Goethe Universität Frankfurt

FIAS Riedberg

Supervisor: Prof. Matthias Kaschube

Optimizing Efficiency of Neural Population Spiking Rate Inference from Wide-Field Calcium Imaging

Amon Khavari

s2121460@stud.uni-frankfurt.de

Contents

1 Introduction **2**

2 Fundamentals **3**

2.1 Calcium Imaging 3

2.2 (De-)convolution 4

2.3 Algorithm: **Convar** 7

3 Optimization **9**

3.1 Linear Algebra Libraries 9

3.2 Chunking 10

3.3 Early Stopping 13

3.4 First Differences Deconvolution and Meaningful Initialization 15

3.5 Adaptive Learning Rate 17

3.6 Binary λ-Search 18

4 Results **21**

5 Discussion **26**

6 Conclusion **28**

Optimizing Efficiency of Neural Population Spiking Rate Inference from Wide-Field Calcium Imaging

Amon Khavari

July 12, 2021

Zusammenfassung

Neuronale Aktivität ist allgegenwärtig in unseren Leben. Sie ist ein wesentlicher Bestandteil des Uhrwerks hinter unseren Gedanken. Sie zu messen ist jedoch eine nicht-triviale Aufgabe. Mit Hilfe von Weitfeld-Kalzium-Imaging können wir über weite Bereiche des Gehirns die neuronale Aktivität von Mäusen aufzeichnen. Leider ist diese Technik mit einem komplexen Problem konfrontiert, das die Dynamik des fluoreszierenden Kalziumindikators betrifft. Die Spikes, die aus der neuronalen Aktivität resultieren, werden durch die zeitliche Dynamik des Kalziumindikators maskiert, oder besser gesagt damit 'gefaltet'. Das Rauschen, das bei der Aufnahme dieser Bilder involviert ist, verkompliziert dieses Problem weiter. Stern et al. [1] haben einen Artikel veröffentlicht, der sich mit 'Entfaltung' im Kontext der Weitfeldmikroskopie befasst, um die zugrunde liegende neuronale Aktivität zu bergen. Der Code, der mit diesem Artikel veröffentlicht wurde, ist jedoch in Bezug auf die Leistung unzufriedenstellend. Wir schlagen mehrere Optimierungsmaßnahmen vor, die darauf abzielen, die Ausführungszeit zu beschleunigen, bis hin zu unserer eigenen modifizierten Version des besagten Programms, die im Durchschnitt etwa 200 Mal schneller ist.

Abstract

Neural activity is ever-present in our lives. It is a substantial part of the very clockwork behind our thoughts. However, measuring it is a non-trivial task. Using wide-field calcium imaging we are able to record neural activity in mice spread across vast areas of the brain. Unfortunately this technique is met with a complex problem concerning the dynamics of the fluorescent calcium indicator. The spikes resulting from neural activity are masked, or rather 'convolved' with the temporal dynamics of the calcium indicator. Noise introduced during the recording of these images further complicates this problem. Stern et al. [1] have published an article that deals with 'deconvolution' in the context of wide-field microscopy in order to retrieve the underlying neural activity. The code that is published with this article however is unsatisfactory in terms of performance. We propose several optimization measures aimed at accelerating execution time, building up to our own modified version of said program that is about 200 times faster on average.

1 Introduction

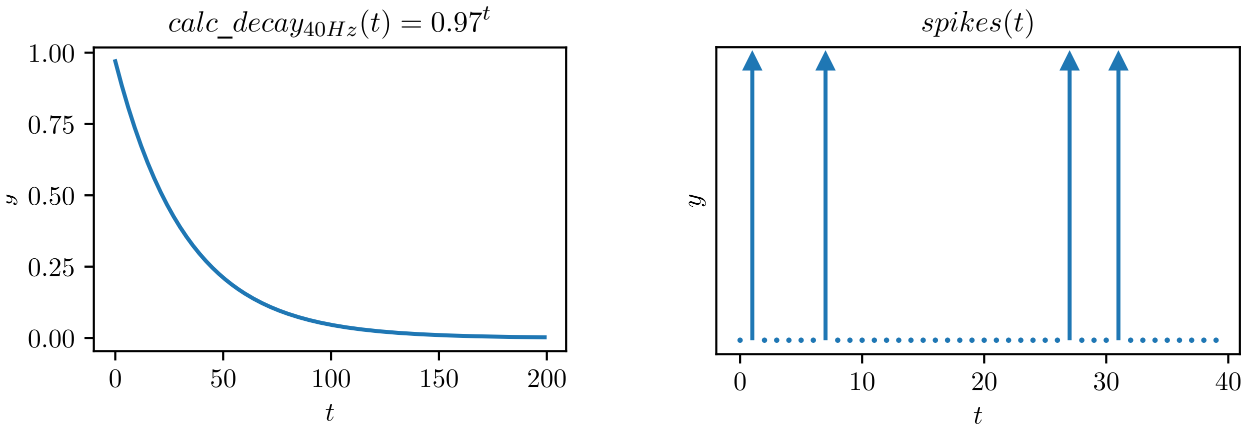
Calcium imaging techniques are commonly used to measure neural activity in the brain. They allow us to image neural activity in awake and behaving animals [2,3,4,5,6] by using fluorescent calcium indicator molecules [7], optically measuring the calcium influx induced by neural activity. The fluorescence traces that result from recording these images over time however are masked by the calcium indicator's dynamics which differ from the neural activity's, inadvertently involving 'deconvolution' to retrieve the actual neural activity. There are a variety of articles dealing with inferring the neural activity from calcium imaging [8,9,10,11,12], yet only Stern et al. [1] use wide-field microscopy in combination with calcium imaging which differs most significantly (regarding the 'deconvolution') in that images cover large areas of the brain [13] but do not provide resolution at a cellular level; thus images contain multiple neurons per pixel [1,5]. Wide-field calcium imaging is essential to analyze even simple behaviours since the brain is anatomically and functionally broken up into regions [13].

The algorithm Stern et al. [1] propose to 'deconvolve' fluorescence traces and estimate underlying neural activity yields satisfactory results [1]. It is however fairly unoptimized and thereby poor in performance. In this work we provide an overview of calcium imaging and in that very context 'deconvolution' (Chapter 2.1). Then we proceed to summarize Stern et al.'s [1] approach to 'deconvolution' (Chapter 2.2) and the corresponding algorithm they publicly provide (Chapter 2.3). Following this we explain our own optimization methods and their effects on performance, starting with general changes to the implementation (Chapter 3.1) to handling large data (Chapter 3.2) and finally optimization measures we consider to be universally usable (Chapters 3.3, 3.4, 3.5). We will then present the results of our optimization (Chapter 4) and subsequently discuss them (Chapter 5).

2 Fundamentals

2.1 Calcium Imaging

To understand calcium imaging we need to know that neural activity induces a rapid calcium influx. We rely on this very principle to locate areas where neurons are active using fluorescent calcium indicator molecules that respond to said calcium[7]. Using wide-field microscopy we are able to capture broad areas of the cortex. On these images the value of each pixel represents the light emitted by the fluorescent calcium indicator molecules which in turn is induced by neural activity. Recording them over time t provides us with a video where each pixel represents a fluorescence trace. However, the fluorescent calcium indicator dynamics differ from the underlying calcium's dynamics and thus the actual spiking activities, effectively masking them. The problem to be dealt with is the slow decay in particular[7,14].



