Application of K-Nearest Neighbor in Identifying serotonin labeling

By Jacob S. Salminen

# KNN as a Machine Learning Tool

KNN or k-nearest neighbor is a non-parametric method used for classification and regression. In the case of fluorescent imaging, we will be using KNN for classifying positively labeled areas of serotonin. The variable we use for classification is the intensity of the pixels in the image. For example, a positively labeled pixel would have some intensity value that is significantly different from the background. We define the background of the image as any part of the fluorescent image that we can identify as the slide or parts of the spinal cord section that are not fluorescently labeled.

The goal of the KNN is to use intensity values for classifying fluorescently labeled serotonin. However, there are more variables other than intensity that would classify a pixel as being positively labeled or not, i.e. shape, location, and neighboring pixels. Because we are not considering these variables when using this instance of KNN there will be limitations involved when deciphering whether a pixel is positively labeled or not. These limitations also mean that when we apply the KNN there is a potential for false positives and false negatives. Understanding these limitations is important when interpreting the output of the KNN data, we can also reduce these errors by increasing the amount of reference or “training” data we consider when determining classification of the query or “test” data; see **Figure 1** for a block flow process diagram of the KNN technique.

In reference to **Figure 1** there are a couple of key concepts that need to be addressed to fully understand the KNN procedure for identifying positively labeled areas in an image. The first is the idea of independent analysis of each test point. Starting at the top of the block flow diagram we have a test point and a set of training data. A test point is the location and intensity of some pixel in the fluorescent image that we are trying to classify. A training data set is a group of points that represent data that has already been classified, either manually by a professional or using another already proven technique. To generate a value that represent similarity between the test point and some point in the training data set we use a distance measure, and we calculate the distance measure between the test point and each point in the training data set.

We then take these distances, sort them from the smallest distance to largest distance, and then we find the points in the training data set that have the smallest distance between them and the test point. The “k” in KNN is the amount of training points we consider when determining the classification of the test point. The “neighbors” in KNN are the points in the training data set that are closest to the test point. After we find the k nearest neighbors to the test point in the training data set, we can use basic statistical measures, e.g. mode, to determine the classification of the test point based on the data that has been determined to be most similar to it in the training data set. WE do this by simply applying the statistic to the group of the neighbors.

Generate distances between a test point and a training data set at point n: Dn = sqrt(t-Tn)2

Training dataset (with N points): **T1,T2,…,TN**

Sort Distances from smallest to largest: sort(D) = Ds

Take the k first elements, or neighbors, in sorted distances. i.e. the first k elements that are closest together: Ds1,Ds2,…,Dsk

Find the k elements inside the training data set where Ds1 = Tn1,…, Dsk = Tnk. i.e. we gather the labels for all the neighbors: Tn1,Tn2,...,Tnk. Note that T1 is different labeling than Tn1.

determine classification using the mode between the k elements: mode({Tn1,Tn2,...,Tnk})

Figure KNN Block Flow diagram

# CUDA Implementation

Although the procedure for KNN involves some simple calculations, the technique requires hundreds of thousands of points in the training data set to be relatively error free, i.e. lacking false positives and false negatives. This means when a computer runs the KNN procedure it requires a lot of computational power to classify even just one point in the test data set, which is made up of hundreds of thousands test points.

We can overcome this barrier by using parallel processing. Typically, a computer runs a program on a single thread, which is a single processing chain. This means that before we can test the next test point in the test data set we would need to classify the test point that is currently be classified. This is a major bottle neck in computing. By using multiple threads and a graphical processing unit (GPU) we can parallelize the KNN technique and increase processing time exponentially.

Compute Unified Device Architecture (CUDA) is a programming language that allows us to program GPU’s to run specific programs, called kernels, on multiple threads at the same time. CUDA is an extension of the C-language, which was used to write the Windows OS, but we can use python to implement CUDA with no loss in processing time and a huge gain in readability.

Using a python implementation of KNN, only using the python language no CUDA, the time to process one 720x960 image would be around 77.7 hours. Using CUDA it takes around 46.7 minutes. 100 times faster!

Zhongheng Zhang, “Introduction to Machine learning: K-Nearest Neighbors”, *Annals of Translational* *Medicine*, 2016, <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4916348/>

S. Liang, Y. Liu, C. Wang and L. Jian, "A CUDA-based parallel implementation of K-nearest neighbor algorithm," 2009 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, Zhangijajie, 2009, pp. 291-296, doi: 10.1109/CYBERC.2009.5399145

CUDA Code github: <https://github.com/chrischoy/knn_cuda>

# Methods

This section is broken down into a literary explanation of the overall analysis process and an example of the application using a poorly represented training dataset\*.

The detection process has [blank] steps: load image data, filtering and normalization\*\*, generation of training and test data matrices, and finally KNN and reorganization of data.

## Preparing Data for Processing:

Before we can throw our image data into the KNN we must first load, filter, and convert the data to data structure the KNN can more efficiently process. For this part feel free to follow along in python running the small bits of code as I go over them, so you can get a feel for how the image is processed (See KNN\_alg\_GPU\_V2.py and ThreeD\_Recon.py).

Our first step is loading the image into memory. We use OpenCV for this task as the module is the most integrated with Numpy, Numpy being the premier module for matrix manipulations in python. Our next step is to filter out the background noise in the image. Ultimately, we want to remove the low frequency background from the high frequency signal that represents the fluorescent serotonin. We will now take a small detour and talk about what frequency in a function means and how we can use this characteristic of functions for the purpose of data processing.

## Fourier Analysis (Frequency Distributions):

We define low frequency areas as large regions with similar intensity values and high frequency areas as small regions with similar intensity values or “spikes” in intensity values. The notable difference between frequencies in images are the length or size of the regions containing similar intensity values. In Figure 2 we have a function in **a)** that is a simple pulse with a value of **A** for some width **W**. The frequency distribution of this pulse is represented in **b)**, lets take some time to understand what’s going on in this “transformation” form **a)** to **b)**.

