CS 267: Application of Parallel Computers Due: 12 May 2010

Final Project: Parallel unsupervised learning with neural data

Amir Khosrowshahi SID: 18208866

Abstract

This note is an accompaniment to my poster for my class project. Despite its seeming length, it attempts to be as brief as is reasonable, explaining the figures in greater detail and providing additional computational details most relevant to the subject of the course.

1 Introduction and description of data

A relatively recent and exciting development in the field of neuroscience are the manifold new methods for recording simultaneously from large populations of neurons in cortex. I have been working with data collected using two such methods, silicon polytrodes and chronically-implanted electrode arrays. Analysis of this data using an unsupervised machine learning algorithm is the subject of my project.

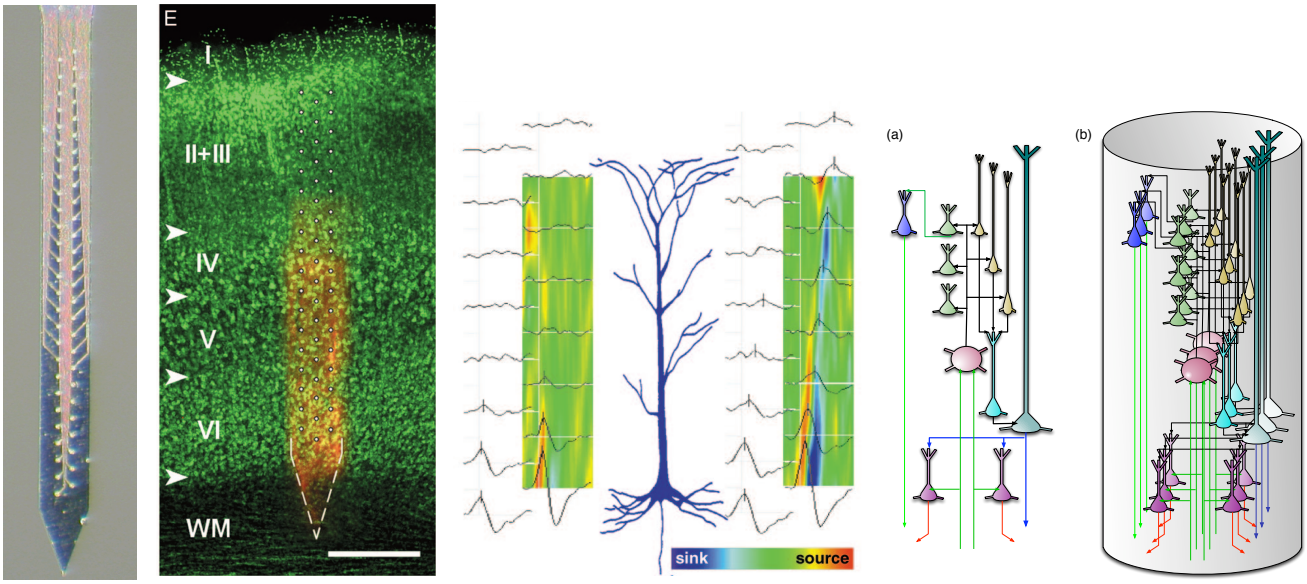
A silicon polytrode [1], depicted in Fig. 1a, has an array of etchings along one surface that allows for recording the extracellular electric potential of nearby neurons. Inserted perpendicularly to the cortical surface, it spans an entire vertical *column* and enables recordings from on the order of one hundred neurons simultaneously. For much of the history of electrophysiology, recordings have been done painstakingly from a single neuron at a time by moving a sharp electrode close enough to the cell body of a frequently spiking neuron. A neuron's activity is encoded in large part by the changes in its membrane potential, characterized by infrequent and punctate *spikes* that manifest as characteristic potential waveforms on the recording electrode.

The ability to record from a vertical span of cortex from such a high-density electrode provides an unprecedented amount of data that will allow us to address several outstanding hypotheses regarding the computational function of a local neighborhood of neurons. A patch of cortex as depicted in Fig. 1a looks like most any other in the same cortex or across a species and even across related species. The anatomy and biology of a cortical patch seems to be widely recapitulated. Is it possible that this canonical microcircuit has a set of common computational functions (Fig. 1c)?

There are a number of downsides to data collected from silicon polytrodes. Primary among them is that the signal is noisier than single electrode recordings. One challenge is to be able to identify when a single neuron is active, a process called *spike sorting*. Neurons typically have an elaborate dendritic morphology with an electric potential that varies in a complex manner in time along its span (Fig. 1b). A sample recording from visual cortex is depicted in Fig. 2. The goal of this project is to use the sparse pattern of activity registered on multiple nearby electrodes to infer the causes of these signals, which in this case are, in most part, the spiking activity of a population of neurons.

Another dominant characteristic of cortical activity is ongoing oscillations in the frequency band less than 150Hz. These oscillations as recorded from a polytrode are depicted in Fig. 3. The statistics of this signal is quite different from the spiking data depicted in Fig. 2. Though the pattern of activity is largely coherent across cortical lamina, it is likely that the phase differences have correlates to particular computations that are being performed. Another goal of this project is to uncover characteristic patterns that recur in this data which will be a first step in understanding what these signals mean and what patterns appear with what frequency.

In order to characterize the activity in a wider region of cortex, we are also using an array of electrodes that are chronically implanted over several visual areas (Fig. 4). The distances between electrodes is on the order of millimeters as compared to tens of microns in the case of polytrodes. A sample lowpass recording signal is depicted in Fig. 5. Recordings are typically made over the course of a month.



(a) Silicon polytrode (b) Pyramidal neuron morphology (c) Canonical microcircuit

Figure 1: (a) At left is the single-shank 54-channel alternating arrangement of recording sites on the silicon polytrode used in our experiments. Inserted perpendicularly to the surface of the brain, it spans all layers of cortex. The green dots represent stained neurons with a similar device in relief [1]. (b) A typical pyramidal cell has an elaborate dendritic morphology with a complex and changing potential signature. To the left, a typical action potential is represented, whereas at right is a putative back-propagating action potential generated from the same neuron [1]. (a) One of many hypothesized functional diagrams of this local circuit of neurons, whose function is almost completely unknown [2].

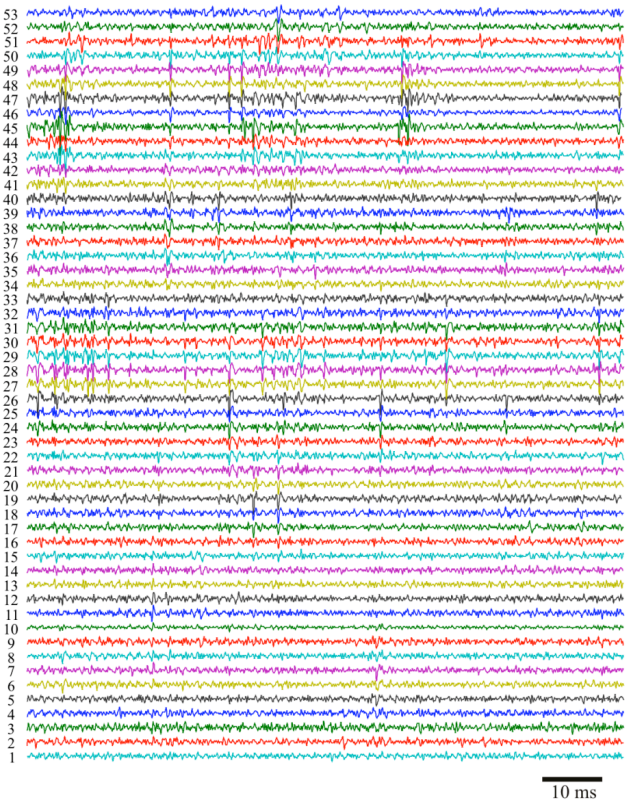


Figure 2: A sample polytrode recording from visual cortex, filtered between 500Hz and 10kHz, shows bursts of activity that are localized in time and along the length of the polytrode. The spike of a single neuron is likely to register on several recording sites and one recording site will likely register the extracellular potential of a number of nearby neurons.

