoef_norm_eq_no_frame,
summed_eq)):
    """Run pipelines with params_dict (produced by the grid search).
    Save final figures and return evalution score."""
    # unpacking parameters
    bool_img_disk_size1 = params_dict['bool_img_disk_size1']
    rad_inner = params_dict['rad_inner']
    width = params_dict['width']
    min_threshold = params_dict['min_threshold']
    max_threshold = params_dict['max_threshold']
    hms_max_size = params_dict['hms_max_size']
    bool_img_disk_size2 = params_dict['bool_img_disk_size2']
    dial_rad = params_dict['dial_rad']
    max_neuron_size = params_dict['max_neuron_size']
    min_neuron_size = params_dict['min_neuron_size']
    img = img_tuple[params_dict['img']]
    # results
    bool_img = morphology.erosion(seg_no_frame, morphology.disk(bool_img_disk_size1)) > 0
    hms = norm_data(hitormiss_donut(summed_eq, rad_inner, width))
    hms_labeled = ndimage.label(((min_threshold < hms) & (hms < max_threshold)))[0]  #
between thresholds
```

```
    hms_selected = size_selection(hms_labeled, hms_max_size, 0)  # selecting smaller
neurons
    hms_selected_bool = hms_selected > 0
    bool_img2 = morphology.erosion(hms_selected_bool, morphology.disk(bool_img_disk_size2))
    comb_bool_img = bool_img | bool_img2
    # json_fname = "grid_search_json"
    size_selected_after_ws = watershed(img, comb_bool_img, fname, dims, dial_rad=dial_rad,
max_neuron_size=max_neuron_size, min_neuron_size=min_neuron_size)
    evaluation_res = evaluation(fname)
    print(evaluation_res)
    img_w_contours = draw_circles(corrcoef_norm_eq_no_frame,
find_contours(size_selected_after_ws))
    show(img_w_contours)
    im = Image.fromarray(img_w_contours)
    im.save(fname+'_corrcoef.jpg')
    img_w_contours = draw_circles(summed, find_contours(size_selected_after_ws))
    show(img_w_contours)
    im = Image.fromarray(img_w_contours)
    im.save(fname+'_summed.jpg')

# best recall:
ind = 12895