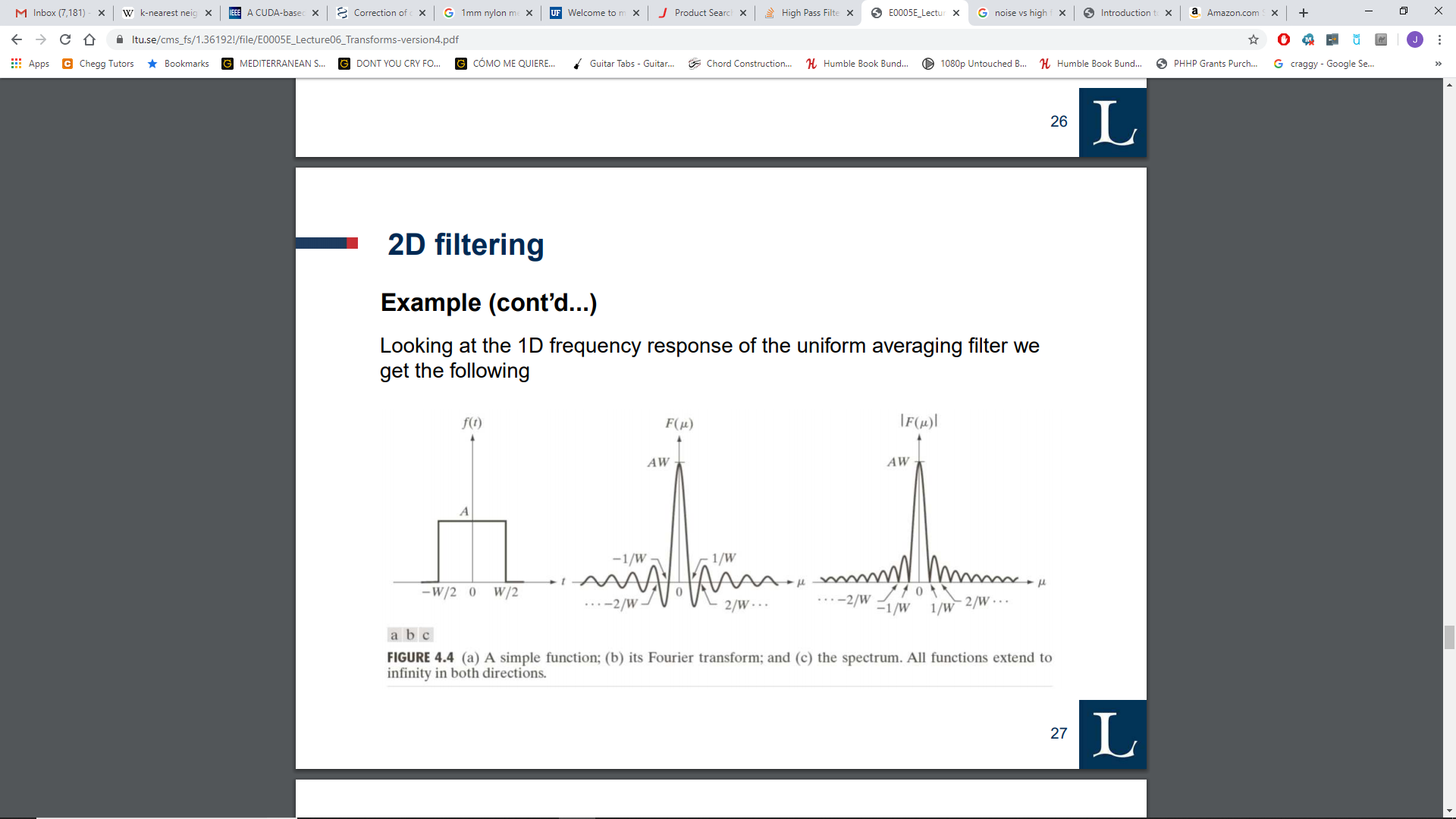


Figure a) function that is some value, A, for a region of width, W. b) The frequency distribution, Fourier transform, of the function in a

\*Note: the data set is a poor representation as it uses detection methods, unlike a manual labeling method, that don’t accurately account for the various factors effecting positively detecting serotonin in an image.

As I mentioned previously frequency is dependent on the length of the region with the similar intensity. When we analyze the frequency distribution of a simple square wave, the function in **a)**, there is some notable features that arise: a spike about the origin and sinusoidal appendages propagating away from the spike. The spike represents the width and intensity of the square wave part, while the appendages represent the discontinuity from the **y=**A value to the **y=**0 value in the function in **a)**. These oscillations are a sign of a very high frequency. If we wanted to remove this high frequency area, the area in **a)** that has a drastic change in its **y** value (i.e. the -w/2 and w/2 x value areas), we eliminate (set to 0) all the values in the appendages other than the spike. Lets look at a function that is continuous to give you a better idea of how the frequency changes.

From now on we’ll be working in python. If you would like to follow along in the “ThreeD\_Recon.py” the functions are: fourier\_fun(), quadratic(), sine\_wave(), and square\_wave(). We reduce the drastic change in **y** values by “smoothing” them out using a quadratic, which is considered a continuous function. This reduces the amount of high frequency items we have in our frequency distribution.

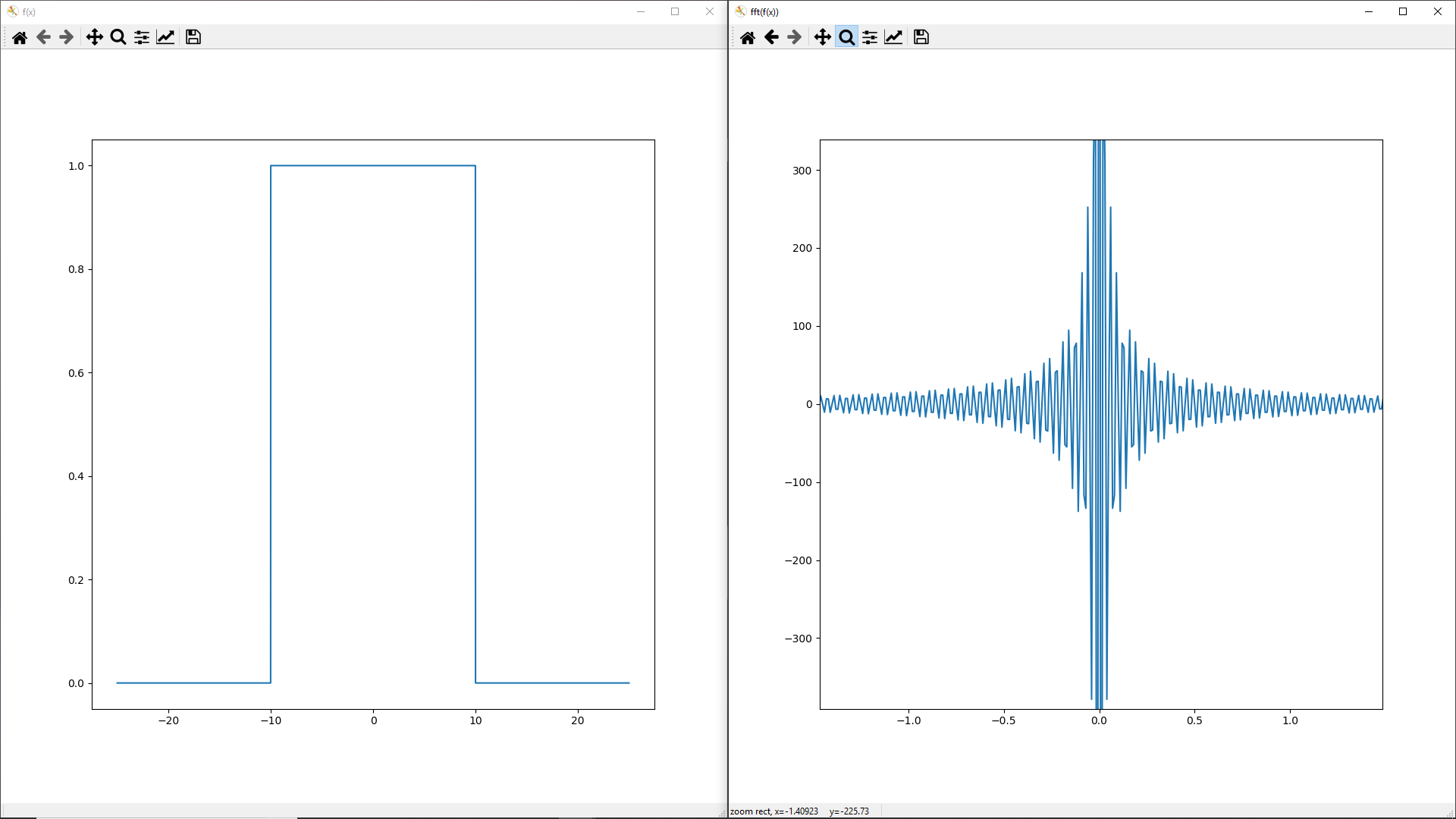


Figure left: square wave function in python, right: frequency domain of square wave

Note the oscillating appendages in **Figure 3**, as seen in the previous example, but also notice how in **Figure 4** these appendages are severely reduced proximal to the spike and become non-existent after a certain point. Convince yourself that because we “smoothed” the square wave that we now have fewer high frequencies. Also play around with the width and intensity of the functions to get a better idea of how the frequency distribution changes with these values. Also try the sine wave, which spans a singular frequency, what does that frequency distribution look like?