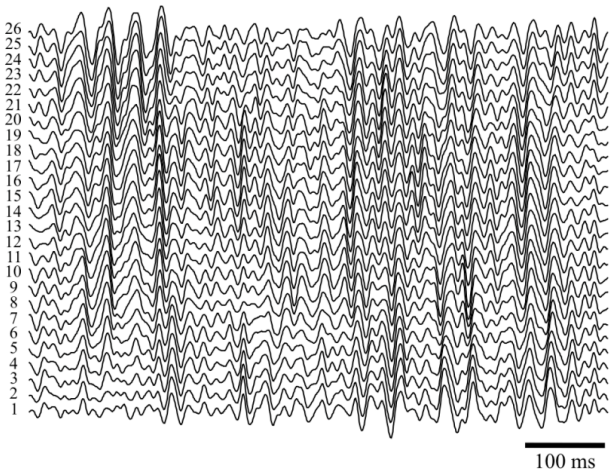
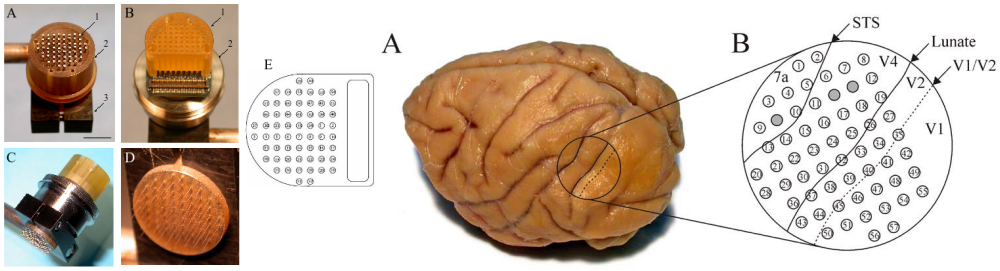


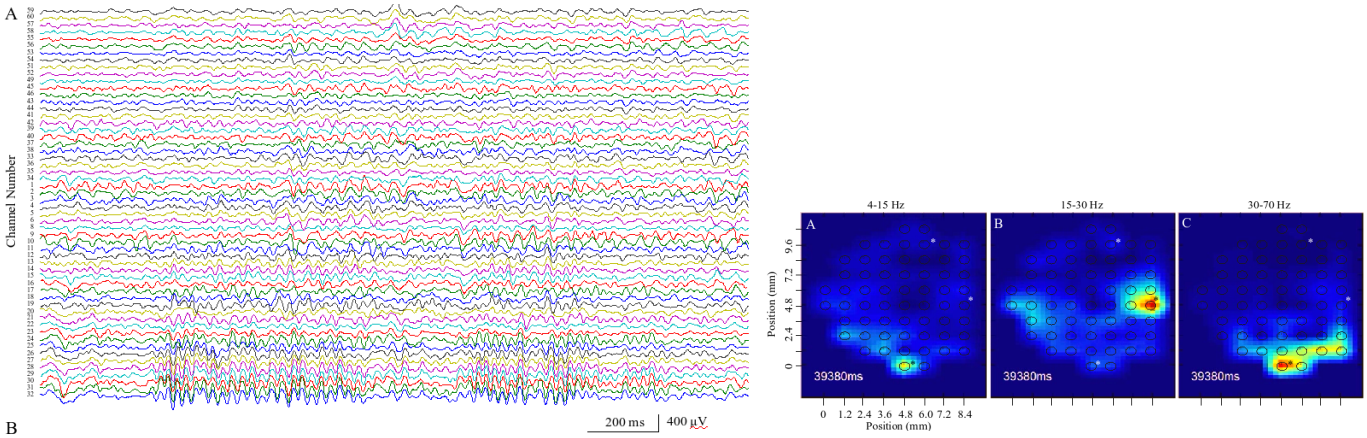
Figure 3: A sample polytrode recording from visual cortex, filtered between 0-150kHz (called the local field potential, or LFP), shows a burst of activity in the *gamma* band. These oscillations, which are thought to represent the summed synaptic activity of a population of neurons, have been associated with many functions such as level of attention. But in most part, their significance is unknown.



(a) Recording array (b) Visual areas

Figure 4: (a) A 64-channel array of individually adjustable electrodes are implanted in cortex. During periods of recording, a protective cover cap is removed from the device and a recording apparatus is attached. (b) The particular recordings used in my project are over the visual areas V1, V2, V4 and a non-visual area, STS.

.



(a) LFP (b) Visual areas

Figure 5: (a) Sample LFP from a chronic array shows markedly different activity in different regions of cortex. (b) Simultaneous activity in different frequency bands segregate spatially across the array.

.

2 Model

2.1 Linear generative model

A first step at a simple representation of the statistical structure of these different data sets is to use a linear generative model. Given a set of data samples $\{\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_D\}$ with $\mathbf{y} \sim p(\mathbf{y}) \in \mathbb{R}^M$, we can propose the following linear model,

$$\mathbf{y} = \mathbf{\Phi}\,\mathbf{s} + \epsilon$$

where $\mathbf{\Phi} \in \mathbb{R}^{M \times N}$ is a *basis* or signal dictionary, $\mathbf{s} \in \mathbb{R}^N$ are *coefficients* and $\epsilon \sim \mathcal{N}(0, \sigma_n^2 I)$ is small Gaussian noise. The goal is to find an appropriate basis $\mathbf{\Phi}$ and a set of coefficients $\{\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_D\}$ corresponding to the data. We have that,

$$p(\mathbf{y}|\mathbf{s}) \propto e^{\frac{-\|\mathbf{y}-\mathbf{\Phi}\mathbf{s}\|^2}{2\sigma_n^2}}$$

We can impose additional structure on the coefficients $\mathbf{s}$. For reasons that will become clear, we choose the coefficients to be sparse and factorial (conditionally independent) [3]. To start, we impose a Laplacian prior,

$$p(\mathbf{s}) \propto e^{-\lambda \sum_i |s_i|}$$

with sparsity parameter λ and we choose $N \gg M$ so that $\mathbf{\Phi}$ is a highly *overcomplete* basis. We can choose the mode of posterior to estimate,

$$\mathbf{s}^* = \arg\max_{\mathbf{s}} p(\mathbf{s}|\mathbf{y}, \mathbf{\Phi}) \approx \arg\min_{\mathbf{s}} \frac{1}{2\sigma_n^2}\|\mathbf{y} - \Phi\mathbf{s}\|_2^2 + \lambda\|\mathbf{s}\|_1$$

