1 Introduction

This is a short summary of the work I did for 6 weeks as a summer intern with Conor Houghton. The topic of the internship was spike sorting, in particular to test an algorithm suggested by Dr. Houghton. I’ll first briefly describe spike sorting, at least my understanding of it.

A few small biological details (which I’m sure are a simplification) - a neuron fires ‘spikes’ of voltages; different types of neurons fire different spikes, but all of the spikes from any single neuron will be nearly identical. If we record the electrical activity in a part of the brain, we will get a signal that is composed of spikes from many different neurons. But what if we want to break this down into its component spikes, and identify which spikes come from which neurons? This is spike sorting. The problem is, given an arbitrary recording, we don’t know how many neurons are involved, or what their spikes look like.

2 The Greedy Gradient Descent Algorithm

This is the initial algorithm tested. It is very simple, but didn’t quite work as we’d hoped.

2.1 Description

We start out with some approximate spikes, which we refer to as our atom functions, $A_i$. In our case these were generally sines with varying wavelengths. Each atom function then is placed at various time indices with
some amplitude. So, if \( r(t) \) is our signal, then we have decomposed it into

\[
\tilde{r}_t = \sum_i \sum_{k \in C_i} \lambda_{i,k} A_{i,t-t_k}
\]

where \( \lambda_{i,k} \) is the amplitude of the \( k^{th} \) placing of the \( m^{th} \) atom function, and \( t_k \) is the corresponding time index. This decomposition is suggested by Smith and Lewicki\(^1\). The goal is to find the values of \( t_k \), \( \lambda \) and the exact atom functions \( a_{m,t} \).

The algorithm is as follows. We first place the initial approximate atom functions by a greedy algorithm. That is, we take the dot product of each atom with the signal at each possible time index. We find the maximum of these, and place that atom at that position. We then repeat this process until the highest dot product is below some threshold. Once we have this initial placing, we perform gradient descent on each of the atoms in turn. For this we need \( \frac{\partial \varepsilon}{\partial A_{j,t}} \), where \( \varepsilon \) is the error function given by

\[
\varepsilon = \sum_t \left( r_t - \tilde{r}_t \right)^2
\]

and \( r \) is the signal, \( \tilde{r} \) is our approximation. We find the value of this partial derivative in the next section.

We then take our optimized atoms, and begin again. We place them with the greedy algorithm, and then perform gradient descent once again. We repeat this until (hopefully) the error becomes very close to 0.

### 2.2 Partial Derivative Derivation

So we have

\[
\tilde{r}_t = \sum_i \sum_{k \in C_i} \lambda_{i,k} A_{i,t-t_k}
\]

So

\[
\frac{\partial \tilde{r}_t}{\partial A_{j,t'}} = \sum_i \sum_{k \in C_i} \lambda_{i,k} \frac{\partial A_{i,t-t_k}}{\partial A_{j,t'}}
\]

\[
= \sum_i \sum_{k \in C_i} \lambda_{i,k} \delta_{i,j} \delta_{t-t_k,t'}
\]

\[
= \sum_{k \in C_j} \lambda_{j,k} \delta_{t-t_k,t'}
\]

Now, \( \varepsilon = \sum_t (r_t - \tilde{r}_t)^2 \). So

\[
\frac{\partial \varepsilon}{\partial A_{j,t'}} = 2 \sum_t (r_t - \tilde{r}_t) \frac{\partial (-\tilde{r}_t)}{\partial A_{j,t'}} = -2 \sum_t (r_t - \tilde{r}_t) \sum_{k \in C_j} \lambda_{j,k} \delta_{t-t_k,t'}
\]

Let \( t'' = t - t_k \).

\[
\frac{\partial \varepsilon}{\partial A_{j,t'}} = -2 \sum_t (r_t - \tilde{r}_t) \sum_{k \in C_j} \lambda_{j,k} \delta_{t''-t',t'} = -2 \sum_{k \in C_j} \sum_t (r_t - \tilde{r}_t) \lambda_{j,k} \delta_{t''-t',t'}
\]

Now \( t'' = t' \Rightarrow t = t' + t_k \). So

\[
\frac{\partial \varepsilon}{\partial A_{j,t'}} = -2 \sum_{k \in C_j} (r_{t'+t_k} - \tilde{r}_{t'+t_k}) \lambda_{j,k} = -2 \sum_{k \in C_j} e(t' + t_k) \lambda_{j,k}
\]

where \( e(t) = r_t - \tilde{r}_t \).