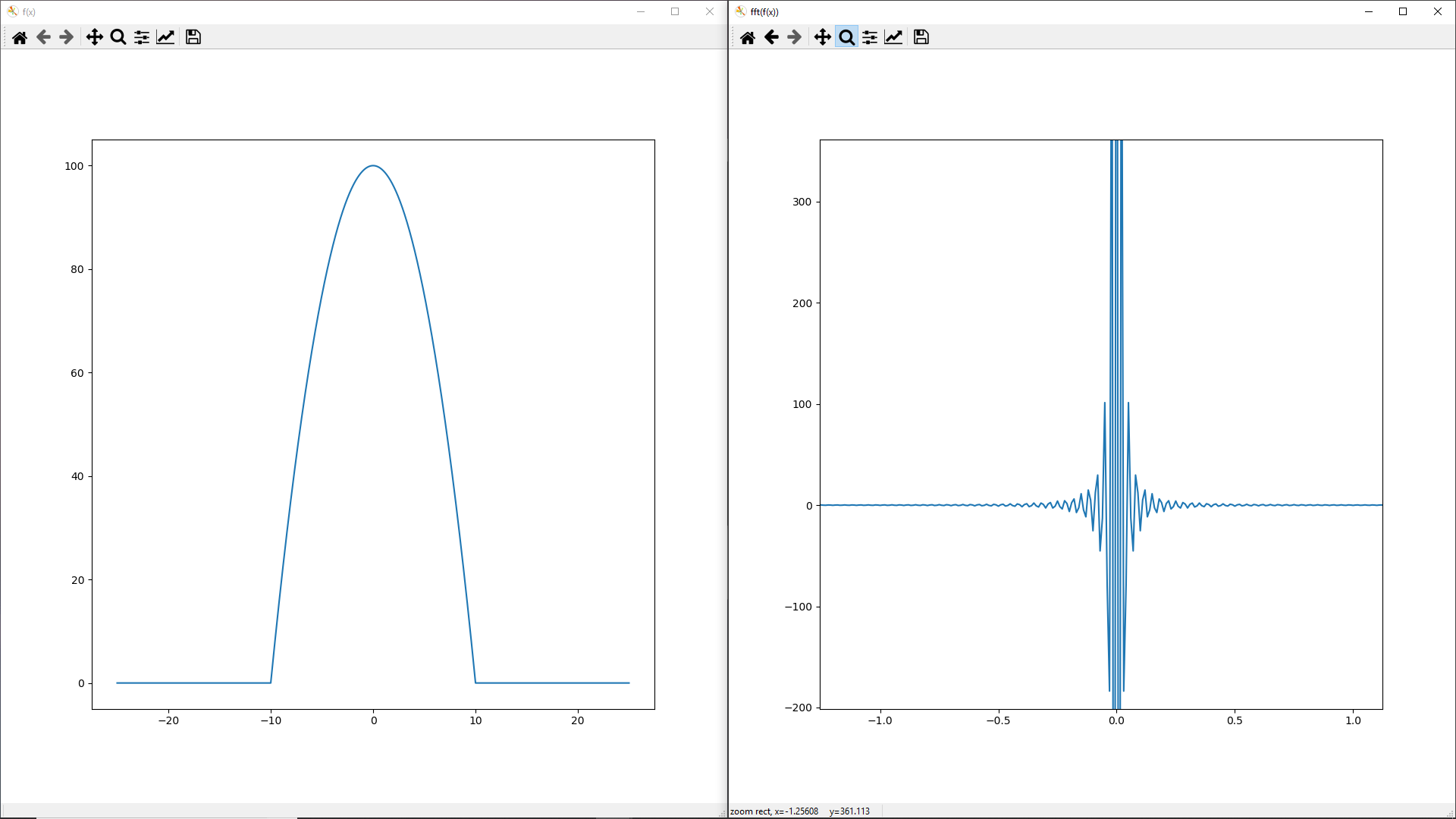


Figure left: quadratic function in python, right: frequency distribution of quadratic function

This is a large oversimplification of what is really happening when we use a Fourier transform to analyze functions and their frequency distributions. What is really happening is more complicated and has more caveats that, if someone were interested, should be read about in a book. See: Applied Fourier Analysis by Tim Olson (ISBN 9788-1-4939-7393-4)

## Gaussians, Convolution, and Training Set Creation:

After we filter out the frequencies we don’t want, we turn to creating a **test domain** and a **training dataset**. For those unfamiliar a **domain** is a region is the complete set of possible values of the independent variable, or in our case the **test domain** is the region on the image with which the KNN will perform its analysis.

### Test domain

To get the test domain we convolve, a technique by which a filter is passed over an array, a gaussian function across our filtered image. This technique is used in machine vision to identify important features in an image and is also a way of “smoothing” an image (e.g. gaussian blur). See: KNN\_alg\_GPU\_V2.py. Ultimately we generate a bitmap by thresholding the gaussian image at some set value, note this method does not take into account image to image variability, and use the bitmap in identifying points on the image that we will put into the KNN.

### Training dataset

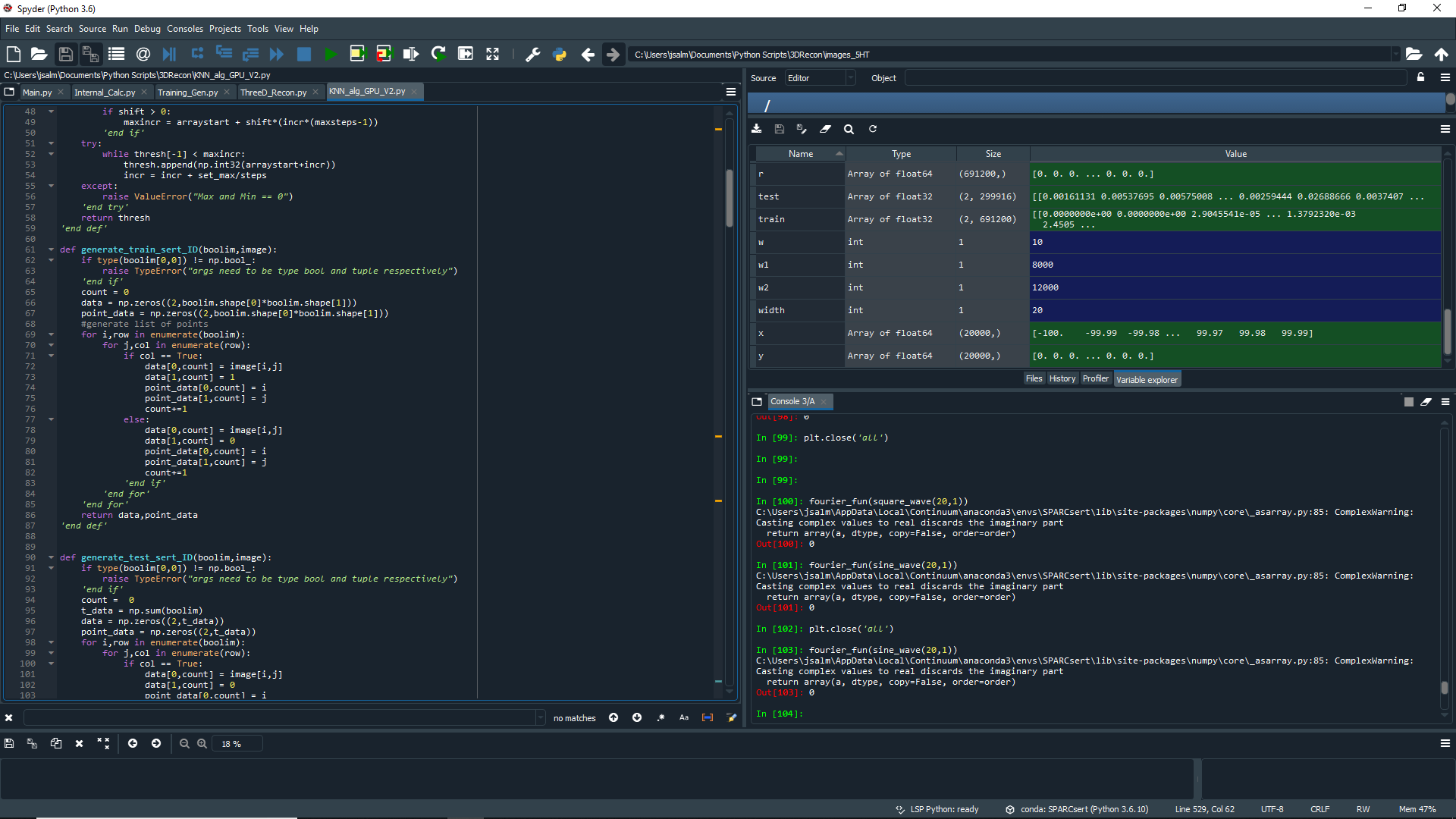
The training dataset is a set of points that are positively identified as being serotonin labeled. To create this dataset we can use a manual method by which a person uses a computer application to highlight areas on the image where the serotonin exists. We can also use a less appreciable method such as a complicated algorithm. The algorithmic method is less desirable as it also requires computation time and isn’t an arguable way of identifying regions of interest in the complexities of fluorescence imaging.