This convex objective, a quadratic loss with an L_1 regularizer has been intensely studied in the signal processing and statistics communities and goes by various names such as the Lasso [4] and Basis Pursuit Denoising [5] and there are a number of fast methods to find a minimum. Due to special properties of the L_1 norm, that the Lasso solution is sparse, with many components of the coefficients being exactly zero (see Fig. 6). I used a modification of one such method, orthant-wise l-BFGS [6], in this project. Given an estimate of $\mathbf{s}$, the objective is then convex in $\mathbf{\Phi}$ and a minimum of the loss in terms of the basis can be found. A possible overall strategy is to randomly initialize the basis $\mathbf{\Phi}$, receive online batches of data $\mathbf{y}$, *infer* coefficients $\mathbf{s}$ via Lasso and do a *learning* update of Φ [7]. This online alternative minimization procedure can be written in matrix form as,

$$\arg\min_{\Phi, S} \|Y - \Phi S\|_F^2 + \lambda\|S\|_1$$

There is a degenerate solution with S going to zero and $\mathbf{\Phi}$ unbounded. We therefore choose the columns of $\mathbf{\Phi}$ to be of at most unit norm, that is $\sum_j \Phi_{ij}^2 \leq 1$. This added complication transforms the learning step into a quadratically-constrained quadratic program (QCQP). The main goal of the algorithm is to find, in expectation, the basis $\mathbf{\Phi}$ that gives the smallest L_1-regularized error over the whole dataset.

2.2 Sparse convolutional basis

In the case of the electrophysiology data, the data $\mathbf{y}$ consists of time series of electric potentials on a set of M channels. We express the data as a linear combination of convolutions of a set of basis elements $\mathbf{\Phi}$ with coefficients $\mathbf{s}$. We again force the coefficients to be sparse. This computation is illustrated for one channel in Fig. 7. For a random sampling of the electrophyiology data of length T, we can write an objective as,

$$E(s, \phi) = \frac{1}{2Q}\sum_{qmt}(y_{mt}^q - \sum_{np}\phi_{mnp}s_{n,t+p}^q)^2 + \lambda\sum_{qnt}|s_{nt}^q|$$

where the indices $t \in \{1 \ldots T\}$, $m \in \{1 \ldots M\}$, $n \in \{1 \ldots N\}$, $p \in \{1 \ldots P\}$, Q is the number of batch elements, P is the time length of the convolutional kernels, $P \ll T$, and N is the number of such kernels. In this project N is picked to be on the order of the number of channels M. This is still a linear generative

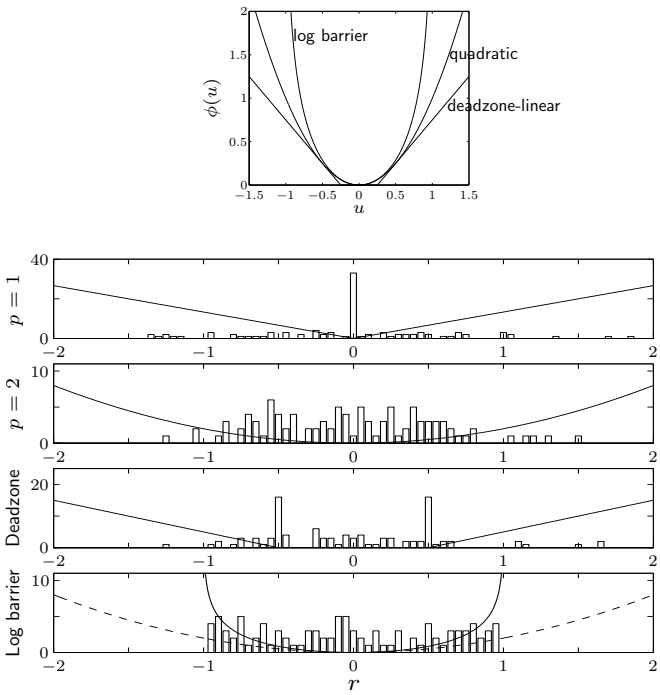


Figure 6: A variety of regularizers ϕ are displayed at top. The distribution of coefficients for sample random data is shown below. Note the accumulation of points at the origin for the $p = 1$ case. The $p = 2$, know as Ridge Regression, produces coefficients that are less likely to be very large but are not concentrated at the origin. The last and possibly the most widely used regularizer is the log-barrier, which is the main component of interior point solvers for conic programs. Figure taken from [8].

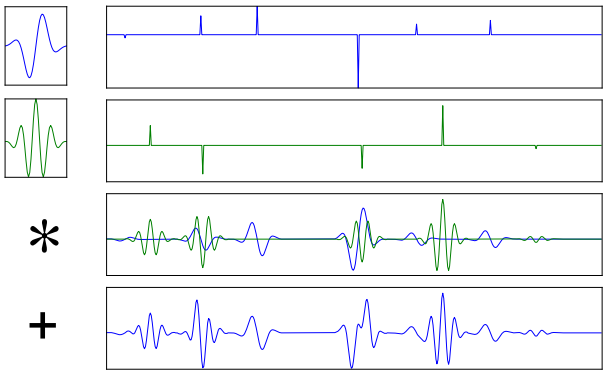
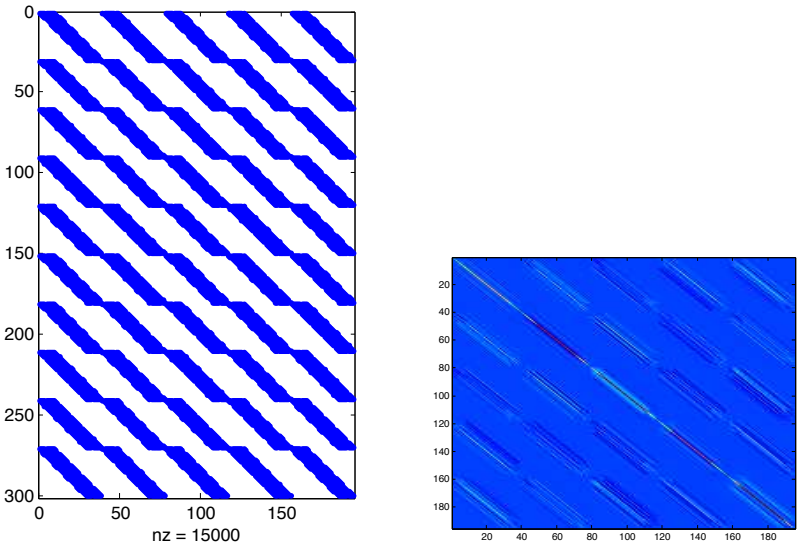


Figure 7: A schematic of how to reconstruct a portion of one channel of data at bottom using two basis elements at left convolved with corresponding sparse coefficients at right and summed. We would like to infer the coefficients and learn the kernels.