best_recall_dict = grid_search_res[ind]
pipeline_with_params(best_recall_dict, 'best_recall')

# best combined:
ind = 14404
best_combined_dict = grid_search_res[ind]
pipeline_with_params(best_recall_dict, 'best_combined')
```

Discussion

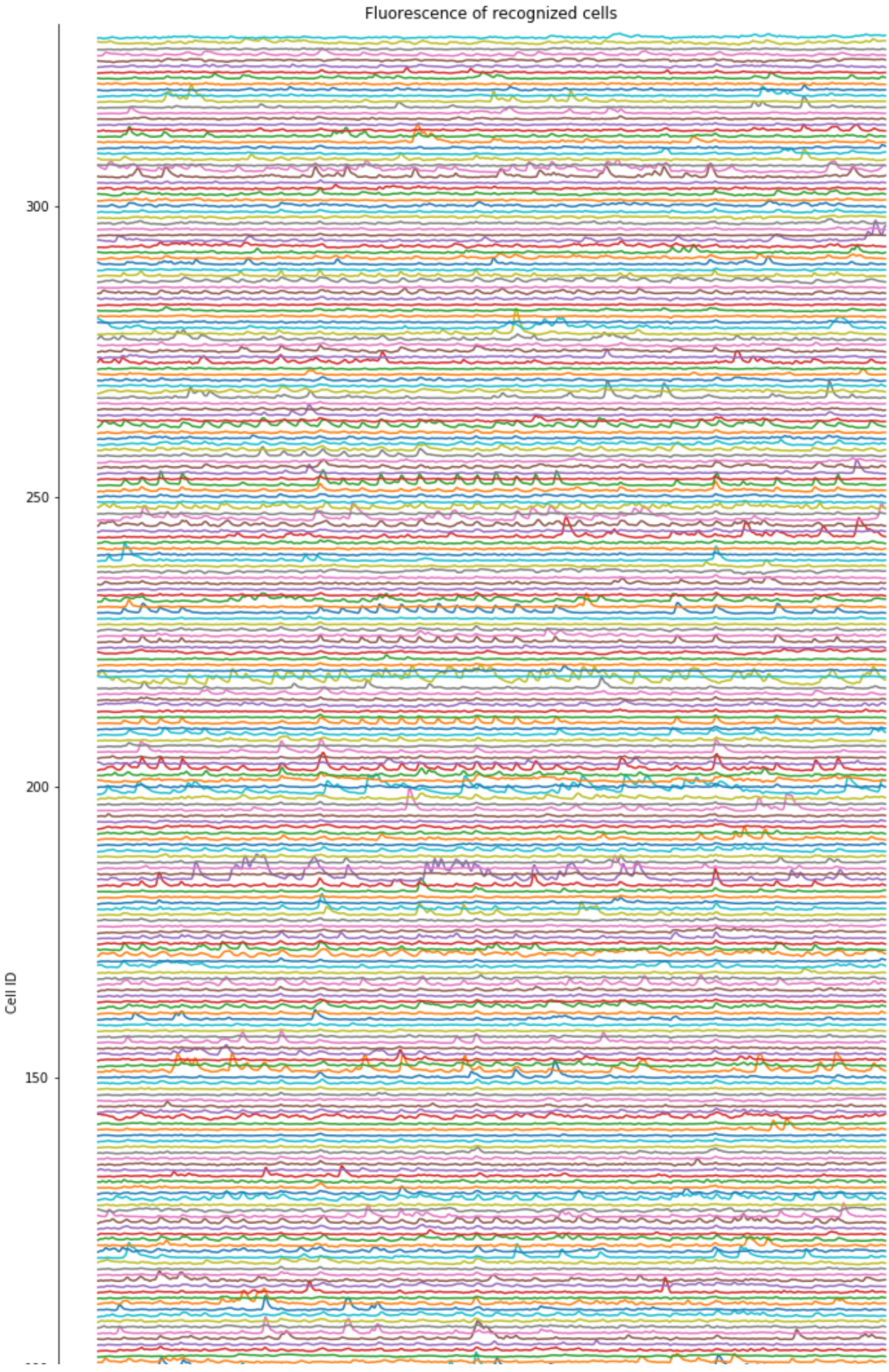
In this project, we used various morphological filters, frequency filters, segmentation algorithms and various algorithms we applied and developed for our data. We compared our results to a given “ground truth” neuron positions and used the ground truth to measure our results and optimize our results. Although there were certain discrepancies between the neurons detected by our algorithms and the ground truth, we did manage to achieve increasing improvement with the various methods. However, due to the lack of time, we have not been able to pursue additional directions that we believe had a lot of potential.

For example, since we know we are able to easily and accurately detect the active neurons in the FOV, if we could estimate the number of expected “silent” neurons from the number of discovered active neurons (say, we expect 10% of the neurons to be active), then we could have the algorithm adjust its sensitivity/precision parameters (such as parameters for size selection, or binary mask thresholds) to fine-tune the number of discovered neurons to the number of expected neurons.

Moreover, we can use various machine learning and deep learning algorithms to find delicate thresholds and detect neurons which are especially hard to detect.

Appendix A: Neuron Fluorescence Values

We were also requested to plot the fluorescence levels of each of the neurons. The whole process and code are available in the “Calcium Imaging - Plot Fluorescence” notebook, attached to this project files. Below is a plot of a selection of 60 neurons out of the neurons we detected (**Fig. 6**). In the next page, you may find the entire neurons detected (**Fig. 7**).