## Data Vector Creation:

The data vector creation methods are located in the “KNN\_alg\_GPU\_V2.py” file as: generate\_train\_sert\_ID() and generate\_test\_sert\_ID(). If you would like to follow along, I will be refencing some snips of code in these functions.

The main idea here is to take an image and a mask, or domain, and create vectors, single dimensional data arrays, that we will be putting into our KNN. The reason we want them in vectors is so that the KNN can more easily organize and manage the data. Vectorizing also allows for the KNN to be more versatile as we can theoretically vectorize any chunk of data whether it be survey results or images.

Let’s start by breaking down what the generate\_train\_sert\_ID() method does to an image.



The first 3 lines of the function contain an “IF” statement that checks to make sure the boolim, or mask, contains bools, or truth values (True and False). We do this as a precautionary measure as the function would not work properly if that condition is false. Next we initialize 3 variables: count, data, and point\_data. The count variable is an incrementing value that controls the index in each of the data vectors to which we assign values. This is important as we will be looping over the entire image to linearize, or create a one-dimensional array, from the data extracted. The data and point\_data variables are there for storing that data from the image and are 2 Dimensional arrays, but can in this state can be easily spliced, or split, to create a 1 dimensional array, our ultimate goal. We start them as 2 dimensional so that we can have reference points that can be easily indexed and found later in the KNN process.

There’s a good bit happening in the “FOR” loops that follow the variable declaration, so lets go through that line by line.

Line 8: “for i,row in enumerate(boolim)”, enumerate is a function that extracts the contents of the index counter, I, and the contents of the 0th index, row values, from an array. We use a “FOR” loop to go row by row through the array extracting its contents and keeping track of the row number we are on. Enumerate() especially useful as its JIT encoded, Just-in-time, so it runs around the same speeds as machine code.

Line 9: “for j,col in enumerate(row)”, this is a step further where now we loop through the contents in each row we are looking at, so instead of extracting rows of data we are now extracting individual points in the array. Lines 1 and 2 tend to go together when we are dealing with arrays as its one of the quickest ways to loop over array data.

Line 10: “if col == True”, recall that we wanted to insure the boolim data was Booleans, this line is where that becomes crucial. As we loop through every point in the array we are checking to see if the mask is true or false, and we assign specific values to our data vectors accordingly.

Lines 10-14 and 16-20: In these two chunks of code we use indexing to assign values to our data vectors. For our data variable, specifically for the training dataset, we want the intensity value and the truth value, the value that represents if serotonin is present or not, and for our point\_data variable we want the x and y, i and j, values for each point in the image. The if statement in line 10 controls whether we have a 1 or a 0, true or a false respectively, in our data vector.

Finally, we return our two data vectors so that we can later use them in our KNN.

The generate\_test\_sert\_ID() method is similar, but does not include the truth value assignment as that is the classification we want the KNN to determine. In this case we do output a single dimensional array and we won’t need splicing in the future. Take a look at the function located in the “KNN\_alg\_GPU\_V2.py” file.

## KNN:

The explanation given at the beginning of this guide is a good overlay of how a KNN works, but when implementing the KNN in computer language we have several factors we need to consider on top of ensuring the proper algorithmic implementation of the KNN. Our first consideration is processing speed, we will be sorting through and performing calculations on a substantial amount of information. Secondly, we should consider where the information is being accessed on the computer, are we putting the data in the memory on the motherboard or the GPU device. And lastly, we need to consider bugs that might cause miss calculations and to ensure we have redundancies in our code to catch for these kinds of things.

Speeding up the Process

KNN, as simple as it may look compared to other algorithms, can require a lot of computation time when performed on large data sets. When we implement the KNN in the computer we need to make sure we are performing the algorithm in the most efficient way possible, by limiting the lines of code we use, and we are performing the algorithm in multiple threads or streams. We can use CUDA processing to ensure we accomplish both goals. CUDA is a computer language developed by NVIDIA that allows you to control the graphics card on your computer if its NVIDIA branded, of course. CUDA is like the C language and can be complex and difficult to implement. Luckily, we have python and a handy module for implementing the CUDA code.

As a lot of this explanation would require knowledge of memory management, GPU processing, parallelization, and thread-block-grid structure; I’m going to post a couple of good links that I used to learn about these concepts and go more in depth into how we implement this in python. I recommend listening to the 30 minute video at minimum. The video goes over important concepts that are needed when beginning to explore CUDA.

CUDA Resources:

1. Official introduction to CUDA programming: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction>
2. Lecture on CUDA in Python: <https://www.youtube.com/watch?v=CO4ifMknS84>. He does a good job of explaining the advantages and disadvantage of CUDA implementation.
3. Numba, a python module that uses Just-in-time coding to implement CUDA: <https://numba.pydata.org/numba-doc/latest/cuda/overview.html>

All the modules I will be talking about will be located in the KNN\_alg\_GPU\_V2.py file, their names are: distance\_kern, sorting\_kern, knn\_cuda, main. The big picture here is we have two kernels and two implementation python functions. The kernels tell the threads on the device what to do, while the knn\_cuda and main functions set up our device, GPU, to support the kernels. Set up involves allocating memory, determing thread/block/grid strucutre on GPU, and host processing (operations we can’t use the GPU for).

* Distance\_kern():
  + This function is a CUDA kernel that calculates the distances between a point in a test dataset and all the points in the training dataset for all the points in the test dataset. This function generates an array containing all the distances that we will need to determine our neighbors in the sorting step.
  + NOTE: This is a parallelized function, so indexing becomes more abstract than usual as the kernel will asynchronously, in no particular order, assign and process values on our shared memory. Make sure you understand the difference between a synchronous and asynchronous processing. Heres a formal definition: <https://docs.nvidia.com/cuda/cuda-runtime-api/api-sync-behavior.html>, and here’s a more simplified understanding: <https://www.youtube.com/watch?v=SLjjgjp2bAA>, its not relating to cuda programming but the concept is the same.
* Sorting\_kern():
  + This function is a CUDA kernel that sorts the distances generated from the distance kernel, and outputs the distances and indices of the neighbors. The sorting algorithm used is advanced, but if you wanna see that it works simply use the code inside knn\_cuda() to check its output.
* Knn\_cuda():
  + This function allocates memory, assigns thread/block counts for each kernel, and transfers the memory from the GPU to our host memory, i.e. CPU, so that we can later perform statistics on the data.
* Main():
  + This function helps with breaking down the input data into digestable chunks that our GPU memory can handle. The function also performs statistical tests to determine the classification of our test data.