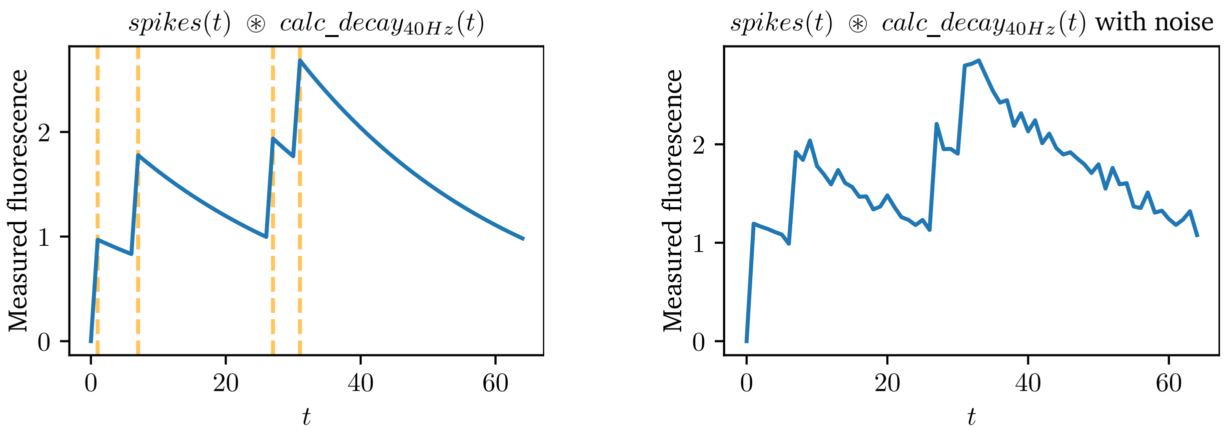
(a) Calcium indicator decay function that acts as a convolution kernel in calcium imaging.

(b) Multiple spikes of activity called 'spike train'.

Figure 1: Basic examples of calcium imaging components. (a) represents the fluorescence calcium indicator's decay behaviour. (b) represents spikes resulting from neural activity.

We model the decay behaviour as an exponential function $f(t) = \gamma^t$ where γ is the calcium decay rate over time t. Other works measured γ to be 0.97 at 40 Hz for the fluorescence calcium indicator we deploy[7,15], represented by $calc\_decay_{40Hz}(t)$ (Fig. 1a). The neural spiking activities of a single neuron on the other hand can be modelled as a sum of delta-functions $\delta(t)$, each of them with a different horizontal shift, represented by $spikes(t)$ (Fig. 1b).

Now $spikes(t) \circledast calc\_decay_{40Hz}(t)$ describes the convolution between a neuron's spiking activity and the decay kernel of the calcium indicator. It is noteworthy that the delta-function acts as the identity for convolution, $f(t) \circledast \delta(t) = f(t)$. Considering the sifting property of the delta-function we can introduce a constant c for horizontal shift, $f(t) \circledast \delta(t-c) = f(t-c)$. In other words, at every single horizontally shifted delta-function spike there will now be an instance of the calcium decay function where each instance bleeds into the next (Fig. 2a). To retrieve the underlying spiking activities we now need to perform a *deconvolution*. Looking at Fig. 2a one could very easily decode the underlying neural activity by simply marking every timestep where the observed fluorescence is higher than in the previous timestep. Noise introduced in involved processes however masks our desired information to an extent where this task is non-trivial (Fig. 2b).



(a) Neural activity convolved with calcium decay function. Spikes are highlighted.

(b) Neural activity convolved with calcium decay function and added noise.

Figure 2: Simulated models of fluorescence traces resulting from calcium imaging. (a) shows the spike train from Figure 1b convolved with the calcium decay kernel from Figure 1a. (b) differs from (a) in that noise from involved processes has been added, masking the underlying neural activity to a non-trivial extent. No baseline fluorescence is included in this toy example.

2.2 (De-)convolution

As explained earlier, the neural activity we want to retrieve from calcium imaging recordings is masked by the calcium indicator's dynamics where the decay paired with noise obscures data beyond triviality. Vogelstein et al.[8] model this problem using an auto-regressive model for the activity of a single neuron as seen in the following equation,

$$\begin{aligned} &y_t = \beta_0 + \beta_1 c_t + \epsilon_t, \qquad \epsilon_t \sim_{ind.} (0, \sigma^2), \qquad t = 1, ..., T, \\ &c_t = \gamma c_{t-1} + s_t, \qquad t = 2, ..., T, \end{aligned} \tag{1}$$

where y_t describes the observed fluorescence, $s_t \geq 0$ a potential spike, c_t the unobserved calcium and $\gamma \in (0, 1)$ the previously explained calcium decay rate, each respective to their corresponding timestep t. The baseline fluorescence β_0 describes the amount of fluorescence already present at $t = 0$ and must be estimated while β_1, the baseline fluorescence at $t = 1$, can be set to 1 without loss of generality since it's equivalent to scaling with a constant factor[1]. To briefly word Model (1), c_t decays exponentially unless a spiking event $s_t > 0$ occurs, in which case $c_t > \gamma c_{t-1}$. Also as already stated the observed fluorescence y_t is influenced by noise ϵ_t.

Model (1) is fit with $\beta_1 = 1$ by solving the minimization problem[9,10,11]

$$\begin{aligned} &\text{minimize}_{c_1,\ldots,c_T,\beta_0} \left\{ \sum_{t=1}^{T} (y_t - \beta_0 - c_t)^2 + \lambda P(c_t - \gamma c_{t-1}) \right\} \\ &\text{subject to } c_t \geq \gamma c_{t-1}, \quad t = 2, \ldots, T. \end{aligned} \tag{2}$$

The first term of the minimization term equates to the noise ϵ_t as

$$y_t = \beta_0 + c_t + \epsilon_t \tag{3}$$

$$\Leftrightarrow \qquad \epsilon_t = y_t - \beta_0 - c_t, \tag{4}$$

which signifies that the minimization problem gives us a set of $c_1, \ldots, c_T$ and β_0 with a minimal ϵ_t while including a penalty function $P(\cdot)$ that prohibits the trivial solution $\beta_0 = 0,\ c_t = y_t$. In addition to that, $P(\cdot)$ is also intended to induce sparsity in its arguments $c_t - \gamma c_{t-1}$ for $t = 2, \ldots, T$ such that $c_t - \gamma c_{t-1} = s_t$ is estimated to be zero at most timesteps[1]. Other works employ l_0 [10,11] and l_1 [9,16] penalty functions for this purpose. $\lambda \in (0, \infty)$ describes a tuning parameter that is probed for as a prerequisite of the main deconvolution algorithm.

Stern et al.[1] transform the auto-regressive model from previous works stated in Equation (1) to be applicable to wide-field calcium imaging which is necessary since wide-field calcium imaging contains a population of neurons as opposed to a single neuron per fluorescence trace (or pixel over time),

$$\begin{aligned} &\overline{y}_t = \overline{\beta}_0 + \overline{c}_t + \overline{\epsilon}_t, \qquad \overline{\epsilon}_t \sim_{ind.} \left(0, \sum_{j=1}^{p} (\sigma^j)^2 \right), \qquad t = 1, \ldots, T, \\ &\overline{c}_t = \gamma \overline{c}_{t-1} + r_t, \qquad t = 2, \ldots, T, \end{aligned} \tag{5}$$

where the overline above a variable indicates that we consider the variable to be the sum over all p neurons inside a pixel. Instead of the spiking activity per timestep s_t, the variable r_t now appears as the amount of calcium increases due to spiking events in the p neurons. This is referred to as the *spiking rate*. Note that since we use wide-field microscopy each fluorescence trace (pixel over time) potentially contains a large number of p neurons depending on the density of neurons in the respective regions of the subject's brain and spatial resolution of the experiment. These numbers range from a handful to thousands of p neurons.

Applying the changes from Model (5) to the minimization problem in Equation (2) Stern et al.[1] state

$$\begin{aligned} &\text{minimize}_{\overline{c}_1,\ldots,\overline{c}_T,\overline{\beta}_0} \left\{ \sum_{t=1}^{T} (\overline{y}_t - \overline{\beta}_0 - \overline{c}_t)^2 + \lambda P(\overline{c}_t - \gamma\overline{c}_{t-1}) \right\} \\ &\qquad\qquad \text{subject to } \overline{c}_t \geq \gamma\overline{c}_{t-1}, \quad t = 2, \ldots, T, \end{aligned} \tag{6}$$

which differs only in its consideration of wide-field microscopy. The penalty function however should, in contrast to works that deal with single neurons per trace, not be sparsity inducing since each timestep represents the measured fluorescence resulting from multiple neurons. Instead they use a penalty function that encourages continuosly varying spiking rates over time $r_t \approx r_{t-1}$,

$$P_n(r_2, \ldots, r_T) = \sum_{t=3}^{T} (r_t - r_{t-1})^2, \tag{7}$$

while also reparametrizing our penalty function arguments to $r_1, \ldots, r_T$ since $r_t = \overline{c}_t - \gamma\overline{c}_{t-1}$ for $t = 2, \ldots T$, as stated in Equation (5), with $r_1 \equiv \overline{c}_1$. This also applies to the minimization term arguments.