(a) $\mathbf{\Phi}$ (b) $\mathbf{\Phi^T\Phi}$

Figure 8: (a) A Matlab `spy` plot of a toy Φ with $M = N/2$. (b) The structure of the Hessian $H = \Phi^T\Phi$. A component of orthant-wise l-BFGS is maintaining a positive definite estimate of H^{-1}. My implementation of l-BFGS allows this inverse Hessian to be unrolled to compare with the actual Hessian. In practice, the l-BFGS Hessian, which is essentially the identity matrix plus a small number of vector outer products, is more concentrated on the diagonals.

model as presented above where $\mathbf{\Phi}$ is a block circulant matrix as depicted in Fig. 8a. The algorithm I used in this case (Algorithm 1) is to use orthant-wise l-BFGS in the inference step and an accumulative projected gradient step in the learning step. This is a modification of a recent publication [9] where a block coordinate-wise descent is used for the learning step. There are several key items to keep in mind regarding this strategy:

- Orthant-wise l-BFGS [6], though not the fastest Lasso method, is advantageous for several reasons. Unlike other Lasso methods, it works for a general convex and differentiable loss, not just a Gaussian squared error. I am also using this l-BFGS implementation for an accompanying project where I use other members of the exponential family of distributions for the loss. It uses little memory and scales well to large problem sizes. One outstanding theoretical issue that has not been addressed to my knowledge is that the true inverse Hessian becomes increasingly poorly conditioned as the learning progresses, even when the basis is highly overcomplete. l-BFGS effectively regularizes this Hessian so that the inference step does not suffer from poor conditioning and the algorithm can converge faster. To use other Lasso methods in this convolution setting, significant work is required to implement all relevant BLAS operations using a custom compressed format for the block circulant $\mathbf{\Phi}$. Using a standard sparse format like CSR is impractical. This is a potential area for further study.

- Though the inference (Lasso) step is convex, the L_1 regularizer is not strictly convex and the Hessian $\mathbf{\Phi}^T\mathbf{\Phi}$ could effectively be less than full-rank. Therefore, though the objective has a global minimum, it is possibly not unique, which introduces possible interpretation issues of the coefficients $\mathbf{s}$. There are theoretical conditions on $\mathbf{\Phi}$ to have a unique minimum, such as a restricted isometry, incoherence, or restricted eigenvalue property that is actively being researched in the compressed sensing community. In practice, it is impossible to determine if these conditions are satisfied given an unknown underlying degree of sparsity. In the convolutional sparse inference case, I suspect that this condition will almost surely be violated. This is a key theoretical question that must be answered at some point. Given a

ground-truth basis with certain properties, if you generate data with a given sparsity, can this algorithm recover the basis? In the literature, either a custom basis, such as a wavelet, is given or the basis is assumed to be suitably random.

- The proposed algorithm separates the quadratic error into two terms that are a function of $\mathbf{\Phi}$, A and B. A consists of outer products of the coefficients and B consists of outer products between coefficients and the data. A decay term μ allows past A and B to be weighted so that the effect of learning prior $\mathbf{s}$ with incorrect $\mathbf{\Phi}$ can be forgotten.

- A and B are initialized in such a way as to minimmize large steps in the first few iterations by effectively reducing the step size.

- The projected gradient step [10] is quite slow to converge. The coordinate-wise descent method used in [9] is fast in practice but would require computing a matrix inverse in the convolution case (in step 6 of Algorithm 1). Since $\mathbf{\Phi}$ changes slowly each iteration, these inverses could be warm-started and computed using a symmetric iterative solver. An attractive feature of coordinate descent is that it is easily parallelizable and can be proven to converge. I am working on adapting my method to use such a scheme. The truncated method I use, running projected gradient for only a few steps, works well in practice. Other methods for QCQP such as interior point methods using truncated Newton's method may be appropriate (e.g. [11]). Less trivial convex constraints could also be imposed on the basis elements. Such high dimensional projections would be an interesting area to explore for parallel implementation.

- The most successful implementation of this learning algorithm is by a student in my lab, Jack Culpepper, that uses a generic quasi-Newton solver for Lasso, ignoring the non-differentiability at the origin and using a simple stochastic learning update. We are actively working together to incorporate elements of both strategies. Simple annealling stochastic gradient methods have attractive convergence properties and can often significantly outperfom more sophisticated methods particularly in an online setting where the dataset is practically infinite in size.

- A number of Lasso methods have potentially easy parallel extensions. One example is Pathwise Coordinate Descent [12] which could be effectively parallelized per coefficient element. However, this algorithm has some undesirable properties such as dependence of the solution on the order of the coordinates used in descent. Also one aspect of the algorithm depends on knowing that other coefficients are currently at zero to avoid an expensive computation. This part would be difficult to implement in parallel. However, one group has done so on a GPU with modest reported speed-ups [13].

- Lastly, and perhaps the most important computational property of the objective is that the Lasso is the most expensive part of each step and the inference for each element of a batch can be trivially performed in parallel. This is done via MPI in this project.

Two import instances of prior work in learning such convolutional kernels is by a collaborator [14] and my advisor [7], though the underlying methods used are different.

3 Results

I ran the convolutional sparse coding learning algorithm described above on several datasets. The first dataset was patches taken from a 15K frame natural movie where each frame was processed by a whitening filter, effectively removing 2-nd order statistics per frame to speed up learning. (This preprocessing step alone would have been too computationally expensive to be worthwhile without the use of CUDA FFT2's). A subset of the learned basis is displayed in Fig. 9. The statistics of natural scenes are dominated by moving edges and I thought this would be a good test that the algorithm is giving an expected result. The convolution kernels consist of moving bars of various orientations and spatial and temporal frequencies.

Initialization: $\phi_{mnp} \sim \mathcal{N}(0,1)$, $A_{\alpha\beta mn} = \varepsilon I_{\alpha\beta \times mn}$, $B_{mnp} \sim \mathcal{N}(0,\varepsilon)$

for $i \leftarrow 1$ **to** D **do**

Sample a Q-sized batch $\mathbf{y} \sim p(\mathbf{y})$ from data.

Using orthant-wise l-BFGS, calculate:

$$\hat{s} = \arg\min_{s} E(s, \phi)$$

$A_{np\beta\gamma} \leftarrow \mu A_{np\beta\gamma} + \frac{1}{Q} \sum_{qt} s^q_{\beta,t+\gamma} s^q_{n,t+p}$

$B_{\alpha\beta\gamma} \leftarrow \mu B_{\alpha\beta\gamma} + \frac{1}{Q} \sum_{qt} y^q_{\alpha t} s^q_{b,t+\gamma}$

Using truncated projected gradient descent with $\sum_{mp} \phi^2_{mnp} \leq 1$, calculate:

$$\hat{\phi} = \arg\min_{\phi} E(\phi; A, B)$$

end

Algorithm 1: A schematic of the sparse convolutional basis learning algorithm

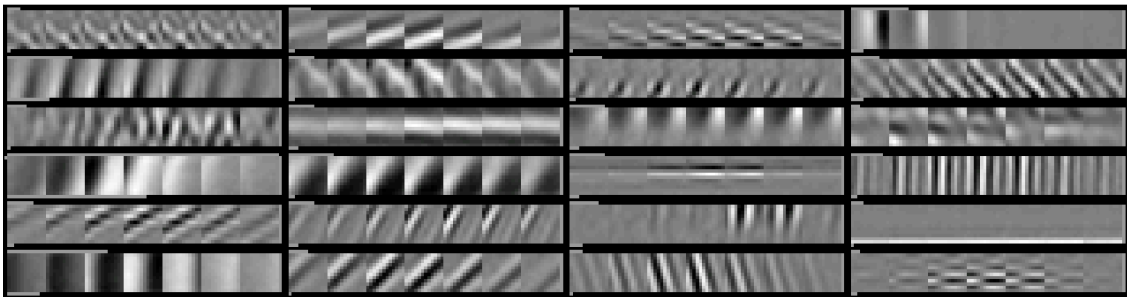


Figure 9: A subset of a large dictionary learned on natural movies. 12x12 by 7 frame random pixel patches were extracted from a whitened black and white natural movie of animals. The blocks represent the basis functions, each spatially 12x12 with a 7 point time kernel. A sparse convolved sum of these basis elements reconstructs the original patch sequence. Note the dominant statistical structure present in this data, moving edges of various orientations.