Fluorescence of recognized cells
300
250
200
150
Cell ID

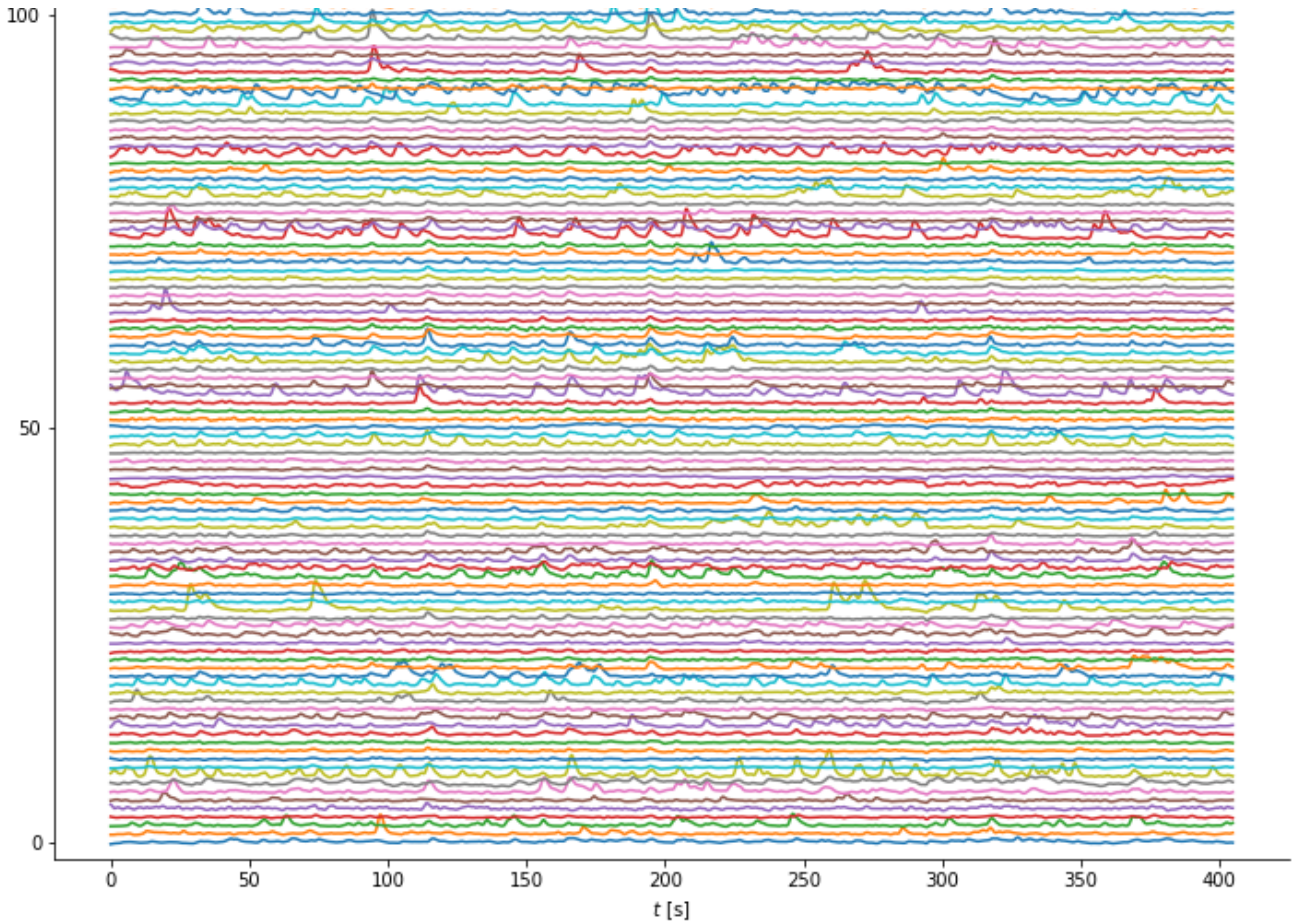


Figure 6: Fluorescence values of 60 of the neurons detected by our methods, after normalization and removal of artifacts.