Further transformation of Equation (6) leads to

$$\begin{aligned} &\text{minimize}_{r_1,\ldots,r_T} \left\{ \|\tilde{y} - Ar\|^2 + \lambda \sum_{t=3}^{T} (r_t - r_{t-1})^2 \right\} \\ &\qquad\qquad \text{subject to } r_t \geq 0, \quad t = 2, \ldots, T, \end{aligned} \tag{8}$$

where various propositions stated in Stern et al.[1] allow us to state the minimization problem in two-dimensional space. In this, $\tilde{y}$ describes the mean subtracted version of $\overline{y}$ by $\tilde{y} = (I - \frac{1}{T}1_T 1_T^\top)\overline{y}$ with I the $T \times T$ identity matrix and $\frac{1}{T}1_T 1_T^\top$ a $T \times T$ matrix of ones. Ar can be abstractly described as somewhat equivalent to the underlying calcium $\overline{c}$ since A is the column-wise mean subtracted Version of D^{-1} where D is a $T \times T$ full-rank matrix with 1s on the diagonal and $-\gamma$ one index ahead of the 1s per row such that $r = D\overline{c} \equiv rD^{-1} = \overline{c}$. Note that we obtain $\overline{\beta}_0$ in the following closed form expression after solving Equation (8) using Proposition 3[1]

$$\overline{\beta}_0 = \frac{1}{T}(1_T)^\top(\overline{y} - D^{-1}r), \tag{9}$$

which is why $\overline{\beta}_0$ doesn't explicitly appear in the minimization problem anymore.

2.3 Algorithm: Convar

Seeing as Equation (8) is a convex optimization problem[17] Stern et al.[1] choose a proximal gradient descent algorithm from Parikh and Boyd[18]'s work that deals with proximal algorithms out of a number of already existing efficient algorithms that would be applicable to this problem. As mentioned above we use a penalty function that encourages continously varying spiking rates over time, hence the algorithm name **Convar**.

```
Convar algorithm[1,18]
Given the signal $\overline{y}$, the calcium decay rate $\gamma$ and the tuning parameter $\lambda$, do:
  1. Calculate $\tilde{y} = (I - \frac{1}{T}1_T 1_T^\top)\overline{y}$.
  2. Set step size $s = \frac{1}{2}\frac{(1-\gamma)^2}{(1-\gamma^T)^2+4\lambda(1-\gamma)^2}$, and $A = (I - \frac{1}{T}1_T 1_T^\top)D^{-1}$.
  3. Initialize $r \in unif[0,1)$
  4. Iterate until convergence (10000 iterations per Stern et al.[1] code default):
     (a) Let $r \leftarrow r + 2sA^\top(\tilde{y} - Ar) - 2s\lambda Zr$
         with $Z_{i,j} = \begin{cases} -1, & \text{if } |i-j| = 1 \text{ and } i+j > 3 \\ 2, & \text{if } i = j \text{ and } 2 < i < T \\ 1, & \text{if } i = j = 2 \text{ and } 2 = i = T \\ 0, & \text{otherwise} \end{cases}$
     (b) Let $r_t \leftarrow (r_t)_+$ for $t = 2, ..., T$.
  5. Let $\overline{\beta}_0 = \frac{1}{T}(1_T)^\top(\overline{y} - D^{-1}r)$.
```

Proximal gradient descent is, in essence, very similar to the gradient descent we might know from machine learning in that it calculates a gradient per iteration that is added to a solution, e.g. a set of weights or, in our case, spiking rates $r_1, ..., r_T$. A significant consequence of Equation (8) being a convex optimization problem is that, in contrast to conventional gradient descent, every local optimum is also a global optimum[17].

The general workflow (Fig. 3) of Stern et al.'s[1] **Convar** deconvolution consists of first executing **Convar** $\#L$ many times, where $\#L$ is the number of elements in a set L of potential λ, and comparing their results to find the best λ. We will be using that λ in a final **Convar** execution. The resulting set of spiking rates $r_1, ..., r_T$ from this final **Convar** execution is considered to be the result of the deconvolution. This process is called λ-**Search**.

The first step in Stern et al.'s[1] λ-**Search** is halving the input signal data $\overline{y}$ into even and odd timesteps resulting in $\overline{y}_{odd}$ and $\overline{y}_{even}$ while also accounting for the halved frame rate by $\gamma_{adjusted} = \gamma^2$.

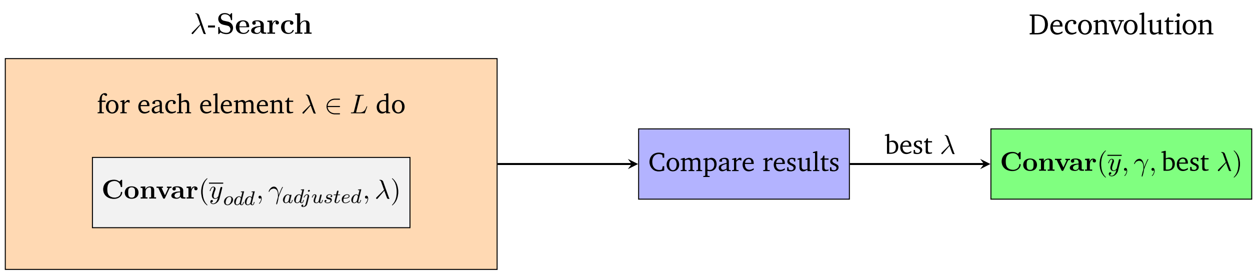

λ-Search
for each element $\lambda \in L$ do
$\textbf{Convar}(\overline{y}_{odd}, \gamma_{adjusted}, \lambda)$
Compare results
best λ
Deconvolution
$\textbf{Convar}(\overline{y}, \gamma, \text{best } \lambda)$

Figure 3: General workflow of Stern et al.'s[1] **Convar** Deconvolution (given the signal $\overline{y}$, the calcium decay rate γ and a set L containing potential λ). The first step in deconvolving a dataset is finding the best λ by utilizing λ-Search where a **Convar** run is executed for each potential tuning parameter $\lambda \in L$ with a set L of potential λ. Then the results are compared, returning the best λ that we finally deconvolve the dataset with it.

We pick one of these signal sets and execute **Convar** for every $\lambda \in L$ with both the halved signal set and $\gamma_{adjusted}$. Stern et al.[1] pick $\overline{y}_{odd}$ in this step although our choice here doesn't matter as long as we pick the opposite (in this case $\overline{y}_{even}$) later on when comparing. For each of our $\#L$ many **Convar** executions we retrieve spiking rates which leave us with $r^{\lambda_1}, ..., r^{\lambda_{\#L}}$. We then reconstruct $\overline{y}_{odd}$ by

$$\begin{aligned} D^{-1} r^{\lambda_x} + \beta_0^{\lambda_x} &= \overline{y}^{\lambda_x}_{reconstructed} \\ &\text{for } x \in [1, \#L] \end{aligned} \tag{10}$$

and use $\overline{y}_{even}$ to declare that λ_x the best where $mean\left(|\overline{y}^{\lambda_x}_{reconstructed} - \overline{y}_{even}|\right)$ is minimal. In other words, we deconvolve $\overline{y}_{odd}$ with every potential λ then reconstruct $\overline{y}_{odd}$ by first multiplying our results of the deconvolution back with the calcium decay matrix D^{-1} then adding the baseline fluorescence β_0 and compare it to $\overline{y}_{even}$, picking the λ where the difference is smallest. Of course, this algorithm is based on the assumption that neighbouring timesteps in our traces have similar values.

Stern et al.[1] designed **Convar** with an eye on parallelization which is why our input signal $\overline{y}$ is typically not a vector of length T containing only a single fluorescence trace with T timesteps but rather a $T \times P$ matrix containing P many fluorescence traces. In the case of 2D $N \times M$ images over time T (i.e. a video) we flatten dimensions N and M so that $P = N \cdot M$ and $\overline{y} : T \times P$ where the exact manner of flattening does not matter as long as the resulting deconvolved spiking rates r are unflattened/reconstructed the same way. Each fluorescence trace in P is completely independent from others.