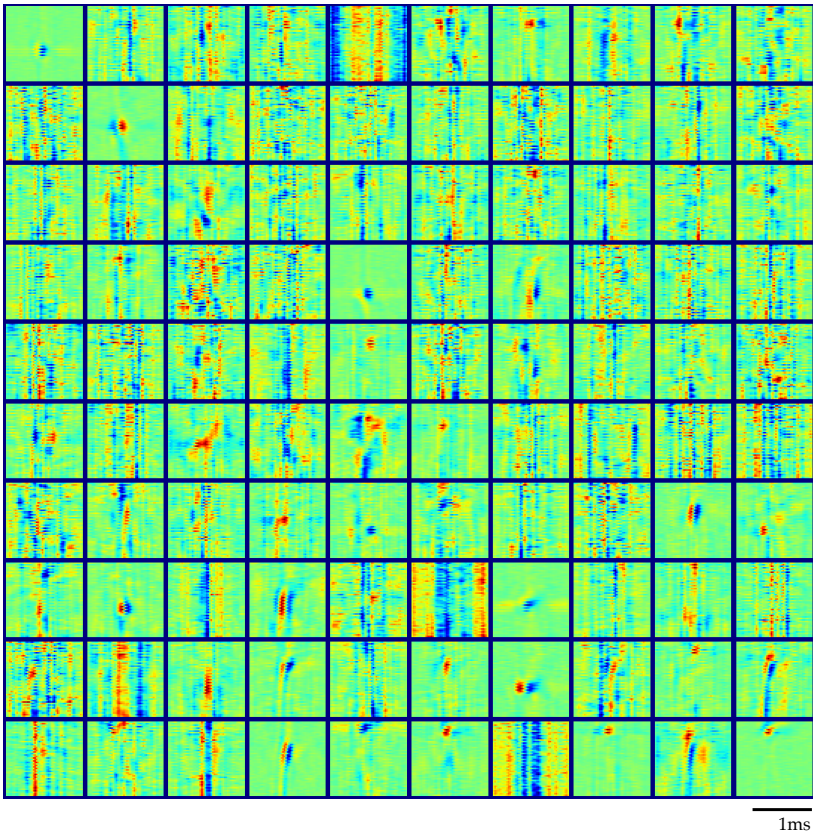


Figure 10: Learned spike convolution basis from polytrode data. Each block represents a convolution kernel with vertical axis corresponding to polytrode channel and horizontal axis representing time. Learned on only 10 seconds of data to demonstrate the ability of learning quickly online, possibly in an experimental setting. As a result, many of the basis elements have not yet learned anything.

Results on the silicon polytrode spike data corresponding to Fig. 2 is shown in Fig. 10. Sample reconstructions for random batches from the data are shown in Fig. 11. Several features of this learned basis are interesting.

- Many components are localized spatially and temporally, corresponding to putative action potentials. It is very likely that a given action potential is a sum a few basis elements rather than there being a direct correspondence between kernel elements and neurons.
- Several kernel elements have extended spatial structure that may hint at the morphology and cell type of the underlying neuron (see Fig. 12a). In practice, it is extremely difficult to categorize neurons isolated from extracellular recordings into different cell types. This method will potentially help in the classification task.
- There is a significant amount of noise in the signal from known and unknown sources (bottom of Fig. 12a). It is possible to denoise the data significantly by removing the contribution from basis functions that are known to be caused by experimental artifacts.

Learned kernel elements corresponding to the local field potential from polytrode data in Fig. 3 is shown in Fig. 13. Sample reconstructions of the data using these kernels is shown in Fig. 14. The learned elements

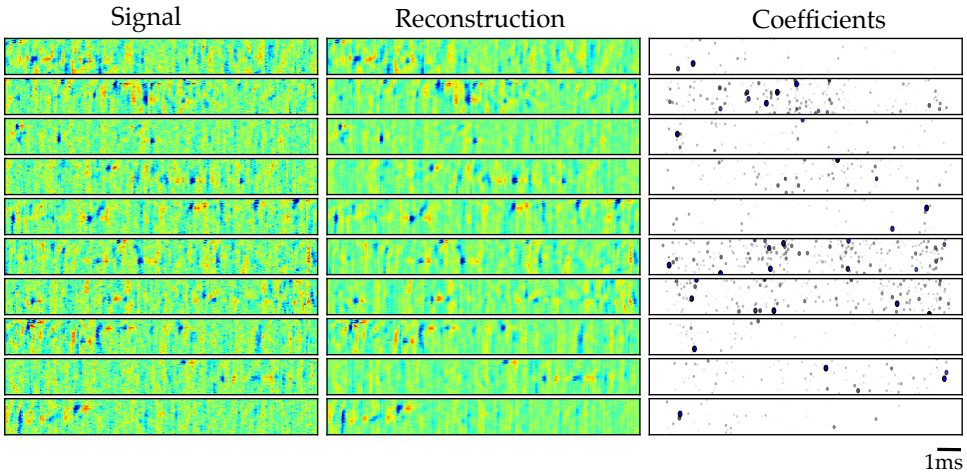


Figure 11: Sample reconstructions from the highpass filtered polytrode data. The first column represents the original signal, the middle column represents the reconstruction given inferred coefficients, and the last represents the sparse values with time of the coefficients with amplitudes proportional to the size of the dots.

have characteristic features that could have functional correlates and will be an area of further study. It is difficult to speculate on what they represent or how informative they are given the lack of knowledge regarding functional characteristics of the underlying biology. Some interesting elements are,

- Some components represent dominant oscillation modes in different frequency bands.
- A subset display layer dependent activity.
- Several frequency bands dominate activity.

Please see Fig. 12b for interesting examples and additional comments.

4 Implementation details

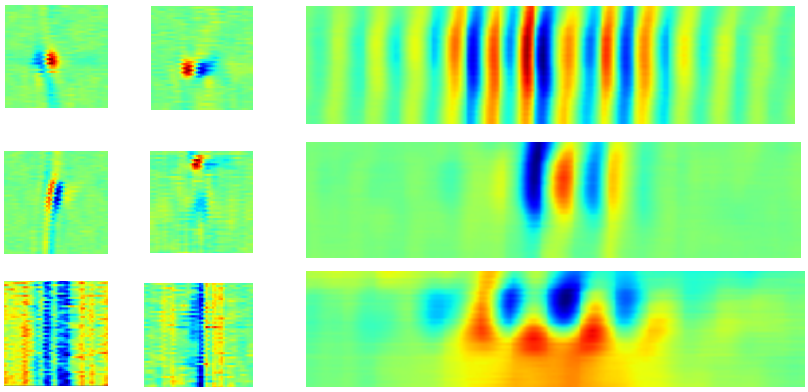
4.1 Objective and derivative calculations

The most expensive component in the overall algorithm is computation of the objective and derivatives of,

$$E(s, \phi) = \frac{1}{2} \sum_{mt} (y_{mt} - \sum_{np} \phi_{mnp} s_{n,t+p})^2$$

I tried a number of methods to optimize this calculation for a given $\mathbf{y}$ and $\mathbf{\Phi}$. These are outlined in Fig. 15. Here is a detailed chronology of what I did for this class project:

1. I began by defining the problem as sparse matrix-vector multiply with a reduction to compute the norm. Naively, I thought stating the problem in this form would allow me to use the broad set of published algorithms for computing Lasso by swapping out all BLAS operations with their corresponding sparse versions. However, there is no existing library that efficiently represents the particular sparse, block circulant matrix in this problem. Representing my matrix as generic sparse matrix was unrealistic for my problem sizes. This is something that is worthwhile pursuing. Though $\mathbf{\Phi}$ changes on each iteration, it's sparsity structure is preserved and the matrix could be efficiently updated.