Memory Management

Now that you are somewhat familiar with the functions in the CUDA implementation of KNN, lets discuss how we will be breaking down our image data to allow the algorithm to run more efficiently. There are a couple of methods we can use to optimize memory in our algorithm: reduce the amount of data put into the KNN, processing subsets of our overall test dataset, and limiting host-to-device and device-to-host memory transfers.

By reducing the amount of information put into our KNN, computation time and the amount of memory necessary to run the algorithm is decreased. We consider every point in the test dataset to be independent from one another, an important assumption for this algorithm, we can use any domain we like when we put information into the KNN. (Note: there are implementations of KNN which consider test points as being dependent, but for this implementation of KNN we are safe to assume independence within the test dataset). This assumption allows us to choose any domain inside our image and still have the KNN run properly.

Reduce

For the examples in this document (see Appendix), I used a gaussian blur method to determine the high value areas of the image then used a basic threshold to generate the bit mask. The idea is to reduce the region of interest to only parts of the image that you consider to be relevant, and by reducing our region of interest we are able to reduce the amount of information the KNN has to process. However, by limiting the domain to only regions of interest we are also introducing bias into our processing, so make sure to control for the information you don’t consider by testing a subset of the “background” data to make sure your preprocessing step is valid. Essentially you want to make sure the KNN is labeling areas of the “background” as “not serotonin”.

Subset Management

We can also reduce the load on our GPU by testing small sections of our test dataset. I implement this exact procedure already in the main() function. I subdivide the test dataset into sections of 1000 points and run each 1000 test points through the KNN, generate their classification, then run the next 1000 points. The GPU I’m running right now is a Nvidia GTX 1050Ti mobile with 4GB of GRAM (i.e. memory), so the amount of data I’m able to process may not be the same as yours. Play around with the main() function to see the limitations of your GPU. You can also calculate the limitations of your GPU by using this inequality:

Float32 = 4 bytes

1 Gigabyte = 1e9 bytes

(# of test points)\*(# of training points)\*Float32 <= GRAM

Ex./ (1000)\*(691200)\*(4) = 2.764e9 <= 4e9 bytes

I have 1.235e9 bytes, or 446 test points I could add to the cycle, which would be more efficient then my 1000 (Note: running more points at once is more efficient as it reduces the amount of memory transfers from the host to the device, which takes up processing time). Disclaimer: this calculation does not take into account GPU memory being taken up by the display and background processes, so it won’t be exact!

Host-to-device transfers

This one is pretty simple and I’ve already given you a good way to counteract this problem previously, but I’m going to give you a small run down of why this problem exists and give you some resources on the underlying principles and how to implement all of this in python.

Host and Device

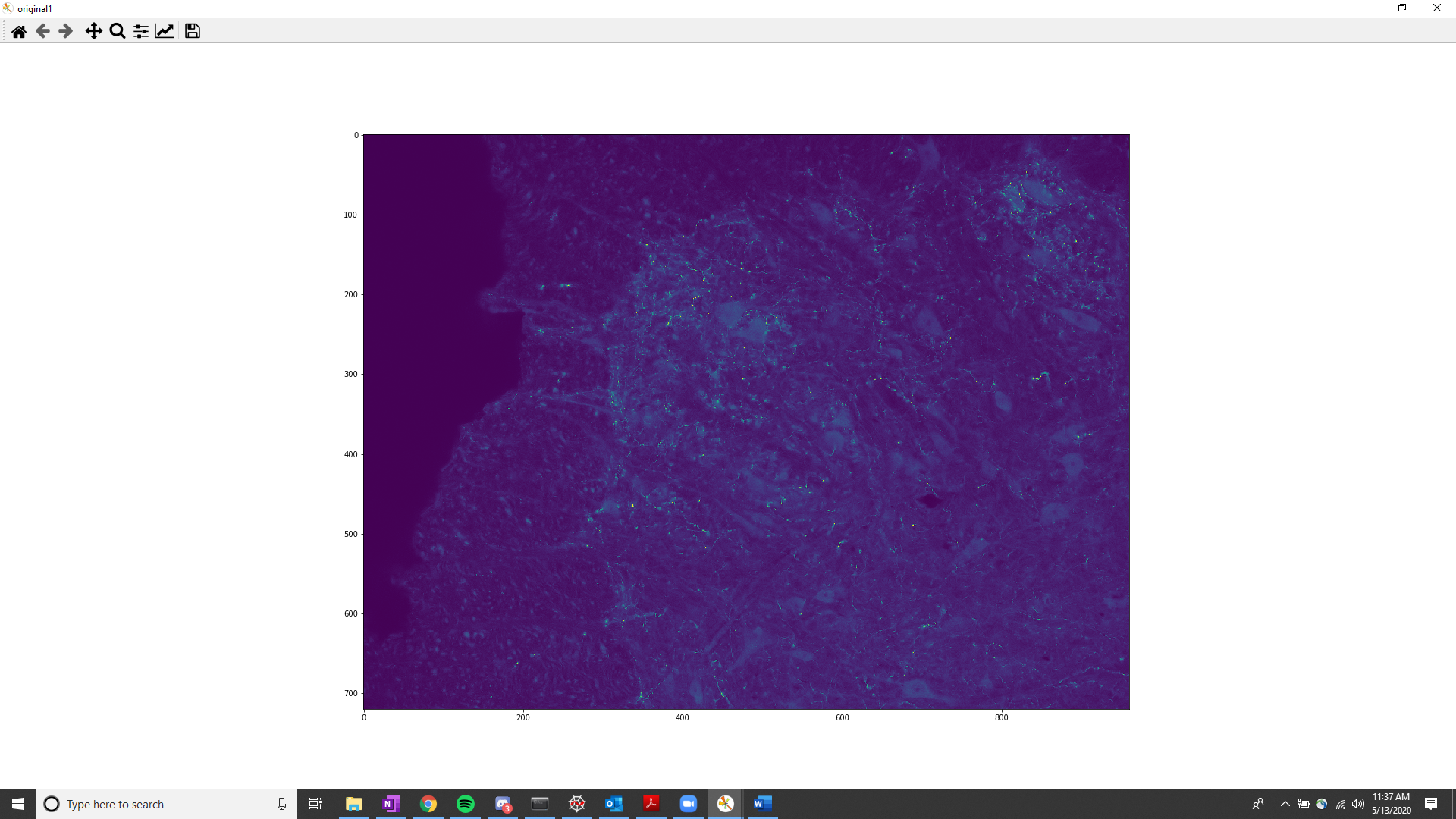
In simple terms: the **host** is anything integrated with the CPU, i.e. the on-board memory (RAM), while the **device** is anything integrated with the GPU. The language we use to describe these two parts of the computer becomes more intuitive when you think of the CPU as being the part of the computer that hosts or controls the other components in the computer. The GPU is a device the CPU uses for high intensity calculations. CPU is the brain, while the GPU is the calculator. Essentially, the CPU tells the GPU what to do and the GPU sends the results back to the CPU when its done. The main issue with this transfer is that it takes time to process, so limiting these transfers is important for reducing computation time.