### 2.3 Implementation Details

The signal used in the testing was generated as follows. There were two types of spikes:

\[
\begin{align*}
5\sqrt{1 - 10t} \sin(20\pi t), & \quad t \in [0, 0.1] \\
3(1 - 40t^2) \sin(40\pi t), & \quad t \in [0, 0.05]
\end{align*}
\]

These were placed randomly at the rate of 30 per second. Noise was added in the form of 100 sine parts (with wavelength 0.25), with amplitudes in \([0, 0.03]\).

Once the signal was generated, the algorithm described above is run. The initial atoms used were sine parts with wavelengths were 0.05 and 0.1.
Various values of the step-size of the gradient descent were used, and left running for generally around 6 hours. Graphs of the error were produced, along with the positions of the spikes placed by the algorithm, which were compared against the known true locations of the spikes in the signal.

2.4 Results

The error did generally decrease, the graph looking something like that of $\frac{1}{x}$. There were occasional ‘jumps’, corresponding to replacements that slightly increased the error (as the greedy algorithm doesn’t guarantee optimal placement), followed by periods of decline (from the gradient descent). However the error levelled out (still decreasing, but very slowly) well before the correct placement was found. This suggests that the algorithm was converging to a local minimum. If the signal was made more sparse (so that there were little or no overlapping spikes), the algorithm does work. While this isn’t particularly useful, it at least suggests that the code was working as it should.

There are a few possible reasons for this failure. The first is the problem of destructive interference of two spikes. Let’s say we have two spikes placed so that the downward peak of the first spike coincides with the upward peak of the second. Now the greedy algorithm may place neither spike, as the dot product is substantially reduced and could be below the threshold. Another possible problem with the algorithm is that it is too dependent on the initial placing. Once the atoms have been placed, we optimize them for that particular position. This might make the atoms look even less like the actual spikes. Then they’re placed in an even worse position, etc. Finally, we probably shouldn’t have the amplitude coefficient - every atom from a given neuron should have the same amplitude.

2.5 Slight Variants

Some variants of the above approach were attempted. Firstly, the amplitude coefficients were removed. This also meant that the atoms were no longer normalised (previously all atoms had been normalised so that the dot product comparison between atoms made sense, and then the amplitude coefficient allowed the atoms to be as large as we wanted). This approach didn’t work. The error still levelled out, and then suddenly exploded, possibly due to the removal of the normalisation.
The next thing we tried was to change the gradient descent to something simpler and cruder - to take the error at each point, and distribute it over the atoms placed at that point. This didn’t work - the atoms got very ‘jittery’.

2.6 Other Ideas

Here are some ideas that weren’t really investigated, but which I think (possibly naively!) might be worth looking at:

- As a possible way to overcome the ‘destructive interference’ problem: for each atom and each time index, place that atom at that time index, then do the greedy algorithm, taking the error at the end. Then compare each of these with the error if we just perform the greedy algorithm, say $\epsilon$. The peaks at which the error is lower than $\epsilon$ are likely to correspond to places where spikes should be placed, but weren’t by the greedy algorithm (possibly due to destructive interference). However, this depends on the atoms looking quite like the true spikes.

- Assume that the wavelength of the atom is $w$, and then take a sine wave and scale it to have wavelength $w$. Take the dot product of this sine wave at every point in the signal. The resulting graph has clear peaks, which (if the wavelength was chosen correctly) correspond to the places where the spikes should be placed. However, this requires knowing what $w$ is, and assumes that the wavelength of each type of spike is different. It might be possible to work around the first problem by making graphs for a lot of values of $w$ and observing around which points peaks tend to stay (but I’m not really sure if this makes any sense and I couldn’t get it to work).

3 The Genetic Algorithm

At this point it seemed that the Greedy Gradient Descent algorithm wasn’t working, so a different approach was tried, one based on the genetic algorithm.
3.1 Description

We start out with an initial approximation of the signal (a ‘gene’) - consisting of atoms, and their corresponding spike trains. We mutate this initial gene in various ways, to create a ‘population’ of genes. Mutations consist of small changes to the atoms, changes to the spike trains (adding, moving or removing spikes) and possibly performing gradient descent on some of the atoms.

Then the population is sorted by error. We then delete most of the genes (weighted towards those with greater error), and repopulate by reproducing the remaining genes. Possible reproductions are splicing together two atoms, or two spike trains. At this point we sort the genes again, etc. We continue to do this for a large number of iterations.

3.2 First Implementation

The signal was generated as above. And again, a greedy algorithm is used to find the initial placement of the atoms and the initial amplitudes. Then we mutate this gene 400 times to give us the initial population. The following mutations were implemented:

- Randomly pick a single spike from a spike train, and move it to another position in that spike train (in such a way that it isn’t placed too closely to a previous spike). Then with probability $\frac{1}{4}$ do this again.
- Remove a spike, and then with probability $\frac{1}{4}$ repeat this.
- Add a spike, and then with probability $\frac{1}{4}$ repeat this.
- Choose an atom, and move a single point on this atom randomly.
- Choose an atom and perform gradient descent with a small step size.