(a) Sample spike basis elements

(b) Sample LFP basis elements

Figure 12: (a) Sample selections from the polytrode convolutional spike basis. At top are two basis elements that are localized in space and time. The middle pair have extended structure possibly indicative of a back-propagating action potential or a pyramidal neuron with two main compartments in different layers. The bottom pair are components making up known sources of noise in the experiment caused by channel cross-talk and, possibly, a layer of residual cerebrospinal fluid covering the electrode due to the process of insertion. Reconstructing the signal after having removed the contribution from these kernel elements will effectively denoise the data. (b) Sample basis functions learned from the polytrode LFP. The top element represents a gamma burst, activity distributed across all layers at a characteristic frequency and temporal extent. This is a dominant characteristic of this particular recording. The middle basis element shows activity localized to the supragranular layers whereas the bottom case shows different activity in the different lamina possibly due to signal passing between layers.

2. From class, I learnt that BLAS-3 operations are particularly efficient and if you can state your calculation solely in terms of `dgemm` operations, you can achieve high percentage of theoretical peak with optimal blocked memory accesses. The objective is trivial to implement, here as a series of CBLAS `dgemm` calls in Python,

```
y = np.zeros((M, T))
for b in range(P):
    y += np.dot(phi[:,:,b], s[:,b:b+T])

dy = yhat - Y
E = 0.5 * np.linalg.norm(dy)**2

deriv = np.zeros((N,T+P-1))
for b in range(P):
   deriv[:,b:b+T] += np.dot(phi[:,:,b].T, y)
```

   It was very difficult to improve on the timing of this strategy other than to notice a column order layout (default in Matlab but not in Python) was more memory and cache efficient due to contiguous access of **s**.

3. Upon Prof. Demmel's suggestion, I tried a variety of FFT schemes as convolutions can often be computed efficiently using the Fourier convolution theorem. I tried a number of R2C-C2R and C2C 2D and 3D FFT schemes on the CPU that were on the order of 15x slower than `dgemm`'s for my problem sizes, where kernel time dimensions are much smaller than the coefficients being convolved with and convolutions are degenerate along one dimension.

4. I also wrote the algorithm in Cython, a powerful and flexible new compiler with a set of extensions to the Python language. I wrote a wrapper for NVIDIA cuBLAS `sgemm` instructions in Cython, having

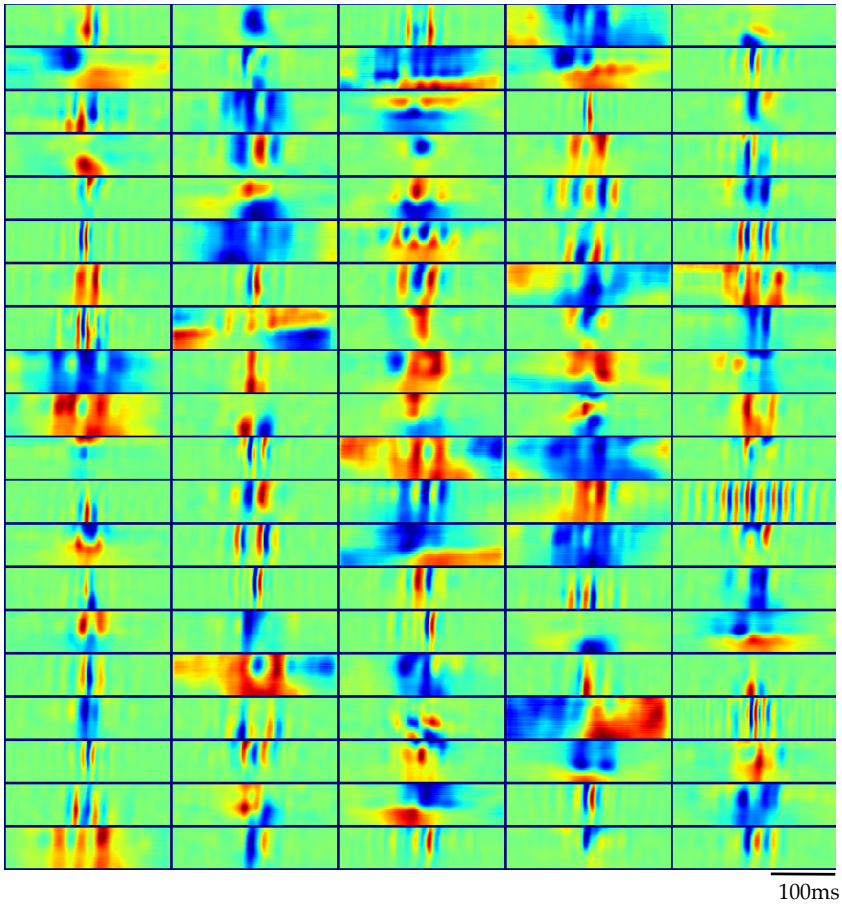


Figure 13: Learned convolution basis from polytrode data on data filtered between 0-150Hz (known as the LFP). Each block represents a convolution kernel with vertical axis corresponding to polytrode channel and horizontal axis representing time.

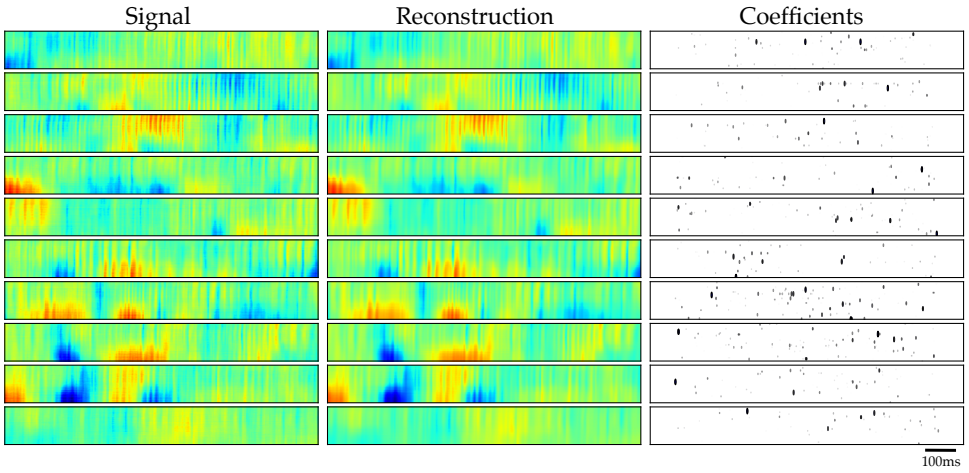


Figure 14: Sample reconstructions from the LFP polytrode data. As with the spike reconstructions, the first column represents the original signal, the middle column represents the reconstruction given inferred coefficients, and the last represents the sparse values with time of a 100 coefficients corresponding to the kernel elements.

little prior experience with CUDA. The implementation was 2-5 times faster than the CPU BLAS version but latency for device to host transfers more than offset the speed gains. For unrealistically large problem sizes, the speed-up was more dramatic as it more fully utilized the GPU per kernel call (only one kernel call may run at a time). I also received the source code for MAGMA from Stan Tomov. I tried the MAGMA `sgemm` and had speed-up for certain matrix sizes. However, for CUDA 3.0, the speed-up disappeared, possibly due to vendor improvements to cuBLAS.