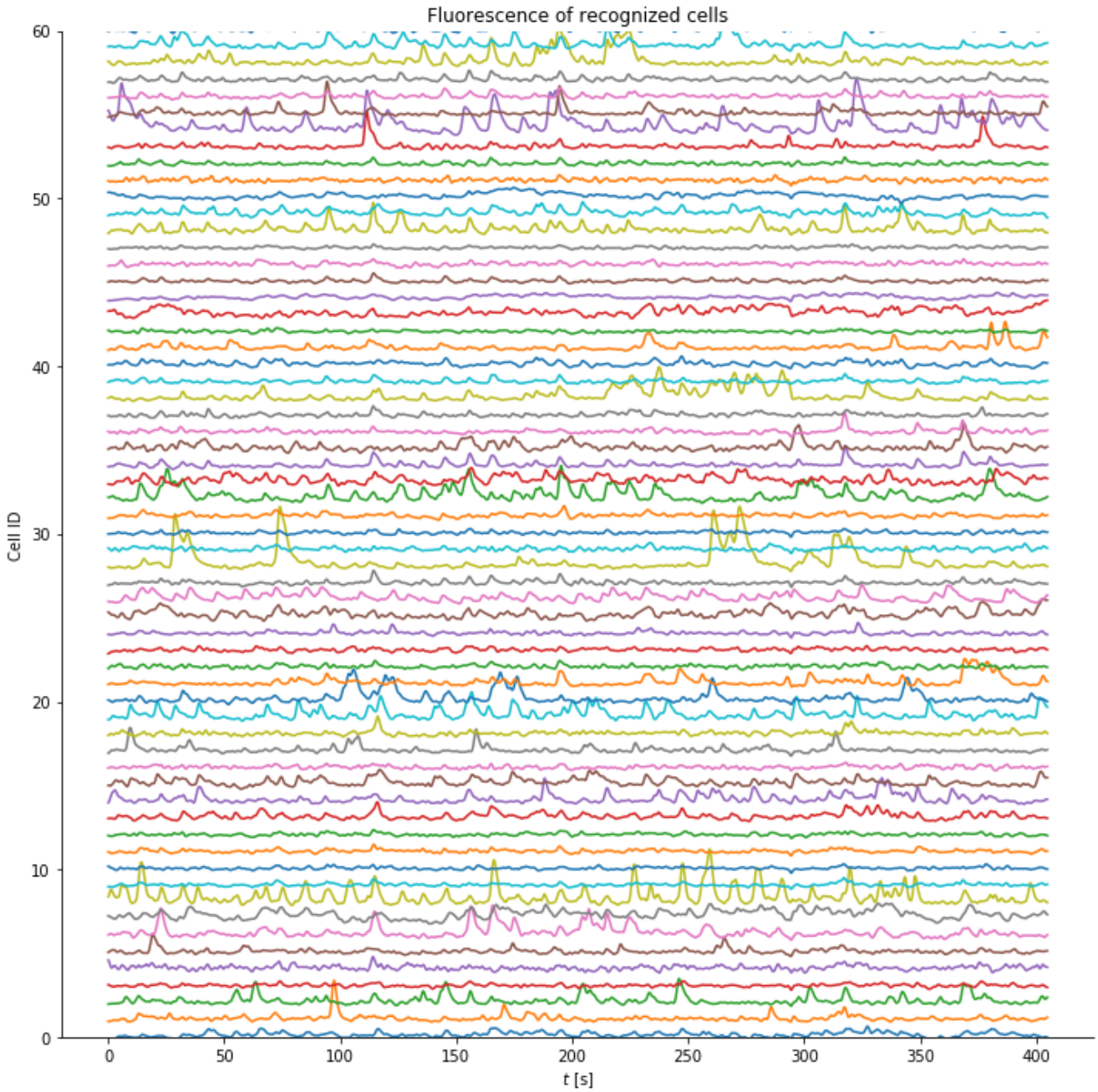


Figure 7: Fluorescence values of all neurons detected by our methods, after normalization and removal of artifacts.