3 Optimization

In this chapter we build upon the previously explained **Convar** algorithm, introducing various optimization measures in each of the following subchapters which lead up to our modified version of Stern et al.'s[1] program. The system we employ in each of the following tests uses a Ryzen 7 3700x CPU with 32 GB DDR4-3200 MHz memory and a NVIDIA GTX 980 Ti GPU.

3.1 Linear Algebra Libraries

Operation	Step	Dimension	Complexity	Amount in Step	Real Time Portion [%]
Matrix inversion	2	$T \times T$	$O(T^3)$	1	<1
Matrix multiplication	4	$T \times T$ & $T \times P$	$O(T^2P)$	$3 \cdot iterations$	~92
Remaining operations	4	$T \times P$	$O(TP)$	$5 \cdot iterations$	~7

Table 1: Most prominent operations and their role in the **Convar** algorithm regarding complexity and runtime. Operations that have not been mentioned in this table are not noteworthy regarding complexity and/or not optimizable, e.g. occasional algebraic operations or data loading.

Taking a high-level look at the operations that make up the **Convar** algorithm (Table 1) we observe that the majority of time required for **Convar** executions is spent in the main loop. Although the initialization, namely the matrix inversion, has the worst complexity of all operations it only needs to be executed once per **Convar**, thus being of low importance as long as T does not grow big enough for the matrix inversion to dominate our runtime. T-chunking does effectively prevent this for datasets with large T however as we will see later (Chapter 3.2). The matrix multiplication on the other hand is executed three times per iteration where the number of iterations per **Convar** is set to be 10000 per default. Considering this, our first line of optimizations is aimed towards accelerating execution time of prominent functions in **Convar**, namely matrix multiplication, and while at it linear algebraic calculations in general.

We do this by implementing **Convar** in several different versions (Table 2), comparing their respective runtimes and CPU usages. Notable entries besides numerous Python versions are Stern et al.'s[1] original MATLAB and our C++ implementation. The MATLAB implementation is interesting since it achieves good benchmarks, fully utilizing the CPU while not requiring extensive coding work to do so. C++ allows us full control over memory and involved processes as opposed to interpreters automatically handling tasks for us. Nonetheless, in the end we opted for Python since it is free (unlike MATLAB), thereby portable, easy to set up and even more performant than the potential alternatives we tested. The SciPy Basic Linear Algebra Subprograms (BLAS) implementation in particular proves to be very CPU efficient, making it the fastest when estimating runtime at 100% CPU

Implementation	Time [s]	CPU load [%]	Time · CPU load [%]
MATLAB	2.2	100	2.2
C++ Eigen	3.52	54	1.9
NumPy	5.65	73	4.12
NumPy + Numba (JIT)*	4.55	55	2.5
PyTorch	3.41	58	1.98
NumPy Init + PyTorch	3.26	50	1.63
SciPy BLAS	4.3	32	1.38

* Just-in-time
Benchmarked with Stern et al.'s [1] example dataset of dimensions $T \times P : 200 \times 50$

Table 2: Different implementations of **Convar** algorithm we built to compare performance and their benchmarks. The Time · CPU load column describes a heuristic that serves the purpose of approximating performance at 100% CPU load.

load (4th column, Table 2).

To fully utilize the CPU we use Python's own multiprocessing library, creating **Convar** child processes that work in parallel from our main process. Table 3 shows benchmarks for the two top contending implementations from Table 2. We will be using the SciPy BLAS implementation from here on out which is ~84% faster than the original MATLAB implementation.

Implementation	Time total [s]	Time per Convar [s]
NumPy Init + PyTorch	27	1.69
SciPy BLAS	19	1.19

Benchmarked with Stern et al.'s [1] example dataset of dimensions $T \times P$: 200×50

Table 3: Implementations of the **Convar** algorithm that performed best in Table 2 Time · CPU load column. We force full CPU load by executing **Convar** 16 times in parallel. Using SciPy BLAS procedures and all computational resources at hand almost halves **Convar** execution time in comparison to MATLAB (Table 2). In the following subchapter we show how this is applicable to incoming data.

3.2 Chunking

Analyzing the complexity of operations involved in **Convar** (Table 1) we observe our fluorescence trace length T to play a significant role, having square and cubic scaling dependencies in matrix multiplication and matrix inversion, respectively. The number of traces P on the other hand only has linear dependencies in respect to the complexity of every operation it is involved in. Both T-chunking and P-chunking has its own use case.

In the previous chapter we showed how utilizing the CPU to its capacity can improve execution

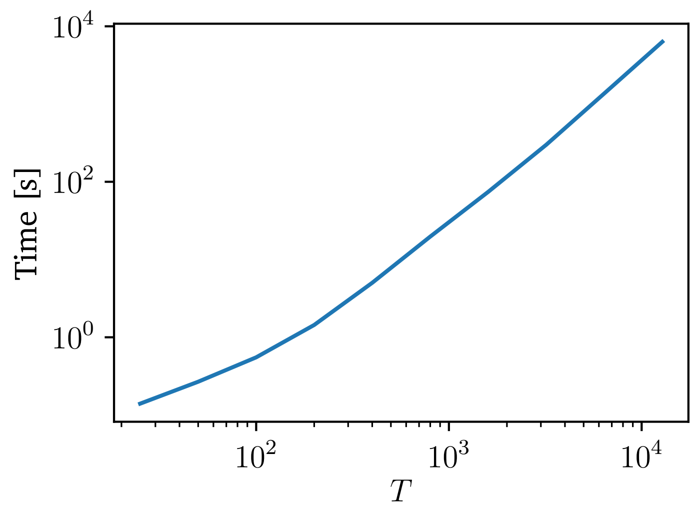


Figure 4: Single threaded **Convar** benchmarks using simulated data with static $P = 400$, doubling T from 25 to 12800, interpolating the graph in between benchmarks.

time of CPU efficient algorithms, executing them in parallel to keep full load. Needless to say, executing x many **Convars** for x different datasets to accelerate execution time seems counter-intuitive if we are only interested in one particular dataset. If we chunk P however we are able to divide one dataset into d many chunks, executing them in parallel to achieve full CPU load. To clarify, since P scales linearly we do not gain any advantages from shortening it per **Convar** execution. If anything, the overhead introduced in the increased number of function calls and parallel initialization slightly deteriorates our runtime but as seen in Table 3 full CPU load far outweighs these drawbacks and chunking P to achieve it is the default setting in our version of Stern et al.'s[1] program.

Chunking T on the other hand does allow us to make runtime gains solely from a complexity perspective. In Figure 4 we examine **Convar**'s runtime behaviour with a static $P = 400$ and scaling T, observing a polynomial increase in runtime while increasing T linearly.

With this information, we assume that dividing one fluorescence trace of length T into d chunks of length T/d yields great runtime gains, given that our resulting deconvolved spiking rates are of similar quality to regular **Convar** runs. We confirm this by first examining potential runtime gains of T-chunked data and then introducing an overlap at the start of each chunk to assure the quality of results from T-chunked **Convar** runs.

Depicted in Figure 5, we can observe the changes in runtime when using chunks. Here we use chunks of 400 and 1600 timesteps length while extrapolating linearly for T bigger than the respective chunk length. Example: Chunking a dataset of size $P = 400,\ T = 6400$, where executing **Convar** takes 1370 seconds, into 16 chunks of $T = 400$, which take 5 seconds each, allows us to execute **Convar** 16 times, taking $16 \cdot 5$ seconds $= 80$ seconds in sum.

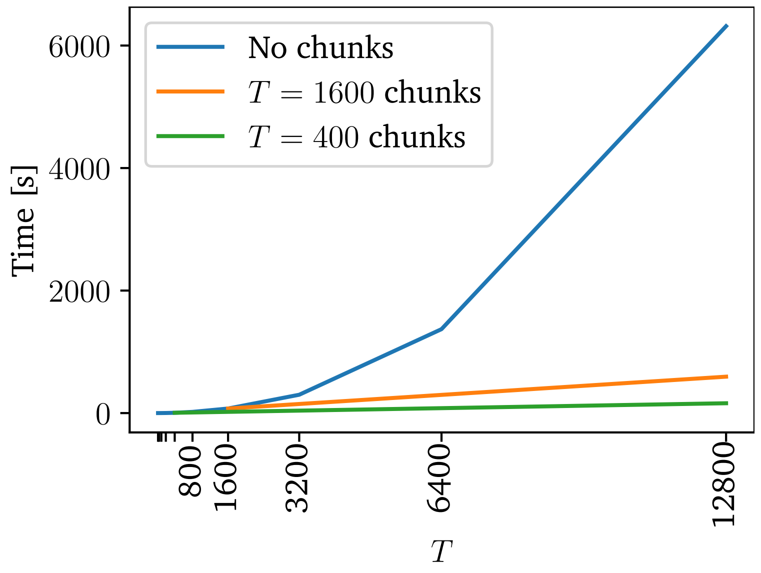


Figure 5: Single threaded **Convar** benchmarks using simulated data with static $P = 400$, doubling T from 25 to 12800, interpolating the graph in between benchmarks. In addition we show two graphs with chunked datasets while extrapolating their runtime to account for multiple **Convar** for data longer than chunk size of course.