Bugs and FAQ

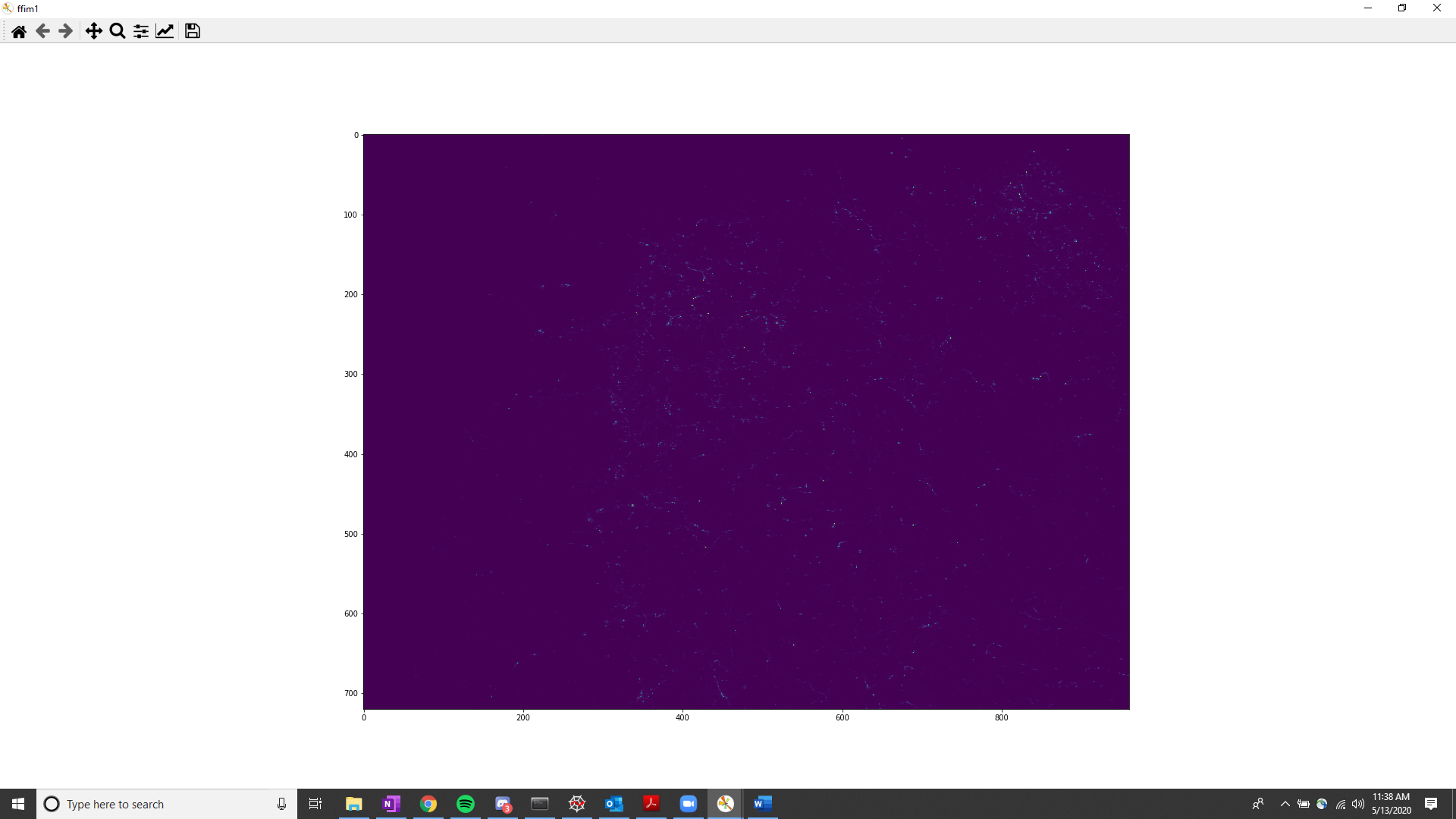
To be continued…

# Appendix:

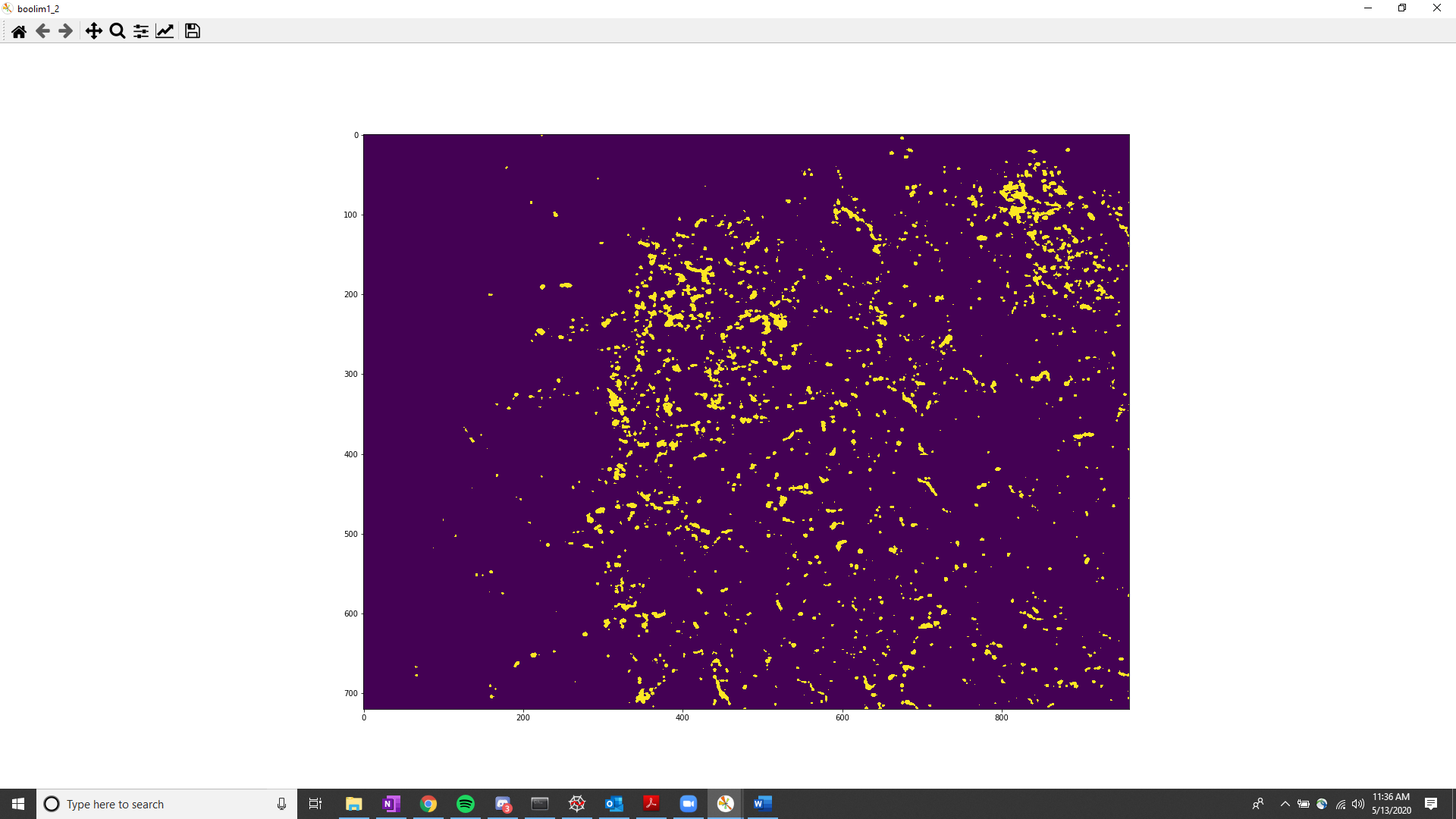
### Train Image: Original no processing



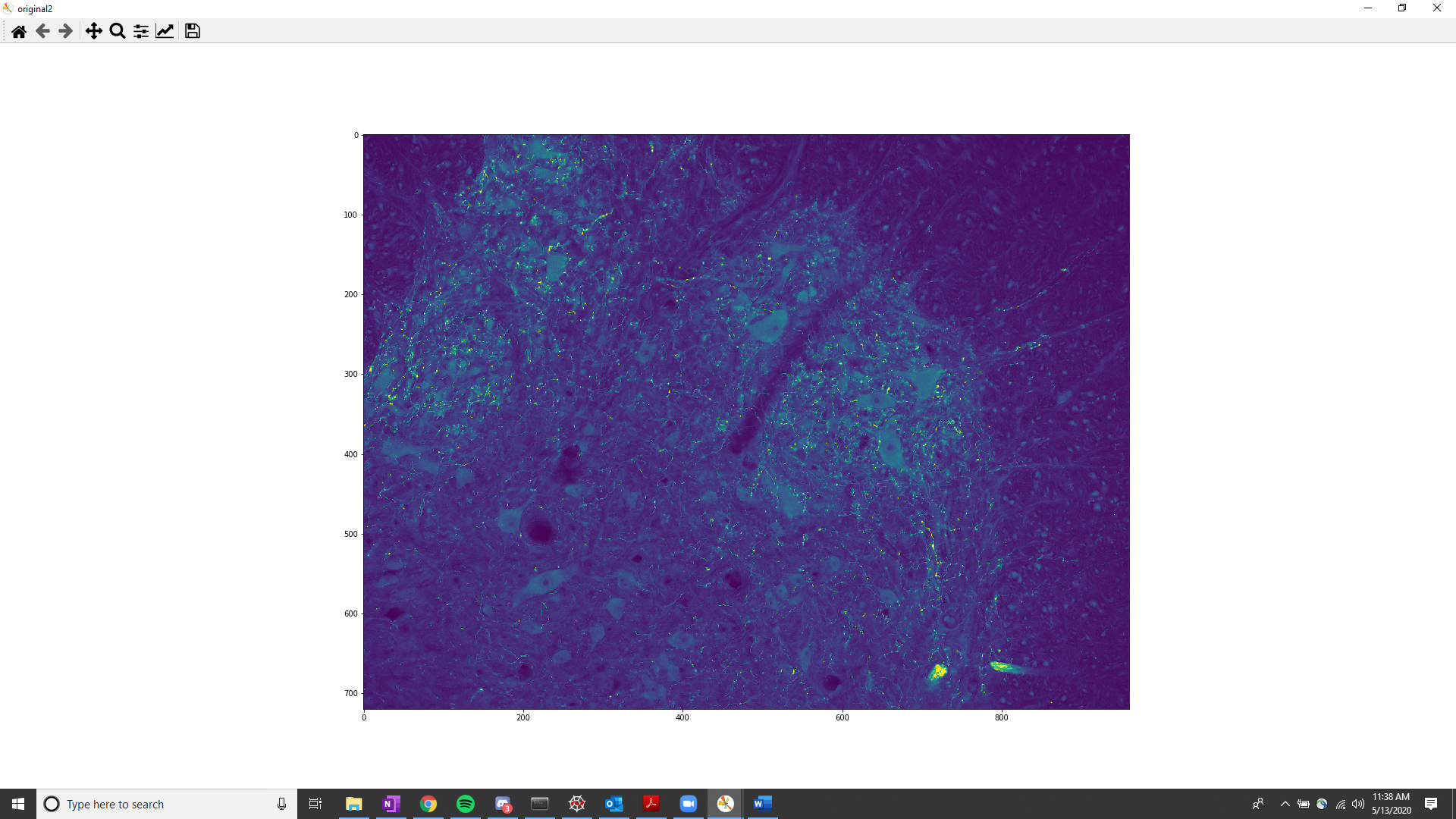