5. With the recent release of CUDA 3.0, it is possible to use the driver API and the runtime API at the same time. This removed a large impediment to using CUDA from Python as the main Python module for calling CUDA, `pycuda` by Andreas Klockner, only wrapped the driver API. I wrote wrappers to directly call cuBLAS and MAGMA BLAS using `ctypes`, which loads and directly calls the functions in the respective shared libraries. I also tested these on the NERSC Turing and Tesla machines for which I was given an account by Horst. However, the NERSC support could not install CUDA 3.0 on these machines yet. I later replaced these `ctypes` wrappers with a just released open source python module called `PARRET` that did essentially the sample thing.

6. I also implemented the calculations as CUDA FFT2 and FFT3 calls. I observed that these calculations were orders of magnitude faster than their CPU counterparts. However, transferring arrays to and from device memory and subsequent array slicing to undo padding effects more than eliminated this speed advantage. I therefore resolved to perform the entire inference portion of the calculation on the GPU to eliminate all transfers (described in the next section).

7. There were several obstacles that took quite a bit of effort to overcome. One was that R2C-C2R FFT2 and FFT3 convolutions were much slower than their C2C counterparts no matter what padding I used. R2C obviates transforming back and forth from real to complex float32 matrices and saves half the memory. I wasn't able to resolve this issue. It is possible that they are just not implemented. A second challenge was array slicing in CUDA. Operations such as padding and slicing out 2D real sub-elements of a 3D complex matrix are difficult to perform efficiently in CUDA due to bank conflicts and uncoalesced memory transfers. The cuFFT library requires its inputs to be in linear memory rather than efficiently padded CUDA Array3's making efficient memory transfers difficult to implement. My

Method	Language	Comments
SpMV	Matlab	Redundant entries in compressed formats. Sparse structure changes with iteration.
Dense matrix	Matlab	Column ordering advantageous
`numpy` dense matrix	Python	32-bit MKL
fftn, rfftn	Python	Uses FFTW. 15x slower.
dgemm	Cython	64-bit ATLAS
cuBLAS sgemm	Cython	Runtime API only, CUDA 2.3.
cuBLAS, MAGMA sgemm	Python `ctypes`	Runtime API only, CUDA 2.3. Magma 0.2.
cuFFT rfft3	Python `ctypes`, `pycuda`	CUDA 3.0. 50x slower than C2C FFTs.
cuFFT fft3	Python `ctypes`, `pycuda`	Power of 2 padding. Texture map array slicing. CUDA 3.0. 30x faster only on GTX 295.

Figure 15: In summary, C2C GPU FFT3 method (in red) was 30x faster than reference (in yellow) using correct padding. cuBLAS was 3-5x faster than reference for normal problem sizes, but significantly faster for large sizes. Error in derivative computations due to 32-bit floats may be problematic.

solution was to predefine a series of maps for padding and slicing operations defined as GPU linear arrays. I then bound the source array to a linear texture map and read the array as a texture while mapping to the destination array. The thread call was done elementwise on the destination array. This seemed an adequate resolution.I am not sure if this is the best way to implement such transfers. Timing them using the CUDA profiler showed these mappings to take two order of magnitude less time than the main computations, the FFT3s, which is sufficiently small to ignore their cost.

8. In summary, the FFT3 implementation of the objective and derivative calculations were about 30x faster than my fastest BLAS-3 implementation when CUDA was performed on a high-end NVidia GTX 295. On a more generic GeForce 9600M card, the speedup was negligible. Our lab has 2 8-CPU-core machines each with two GTX 295's. A 30x speedup is a dramatic one if, additionally, all graphics cards are simultaneously used by separate MPI threads as I implemented (but did only preliminary testing due to heavy load on the machines). It is useful to note that the complexity of the `dgemm` method is $O(NTMP)$ whereas the FFT3 method has complexity of $O(NM(P+T-1)\log(NM(P+T-1))$. Assuming $P \ll T$, and using typical dimensions used in my experiments, the methods are of the same order. All the speed up is therefore due to the parallelism built into the GPU implementation of FFT3. In the future, I plan to use convolutions in more than a single dimension. I expect the speed gains for convolutions in these cases to be significantly faster in the FFT3 case.

4.2 Sparse approximation with orthant-wise l-BFGS

l-BFGS is a standard quasi-Newton unconstrained serial optimization algorithm [15]. It maintains a ring buffer of vectors that it uses to construct and perform rank-2 updates of a positive definite inverse Hessian at each step. Other notable features are that line searches need not be very accurate unlike non-linear conjugate gradient, that memory usage is very limited making large scale optimizations possible, and there is an absence of BLAS-2 or 3 operations. The L_1 regularized objective in the Lasso is not differentiable at

Method	Language	Comments
SSE2 intrinsics	C++, Cython	Modification of liblbfgs.
BLAS	Cython	Warm start of ring buffer. Vector regularization.
cuBLAS, custom kernels	Python `ctypes`, `pycuda`	Done entirely in GPU in batch. Speculative line searches, inverse Hessian warm start.

Figure 16: In summary, cuBLAS was 3-5x faster than SSE2 intrinsics. CPU BLAS was slower than SSE2 on single core, only slightly slower on multi-core. Limit of speed-up in l-BFGS may be due to no use of BLAS-3.

the origin. Therefore, a naive implementation not taking this into account will lead to numerical errors at the origin. This is compounded by the fact that the minimum does in such regression problems typically lie on the coordinate axes.

Orthant-wise l-BFGS addresses this issue in several ways. It recognizes that the objective is linear over a given octant, the Hessian is not a function of the regularizer, and that a clever choice of subgradient on a given sectant results in a provably convergent algorithm [6]. I implemented 3 different versions of this algorithm, summarized in Fig. 16. The first was a modication of the small C-library `libLBFGS` by Naoaki Okazaki which implements orthant-wise l-BFGS. One notable feature of `libLBFGS` is that all vector operations are performed using custom SSE2 intrinsics making the overall algorithm quite fast on a single core. I wrote a Cython wrapper for small modifications of this library. In MPI computations, I made sure my Python distribution and numerical libraries (eg. ATLAS, FFTW) were optimized for a single core as `libLBFGS` would not be able to take advantage of multiple cores.

In order to test convergence issues with the algorithm, particularly conditioning issues with the Hessian $\mathbf{\Phi}^\mathbf{T}\mathbf{\Phi}$, I rewrote orthant-wise l-BFGS in Cython with direct calls to BLAS, taking pains to remove all Python associated overhead. I added diagnostic algorithms to unroll the l-BFGS inverse Hessian to compare with the real Hessian. This Cython BLAS conversion made the optimization roughly 2x slower but it facilitated later rewriting the entire algorithm using CUDA calls using custom elementwise kernels and reductions written using `pycuda` and cuBLAS. The aim of the CUDA conversion was to eliminate the need for device to host memory transfers. I expected the CUDA implementation to be slower that the C version as it is written partly in Python, but it turned out to be several times faster. I added additional optimizations such as batched memory transfers and warm starts of the inverse Hessian for Lasso calculations using a fixed $\mathbf{\Phi}$. The code is most efficient when each kernel call achieves maximal usage of the GPU as only one such kernel may be called at a time.