To assure that the quality of our deconvolved spiking rates are similar to unchunked results we introduce a section at the start of each chunk that overlaps into the previous chunk. This overlap is essential for two reasons. Firstly, as described earlier r_1 is equivalent to $\bar{c}_1$ which would invalidate the crossing points between chunks. Secondly, the baseline fluorescence $\overline{\beta_0}$ is approximated as part of our minimization problem. Not including an overlap would mean that for a given chunk we would have to approximate $\overline{\beta_0}$ for timesteps included in the previous one. Using an overlap length large enough for calcium invoked by a spike at $t_{overlap} = 1$ to have mostly decayed until $t_{chunk} = 1$ solves both problems. Figure 1a gives us an overview of how much overlap we need at $\gamma_{40Hz} = 0.97$. The large dataset we test T-chunking on was recorded at 15 Hz so we adjust our gamma,

$$\gamma_{15Hz} = (0.97^{40})^{\frac{1}{15}} \approx 0.92. \tag{11}$$

Say that in this example we want the calcium to have decayed to less than 1% at the end of our overlap section,

$$\gamma_{15Hz}^{x} < 0.01 \tag{12}$$

$$\Leftrightarrow \quad x > \frac{\log 0.01}{\log \gamma_{15Hz}} \tag{13}$$

$$\Leftrightarrow \quad x > \sim 55.2, \tag{14}$$

rounding x up to 56 overlapping timesteps. At this point it is worth to be mentioned that our program

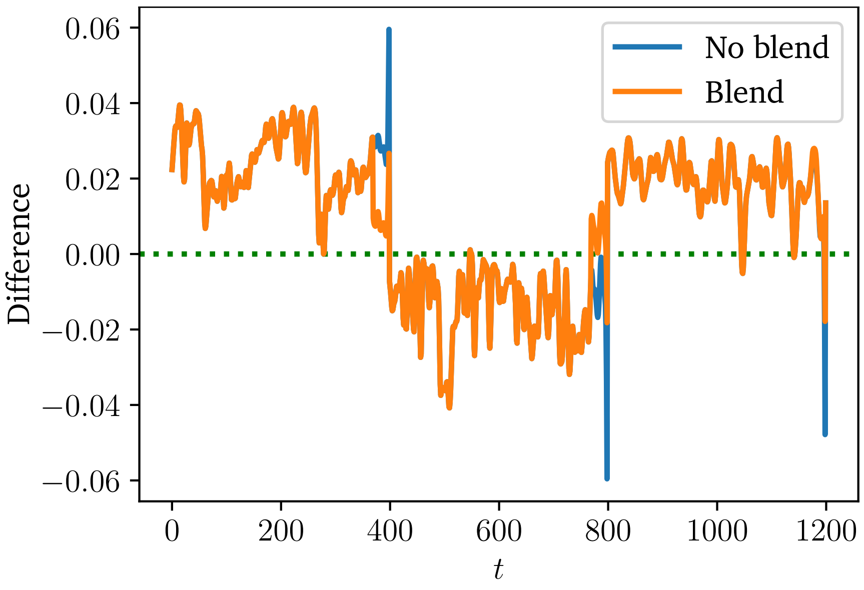


Figure 6: Comparing the difference between **Convar** run without chunks to $T = 400$ chunked run(mean of 10 neighbouring traces). We introduce a blending area to our overlap where the last b timesteps of overlap and previous chunk are averaged. The blue line includes no blending while the orange line uses $b = 30$ timesteps of blending.

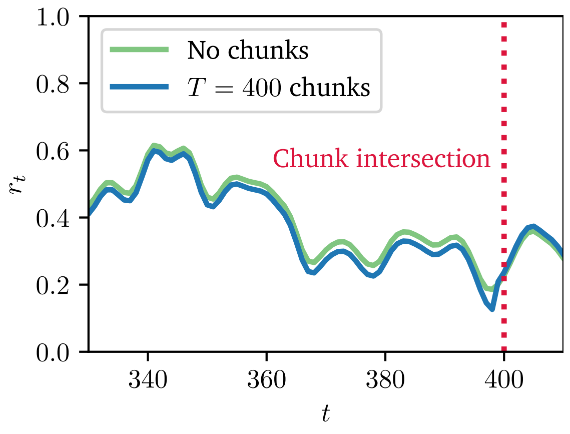
allows full customization via flexible parameterization on function call.

Comparing the mean difference of 10 neighbouring traces over time for a regular **Convar** call versus a T-chunked one with chunk size $T = 400$ and overlap size of $T = 56$, we can see slight increases in error around the timesteps right before crossing points where chunks are stitched together (Fig. 6, blue line). Taking a closer look we recognize these discrepancies to be mere scaling errors while spikes are correctly deconvolved (Fig. 7a). In Figure 7a we chunk the data with some overlap, deconvolve the overlap+chunk, then discard the overlap when chunks are stitched together. To ease the scaling error around intersections we introduce a blending area before intersections where the last b timesteps of overlap and previous chunk are averaged. Here, we choose $b = 30$ timesteps to blend (Fig. 6, orange line and Fig. 7b).

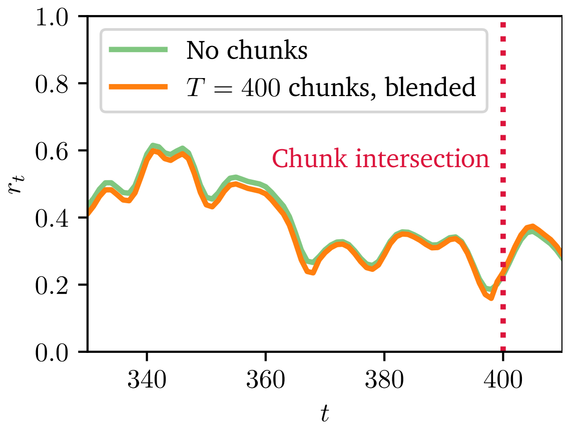
All in all, T-chunking large data is largely beneficial in terms of performance, allowing us to deconvolve long time series with reasonable time investment by avoiding the bottleneck caused by the **Convar** algorithm's complexity.

3.3 Early Stopping

Accelerating the speed of involved processes only gets us so far in our pursuit of fast deconvolution. What we need to accomplish to further speed up the **Convar** algorithm is reducing the amount of tasks that are done on our way to well deconvolved spiking rates. As described in Chapter 2.3, **Convar** is built on a proximal gradient descent algorithm proposed by Parikh and Boyd[18] where



(a) **Convar**: No chunks vs. T-chunks.



(b) **Convar**: No chunks vs. T-chunks, blended.

Figure 7: Taking a closer look at the deconvolved data used in Figure 6 and comparing the resulting spiking rates from **Convar** with non-chunked to $T = 400$-chunked inputs and $T = 56$ overlap. We introduce a blending area to our overlap where the last b timesteps of overlap and previous chunk are averaged. (a) includes slight scaling errors while the $b = 30$ timesteps of blending in (b) help remedy this problem.

in each iteration a gradient towards a global optimum is approximated, multiplied by a step size (or learning rate) and finally added to our return set of spiking rates r. In this, the number of iterations is set in a static manner, being defined at function call.

We propose early stopping, a more dynamic way of determining how many iterations are necessary for satisfactory results, taking inspiration from machine learning techniques[19]. In contrast to machine learning however we do not have labels to our data nor a resulting cost function to assess the quality of our resulting spiking rates r. Instead we examine the approximated gradient by deploying a metric function, applying it to the gradient every iteration i, and checking if $metric(gradient)_i$ is less than a threshold set at function call, breaking the **Convar** algorithm and returning results if true. We choose our metric function to be $metric(gradient) = mean(|gradient|)$.