### Train Image: Filtered using high pass filtering



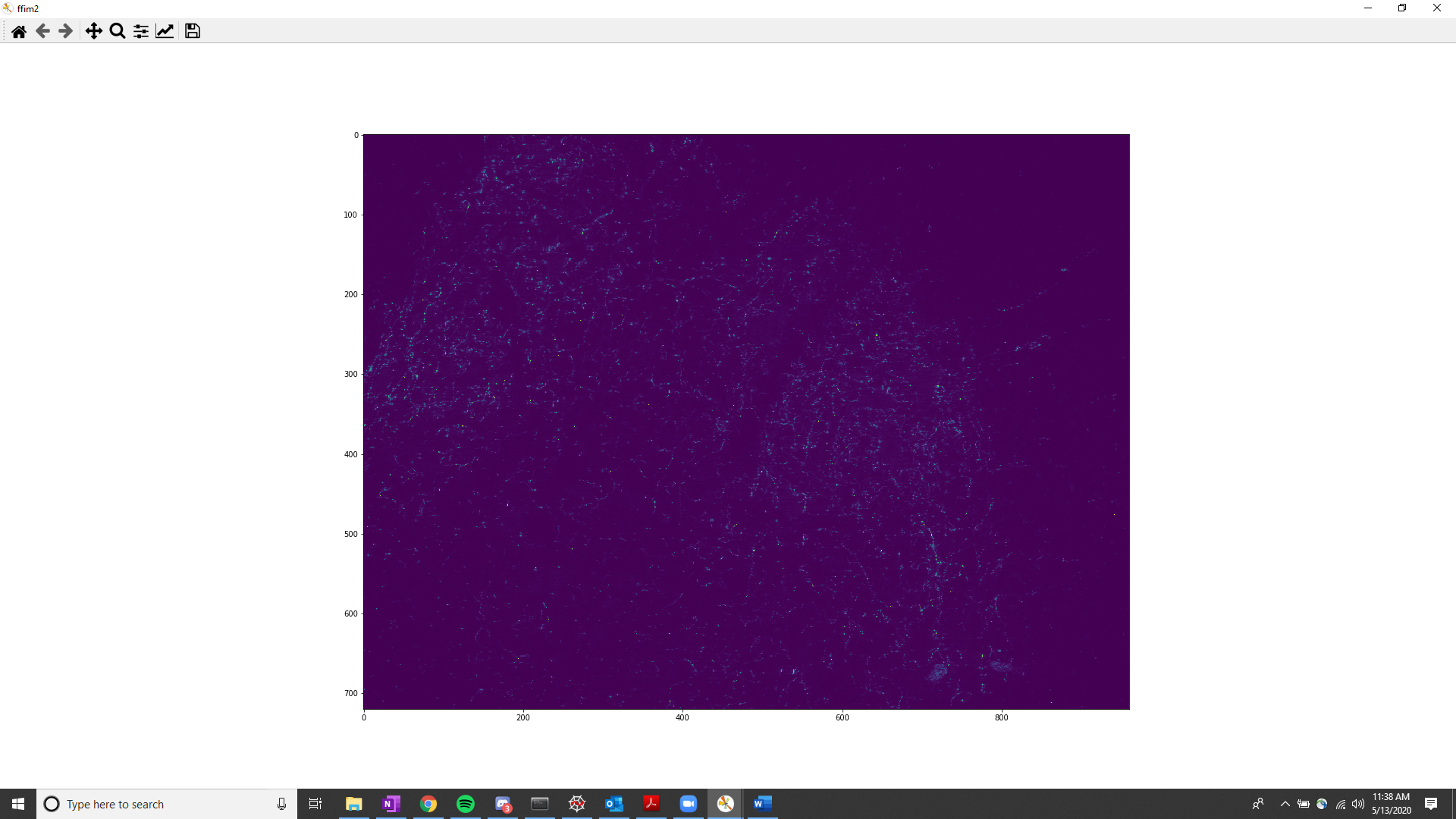
### Train Image: areas being considered as positively labeled serotonin