A mutation consists of one of these, chosen randomly. The first four are equally likely, and the last mutation is only rarely performed.

The genes are then sorted. The first 60 genes are kept, and then the remaining genes are deleted with probability $\frac{1}{2}$. For each deleted gene, we take two genes randomly (which are chosen from the first 60 with probability .6, and from the entire population otherwise) and reproduce these. We then mutate the resulting gene. The following reproductions were implemented:
- Take the $i^{th}$ spike train from each gene, and take a random time index. Take the first gene and splice in the spike train from the second gene, after that time index.

- Take the $i^{th}$ atom from each gene, and take a random time index. Take the first gene and splice in the atom from the second gene, after that time index.

These are performed with equal probability.

### 3.3 Results

Various different values for each of the above probabilities and population sizes were attempted. However, the results were more or less the same - the error would decrease quite rapidly, and then would steady out, in the same way as the Greedy Gradient Descent algorithm. However, it levelled out with a much lower error - about 6 times lower. The lowest error achieved was around 150 - which corresponds to about an average error of 0.39 at each time index (which is roughly $\frac{1}{10}$ of the signal). The placing was also nearly exactly correct. However, even if left running for several hours after correct placing had been found, the error did not decrease further. This indicated that the atoms stopped approaching the true spikes at this point.

In an attempt to fix this, gradient descent was applied aggressively once every few hours. This in fact slowed convergence at first, and then once the signal reached the same point (an error of around 150) had very little effect. Examining the amplitudes revealed why - although the placing was correct, the amplitudes were far too varied. So this too was also a local minimum, even though the placing was correct. So, the amplitudes were removed, and the code was again run (for about ten hours). This worked - the error signal reduced to 7 (about 0.08, or $\frac{1}{25}$ of the signal). The spikes were placed nearly completely correctly, and the atoms were very close to the true spikes. The remaining error was due in part to noise, and in part to the fact that the spikes were placed a little too late in each case, and were correspondingly shifted slightly to the right, which lead to error (as a small part of the spike was missing on the far right).
3.4 Second Implementation

After the success of the very artificial test described above, Dr. Houghton suggested testing the algorithm against a signal generated in a more realistic way. Firstly, we place neurons randomly around the origin with some density. For each neuron we assign an amplitude, which is inversely proportional to $r^2$, where $r$ is the distance from the origin. Then each neuron is assigned a spike template, which varies from one of two types in peak size and wavelength by 10% (the first type, which is long and flat, is chosen with probability 0.8 and the second type, which is short and steep, with probability 0.2). The actual spike is determined by 3 parameters - the size of the first peak, the size of the second peak and the wavelength (it is created by taking a sine wave, then stretching and skewing parts of it to create a smooth looking spike with the appropriate properties). Then each neuron fired as independent poisson processes. The far out atoms acted as noise, and the first few (3 in the most tested case) were large enough to hope that the algorithm might pick out.

The actual algorithm was nearly identical to the previous code (the version without amplitudes). One change was that instead of mutating atoms by randomly changing individual points, we mutated them by denting the atom slightly. The hope was that this would ensure that spikes continued to look like spikes, as the old mutation had the tendency to at times reduce atoms to what looked like scatter plots.

3.5 Results

Unfortunately, this wasn’t as successful as the first implementation. As with all previous tests, the error at first reduced rapidly before leveling off. Unfortunately, it leveled off with a very high error. I tried varying the density and random seed so that the wavelengths of the large spikes in the signal matched the wavelengths of the initial atoms, in order to give the algorithm a slight advantage in the initial placement. However, even in this case the error remained large, with the atoms seemingly approximating the same spike for some reason.

3.6 What’s Next?

My 6 weeks was up before I could properly investigate the results above. In my opinion, the most likely cause of the problem is the way the initial
population is created. The genetic algorithm (in my understanding), works so well because of its ability to test points spread out all over the sample space. This makes it less prone to getting stuck in a local minimum. And yet, in the algorithm above the initial points chosen are all clustered around a single point! Looking at it this way, I don’t think it’s surprising that it didn’t really work.

I think the next step is to remove the initial greedy algorithm placement, and instead make the initial placement completely randomly. This should converge more slowly, but hopefully to the correct placement.

Also, more reproductions and mutations should be added - most importantly ones that add/remove atoms. This isn’t the cause of the problem above (the reason why I tweaked the number of atoms in the signal to match the number of atoms per gene was to make sure of this), but if the algorithm is ever to work in general, I think it’s needed. One problem is that if unlimited atoms are allowed, it will just end up making a new atom for each section of the signal (reducing the error to 0 in a meaningless way).