Depicted in Figure 8 we can see the convergence behaviour of $metric(gradient)$ over the 10000 iterations that were suggested in Stern et al.'s[1] code. In this figure, the dotted red line describes the early stopping threshold, set to 10^{-6}*. Breaking the main loop when the $metric(gradient)$ of a **Convar** run using the previously explained Stern et al.'s[1] implementation (Fig. 8, blue line) intersects with the early stopping threshold line saves a considerable amount of iterations. The optimization measures presented in subsequent sections are specifically aimed towards reducing the amount of work, mainly by accelerating convergence of $metric(gradient)$ to reach early stopping in less iterations.

*Both the metric function and early stopping threshold are customizable on function call.

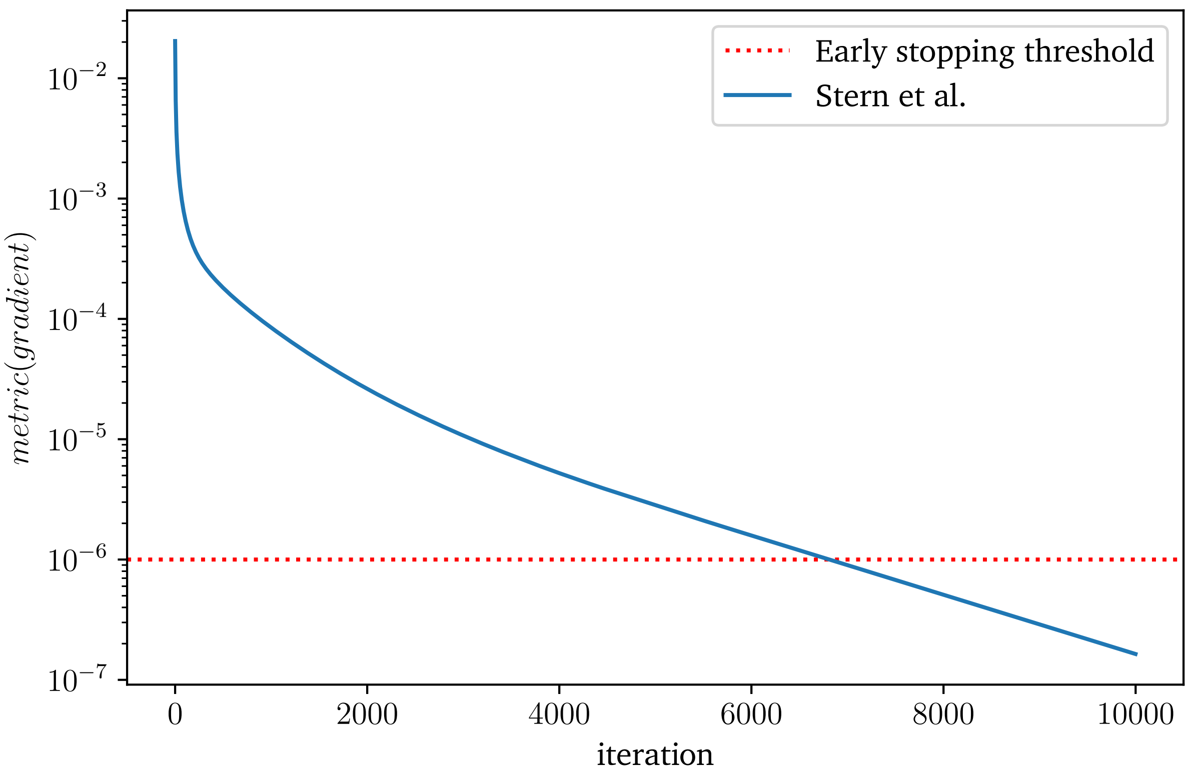


Figure 8: Observing $metric(gradient)$ over iterations on **Convar** deconvolution runs using Stern et al.'s[1] example dataset with dimensions $T \times P$: 400×50. The blue line shows Stern et al.'s[1] original implementation (translated from MATLAB to Python, identical functionality) where the solution to the optimization, or rather the desired spiking rates r are initialized randomly without changes to the learning rate over iterations.

3.4 First Differences Deconvolution and Meaningful Initialization

In Chapter 2.3 we described the **Convar** algorithm in depth. While examining further ways the algorithm could be optimized, we recognized the random initialization of the soon-to-be deconvolved spiking rates r in Step 3 to be improvable. As a replacement, one particular method, that is explained and compared to **Convar** in Stern et al.[1], stands out for its simplicity and performance, called *First Differences Deconvolution* (**Firdif**).

A very substantial difference between **Firdif** and **Convar** is that **Firdif** assumes the measured fluorescence to **be** the underlying calcium, so $\overline{y} = \overline{c}$. Given this assumption and Model 5, we can retrieve the spiking rates with

$$r_t = \overline{y}_t - \gamma \overline{y}_{t-1}, \qquad t = 2, ..., T \tag{15}$$

by iterating through a fluorescence trace once, thus only spending $T - 1$ subtractions worth of time which is incredibly little compared to **Convar**.

Firdif lacks most substantially in that it does not factor in both the error ϵ and the fact that in wide-field calcium imaging the spiking rates stay fairly similar in neighbouring timesteps ($r_t \approx r_{t-1}$, explained in Chapter 2.3), expressed by the penalty and its corresponding weight λ in **Convar**. To dampen these deficits, Stern et al.[1] propose using moving average smoothing in **Firdif**, somewhat resembling the effect that the penalty function has in **Convar** although they do not include a process that adjusts the degree of smoothing dependent on the incoming dataset like λ-**Search**. Moving average smoothing works by iterating through an array, sliding a 'window' with an odd number window size ω over each index i and using the average of the values inside the window ($\frac{\omega-1}{2}$ indices left and right of i) as the smoothed i in the output array. In words, the window size ω determines the degree of smoothing with a larger ω increasing the smoothing effect.

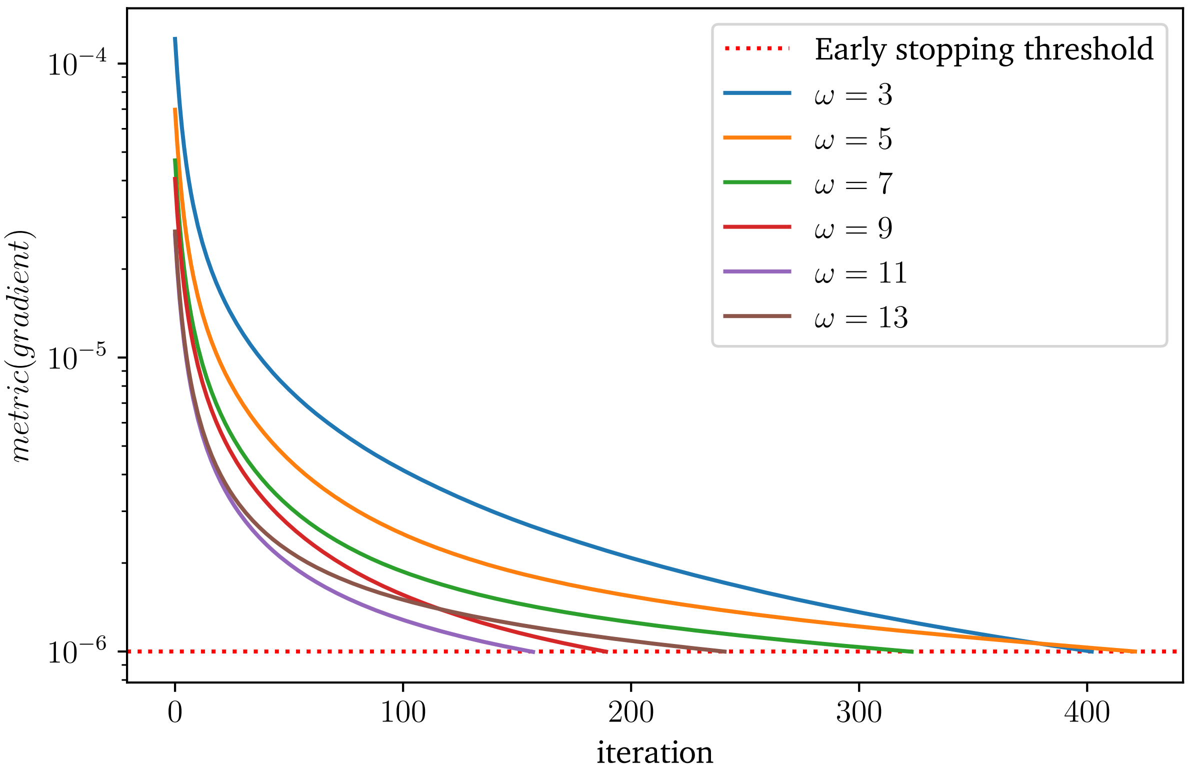


Figure 9: A comparison of different **Convar** runs and their $metric(gradient)$ progressions when being initialized with **Firdif** varying moving average window sizes ω. We implemented an algorithm similar to Stern et al.'s[1] λ-**Search** that finds the best ω for **Firdif**. Our algorithm correctly identifies the best $\omega = 11$ beforehand. The effect a well chosen ω can have on gradient convergence is displayed in this figure.