Future work will focus on quasi-Newton parallel unconstrained optimization algorithms. With some preliminary research, I was surprised how little I could find on this topic. In the problems I have typically encountered, objective and derivative computations dominate all else, so it is not clear much can be gained by parallelizing the underlying quasi-Newton as well. Orthant-wise l-BFGS could be improved by some parallelization, such as speculative line searches that don't lead to optimal steps, but instead more quickly accumulate a local characterization of the Hessian. I would be very resistant to implementing an optimization algorithm exploiting detailed characteristics of a particular CUDA-based graphics card as I anticipate this area will experience a lot of change, making code obsolete quickly.

4.3 Parallelized batch inference

The overall learning algorithm was parallelized using MPI per batch of data in the inference step. The method is outlined in Algorithm 2. The MPI calls were done in Python and `mpi4py` and were almost trivial to implement. I performed some profiling using MPE and `jumpshot`, an MPI instrumentation library that is part of the MPICH2 MPI distribution. Dynamic rebalancing of the `MPI.Scatter` to nodes was done according to time of batch calculation. This was necessitated mostly because students in my lab have a habit

of ssh-ing directly into machines rather than using the PBS scheduler slowing down my runs by slowing down just one node. The algorithm was run on a Mac OS laptop with GPU and a small Linux cluster. I also was able to get it to run unchanged on Hopper with much effort and the help of NERSC support, though my runs were only diagnostic.

Initialization: Initialize GPU if detected and free. Use GPU methods instead of CPU.

```
for i ← 1 to ∞ do
    MPI.Broadcast basis to nodes
    MPI.Scatter data batch to nodes
    Perform inference on data with Hessian warm start on GPU or CPU
    MPI.Gather coefficients to root
    MPI.Gather timing results
    Root update of basis
    Dynamically load balance next Scatter
end
```

Algorithm 2: Hybrid MPI and GPU iteration scheme

An interesting aspect of the algorithm is that, on startup, it checks to see if a CUDA GPU is present. If so, inference calculations are performed using the CUDA based version of orthant-wise l-BFGS instead of the SSE2 intrisics version. I have observed that the use of 32-bit floats necessitated by cuFFT results in inference calculations that are different than the 64-bit CPU versions. They also take longer to converge. This is an area for further work. As with all iterated large matrix operations with large reductions, there can be sources of substantial error that would require, for example, compensated summation as is done in cuBLAS `snrm2` or a more careful choice of algorithms. But lots of care is not so warranted in this case as the optimization is stochastic and online and not convex. Accuracy is not critical. It is more likely that a faster and less accurate algorithm is warranted. Substantial differences in results would suggest a fundamental flaw in the formulation of the problem rather than argue for a change from float to double. The brain itself seems to operate at low precision.

5 Conclusions and plans for future work

My plan for continuing work is as follows in order of importance,

1. Use the inferred coefficients as inputs to the next layer of models. It is critical to demonstrate that these coefficients are more informative for tasks such as assigning spike events on the polytrode to a specific neuron. This will require an additional clustering step and it is unclear whether a sparse overcomplete set of coefficients which lie mostly on the coordinate axes will offer a better space to segregate different clusters. The inferred coefficients can also be fed into system identification regression models that try to relate the activity in cortex with the stimulus or behaviour of the animal.

2. Apply the algorithm to other datasets, such as the public CRCNS data sharing repository, funded by the NIH. This database brings together a wide variety of electrophysiology data. It would be interesting to see what types of learned kernels can be derived from these data sets and if they share any properties in common. The data is maintained at the Redwood Center.

3. There is a large gap in our theoretical understanding of these statistical models. Though the Lasso problem has been studied intensely, the problems of consistency of estimating sparse overcomplete kernel representations is largely open.

4. I've already mentioned these, but parallel algorithms both in the inference and learning steps would be an area of future study. My preliminary literature review has not been encouraging. Is it more

worthwhile to create purely parallel algorithms that will be directly implemented on a given architecture or is it more worthwhile to devise algorithms that reduce to operations in already optimized libraries such as BLAS, LAPACK, or cuFFT? I suspect the answer is the latter, that it is more worthwhile to devise better sparse matrix vector multiply algorithms as they have universal applicability than spend a great deal of effort to create a custom algorithm that exploits some aspect of CUDA shared memory on a particular card.

5. The underlying model for the learning algorithm in this project is a highly impoverished one. It is almost surely the case that coefficients inferred will have non-trivial statistical dependencies that contradict the assumption of independence. Hierarchical probability models can attempt to account for these dependencies and provide richer and more appropriate explanations of the causes of the data. For example, though neurons fire sparsely in cortex, they often do in concert or in highly specific temporal patterns. While a simple generative model could identify single neurons in a recording, a hierarchical model would be able to recognize coincident firings of a population. An ideal hierarchical model would be one from which samples looked indistinguishable from actual recordings from cortex. Other models such as state-space models with non-Gaussian statistics (Kalman filters are Gaussian) are another area of possible study.

6 Acknowledgements

I would like to thank my advisor Bruno Olshausen on whose ideas this work is based and Charles Gray for allowing to me to work on these unique neural datasets. Future work will be done with Jack Culpepper, the undisputed world expert at unsupervised learning of sparse bases.

References

1. Blanche, T. J., Spacek, M. A., Hetke, J. F. & Swindale, N. V. Polytrodes: high-density silicon electrode arrays for large-scale multiunit recording. *Journal of Neurophysiology* **93,** 2987–3000 (2005).
2. George, D How the brain might work: A hierarchical and temporal model for learning and recognition. *PhD Thesis* (2008).
3. Olshausen, B. A. & Field, D. J. Sparse coding with an overcomplete basis set: a strategy employed by V1? *Vision Research* **37,** 3311–25 (1997).
4. Tibshirani, R Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* 267–288 (1996).
5. Chen, S., Donoho, D. & Saunders, M. Atomic decomposition by basis pursuit. *SIAM review* 129–159 (2001).
6. Andrew, G & Gao, J Scalable training of L 1-regularized log-linear models. *Proceedings of the 24th International Conference on Machine learning* 33–40 (2007).
7. Olshausen, B. A. & Field, D. J. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature* **381,** 607–9 (1996).
8. Boyd, S & Vandenberghe, L Convex optimization. *Cambridge University Press* (2004).
9. Mairal, J, Bach, F, Ponce, J & Sapiro, G Online learning for matrix factorization and sparse coding. *Journal of Machine Learning Research* **11,** 19–60 (2010).
10. Bertsekas, D. P. Nonlinear programming. *Athena Scientific* 646 (1995).
11. Kim, S, Koh, K, Lustig, M, Boyd, S & Gorinevsky, D An Interior-Point Method for Large-Scale$ $\text{ell}_1$$ $- RegularizedLeastSquares. Selected Topics in Signal Processing$ (2007).
12. Friedman, J, Hastie, T, Höfling, H & Tibshirani, R Pathwise coordinate optimization. *Annals of Applied Statistics* (2007).

13. Raina, R, Madhavan, A & Ng, A. Large-scale deep unsupervised learning using graphics processors. *Proceedings of the 26th Annual International Conference on Machine Learning* 873–880 (2009).

14. Smith, E. C. & Lewicki, M. S. Efficient auditory coding. *Nature* **439,** 978–82 (2006).

15. Nocedal, J. & Wright, S. J. Numerical optimization. *Springer* 636 (1999).