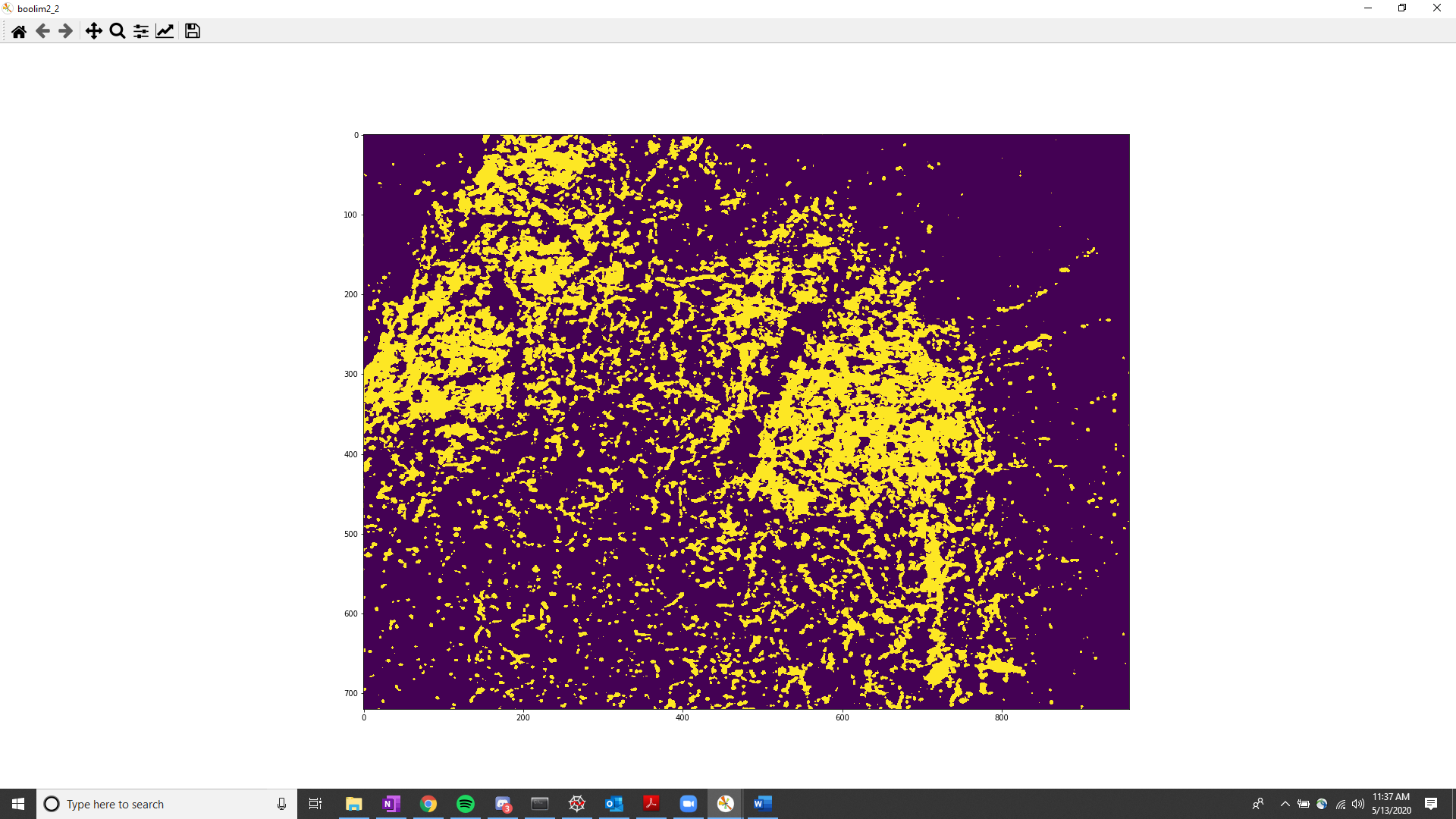
### Test image: Original, no processing



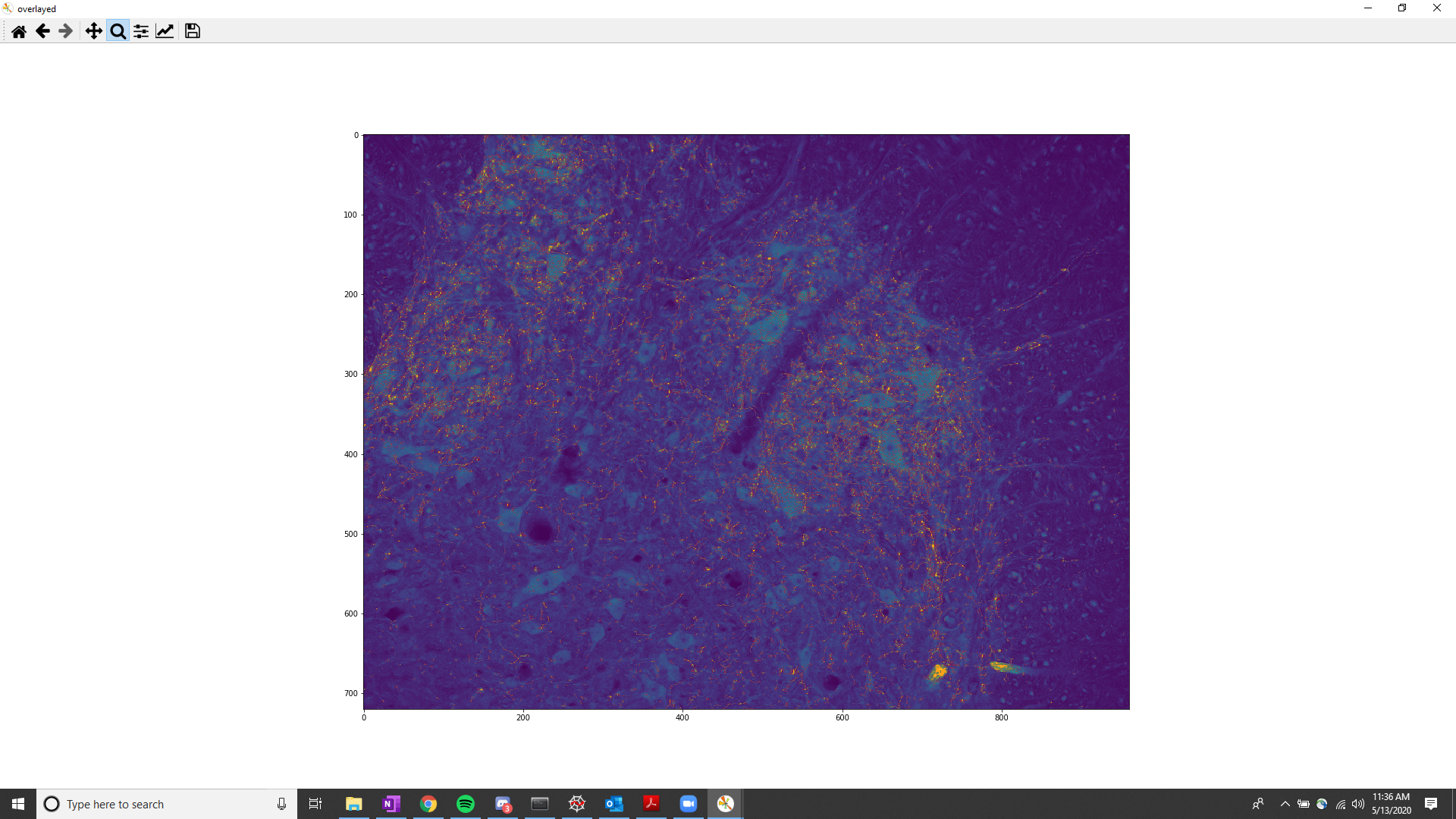
### Test image: Filtered using high pass filtering



### Test Image: test domain, points labeled in yellow will be “test” against the “train” data



### KNN Results: In red are the positively identified serotonin areas as outputted by the KNN.



### KNN Results: KNN results labeled in yellow.