We transfer the logic behind Stern et al.'s[1] λ-**Search** (also explained in Chapter 2.3) to **Firdif**, proposing an analogous ω-**Search** where we adjust the window size ω of the moving average smoothing accordingly. Depicted in Figure 9 we see a direct comparison of the effect different ω can have in **Firdif** when initializing spiking rates r of **Convar**. Our ω-Search algorithm correctly identifies

$\omega = 11$ to be the best ω before executing **Convar**. Similar to λ-**Search** the cost for ω-Search is executing **Firdif** on each potential ω. However, as described above **Firdif** is inexpensive to execute, having a complexity of $O(TP)$, and as such the ω-Search algorithm consumes vanishingly little time compared to the amount of iterations it saves on **Convar** runs. If no best ω from the ω-Search is passed into the **Convar** function we use little smoothing as a default, picking a conservative $\omega = 3$.

Looking at the orange line in Figure 11 we can see the beneficial impact of initializing r with the **Firdif** algorithm in Step 3. As already mentioned, the amount of necessary tasks for **Firdif** is incredibly low compared to how much it speeds up the converge of $metric(gradient)$ when initializing r with it.

3.5 Adaptive Learning Rate

Another improvable component of the **Convar** algorithm as described by Stern et al.[1] is the step size s, referred to as the learning rate in machine learning, which is the prefactor of the approximated gradient when added to our output set r each iteration. In this, the learning rate is initialized in Step 2 and does not change its value at all during function execution which we aim to alter in order to accelerate $metric(gradient)$ convergence.

As described earlier, the **Convar** algorithm is based on a proximal gradient descent algorithm proposed by Parikh and Boyd[18] applied to the minimization problem stated in Equation (8). The specific value our learning rate s is assigned adheres to Parikh and Boyd's[18] specification to keep $s \leq \frac{1}{\phi}$ to ensure convergence where ϕ is the Lipschitz constant of our gradient function $grad\ f(r)$, used in Step 4a of the **Convar** algorithm.

We implemented a simple method to modify the learning rate s every iteration, basing the specific modification of s on the $metric(gradient)$ trend. Here, we reuse information already calculated each iteration i in our early stopping method, namely the $metric(gradient)_i$, and save it for the subsequent iteration. Consequently, comparing $metric(gradient)_i$ with $metric(gradient)_{i-1}$ we adjust the learning rate based on whether the change is desirable or not. We increase the learning rate when said change was desirable ($metric(gradient)$ decreases) and decrease the learning rate otherwise. We call this way of altering the learning rate over iterations *adaptive*.

If we went by Parikh and Boyd's[18] specification on initializing the learning rate and did not change it during **Convar** execution, convergence would be ensured and we would always be heading towards a global optimum[18]. Using our approach however we do cross the learning rate threshold $s \leq \frac{1}{\phi}$ specified by Parikh and Boyd[18] when increasing s, meaning that convergence is not guaranteed and we are at risk of divergence. To inhibit divergent behaviour of the gradient we found that a learning rate decrease is mandatory and must be significantly greater than the increase. We use an increase of $s \cdot c$ and a decrease of $s \cdot c^{-x}$ with $c, x > 1$ and our learning rate s initialized just as described in Step 2 of the algorithm. The default setting in our program uses $c = 1.01$ and $x = 10$, observed to be stable and very beneficial for convergence (Fig. 11, green line).

Furthermore we found that an instant increase of the learning rate s accelerates convergence as well, seen in Figure 11, purple line. We deploy the instant increase right after s is initialized in Step 2 and use $s_{instant\_increased} = s_{init} \cdot f(\lambda)$ with the instant increase function

$$f(\lambda) = \begin{cases} 1.5, & \text{if } \lambda < 0.1 \\ \frac{\lambda - 0.1}{1 - 0.1} * (4 - 1.5) + 1.5, & \text{if } 0.1 \leq \lambda \leq 1 \\ 4, & \text{if } 1 < \lambda \end{cases} \tag{16}$$

which is purely based on various trials that have proven its effectivity. The middle case of $f(\lambda)$ scales its return value to the interval $[1.5, 4]$ in a linear fashion, proportional to $\lambda \in [0.1, 1]$.

Summing up, both the adaptive learning rate and the thereby enabled instant increase are vastly beneficial for gradient convergence, further accelerating the execution time of our modified version of Stern et al.'s[1] **Convar** algorithm.

3.6 Binary λ-Search

The λ-**Search** process has always proven to be a cumbersome operation since we had to compare the results of $\#L$ many **Convar** executions where $\#L$ is the number of elements in a set L of potential λ. Stern et al.[1] had already provided us with advice on how to accelerate the λ-**Search**, only using a fraction of the experiment length in the associated **Convar** executions and using *that* best λ for further data with similar setup (this also applies to ω-**Search**, Chapter 3.4). In addition to that, exploring different options we designed an algorithm where the number of **Convar** executions is minimized and the accuracy of the eventually determined 'best' λ is maximized by increasing granularity of potential λ. We call this algorithm **Binary** λ-**Search**. However, during testing we found out that the assumption we built this algorithm on is not true. Thus the algorithm is *not universally valid*.

As explained in Chapter 2.3, Stern et al.'s[1] λ-**Search** works by dividing a given dataset into even and odd timesteps, picking one of the halves, then deconvolving it with every one of the potential $\lambda \in L$. We pick the odd traces in this example. After deconvolution we reconstruct the odd traces from the resulting deconvolved spikes, as seen in Equation (10), and compare it to the even traces by using the mean absolute difference between them as a score. The lowest score determines the best reconstruction and thus the best λ.

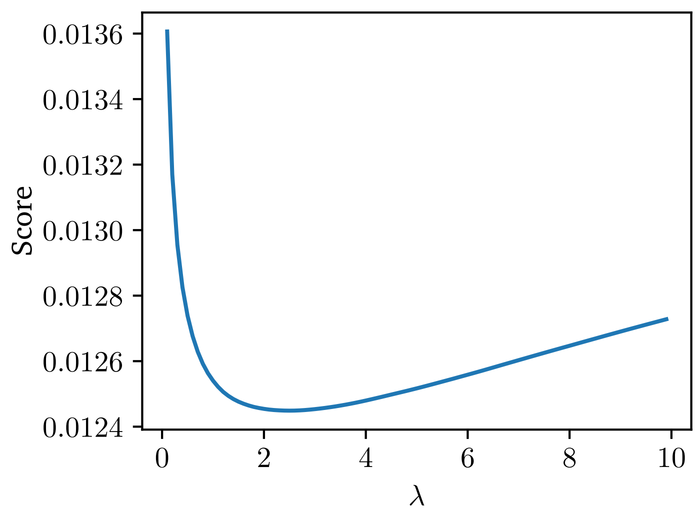


Figure 10: λ-**Search** scores for a list of potential $\lambda \in \{0.1, 0.2, ..., 10\}$ where 2.5 is the true best lambda. This example demonstrates that two different λ equidistant from 2.5 do not have the same score. As a result our **Binary** λ-**Search** algorithm might return a 'best' λ that is slightly off the true best λ. Although this has not been the case in any of the datasets we trialed it is a possibility.

To demonstrate the assumption we based **Binary** λ-**Search** on and how it is invalid, let 2.5 be the true best λ for a given dataset. This means that in λ-**Search** the score between the recreation and unchanged half is minimal for $\lambda = 2.5$ among all $\lambda \in \mathbb{R}$. Now let us take two different λ, equidistant from 2.5. We assumed that these two λ should have an equal or at least very similar score and built the **Binary** λ-**Search** algorithm based on that. Unfortunately, as can be seen in Figure 10 this is not the case.

Binary λ-Search algorithm

Given the signal $\overline{y}$, the calcium decay rate γ and a sorted array L of evenly spaced potential λ, do:

1. Iterate until $\#L == 1$:
   (a) Halve L into left and right side.
   (b) Assign middle λ of respective side to $left\ mid$ and $right\ mid$.
   (c) Regular Stern et al.[1] λ-**Search** algorithm with only $left\ mid$ and $right\ mid$.
   (d) Discard the half containing the worse λ.
   (e) Append better λ score to cache.
2. Last element in L is the best λ.

Nevertheless, the **Binary** λ-**Search** algorithm is useful in quickly and efficiently determining a good λ since its runtime is $O(\log \#L \cdot \textbf{Convar})$ as opposed to $O(\#L \cdot \textbf{Convar})$ with the regular λ-**Search**. We also used a cache for λ that have already been evaluated throughout the algorithm, further reducing the number of necessary **Convar** execution. It is noteworthy that although our initial assumption explained earlier is not true, we have not found any errors in determining the best λ with **Binary** λ-**Search** as opposed to executing **Convar** on every $\lambda \in L$ with the regular λ-**Search**.

4 Results

In Chapter 3 we proposed various methods of optimization that build upon Stern et al.'s[1] **Convar** algorithm, resulting in our modified version of the program. Among these are changes to the implementation (Chapter 3.1) where we benchmarked and compared different implementations and their underlying linear algebra libraries, picking Python's SciPy library with its wrapped BLAS functions in the end. Besides that we introduce chunking (Chapter 3.2) of datasets, allowing us to prevent a large complexity bottleneck (large T immensely hinders performance $\Rightarrow$ T-chunking) and/or fully utilize the CPU in conjunction with multiprocessing (P-chunking). The implementation changes and P-chunking speed the **Convar** algorithm up by almost two times when compared to the original MATLAB version included in Stern et al.'s[1] code while T-chunking ensures **Convar** execution time to increase in a linear fashion instead of polynomial for long time series.

In addition to that, we introduced early stopping (Chapter 3.3) where we analyze the gradient with a metric function in each iteration and break the main loop when $metric(gradient)$ reaches a certain threshold set via function argument. Since the main loop with its numerous iterations is the primary reason for **Convar** to be a time-consuming function, cutting off iterations is incredibly valuable. Consequently, optimization measures like **Firdif** initialization (Chapter 3.4) and our adaptive learning rate (Chapter 3.5) that accelerate $metric(gradient)$ convergence (which directly results from converge of spiking rates r) are essential to the **Convar** algorithm's performance, yielding a 100 times speedup.

Taking a look at Figure 12, we can see the impact our optimization features have on $metric(gradient)$ convergence and thus performance since we need less iterations for equivalent results. Table 5 provides exact statistics to Figure 12, showing the exact iteration of early stopping and time taken in seconds. Compared to not using early stopping at all we see a speedup of more than 150 times. When only comparing the time spent in the main loop (ignoring constant overhead) we reach about 270 times speedup. It is noteworthy that, although only slightly, the time per iteration increases with every added optimization measure. However, the gain in total execution time outweighs this drawback since we save a considerable amount of iterations.

It is also noteworthy that the resulting spiking rates r are of similar or better quality, depending on the value of the early stopping threshold, since every local optimum is also a global optimum[17]. This means that the accuracy of the approximation is dependent on how close the gradient is to convergence. This means that our version of the program actually improves the quality of the deconvolved spikes by accelerating convergence. Looking at Figure 11 we can see that $metric(gradient)$ would not intersect with an early stopping threshold at 10^{-7} within 10000 iterations when using Stern

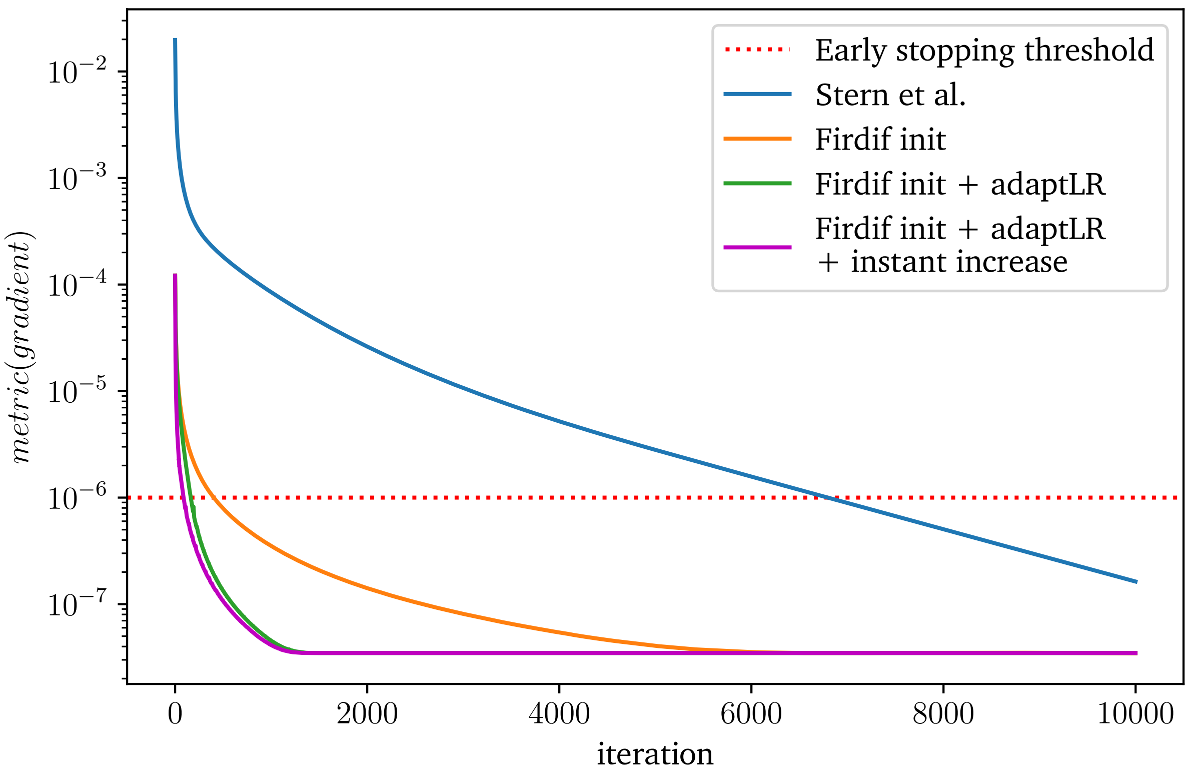


Figure 11: Observing $metric(gradient)$ over iterations with different optimization features on **Convar** deconvolution runs using Stern et al.'s[1] example dataset with dimensions $T \times P : 400 \times 50$. The blue line shows Stern et al.'s[1] original implementation (translated from MATLAB to Python, identical functionality) where the solution to the optimization, or rather the desired spiking rates r are initialized randomly without changes to the learning rate over iterations. The other lines enable optimization features successively (refer to legend) using the **Firdif** initialization for r (Chapter 3.4), adaptive learning rate and an instant increase to the learning rate on initialization (Chapter 3.5). In the end enabling all optimization features is most beneficial to **Convar** execution time.

Optimization features	**Early stopped at i**	**Time total [s]**
Stern et al.[1]	(10000)	13.86
Stern et al.[1] with early stopping	6831	9.56
Firdif init	402	0.59
Firdif init + adaptLR	163	0.23
Firdif init + adaptLR + instant increase	89	0.13

Data in the third column (Time total [s]) is rounded to the second decimal place.
Each benchmark contains about 0.04 s of constant overhead.

Table 4: Statistics of Figure 11 including the exact iteration i of early stopping and execution time.

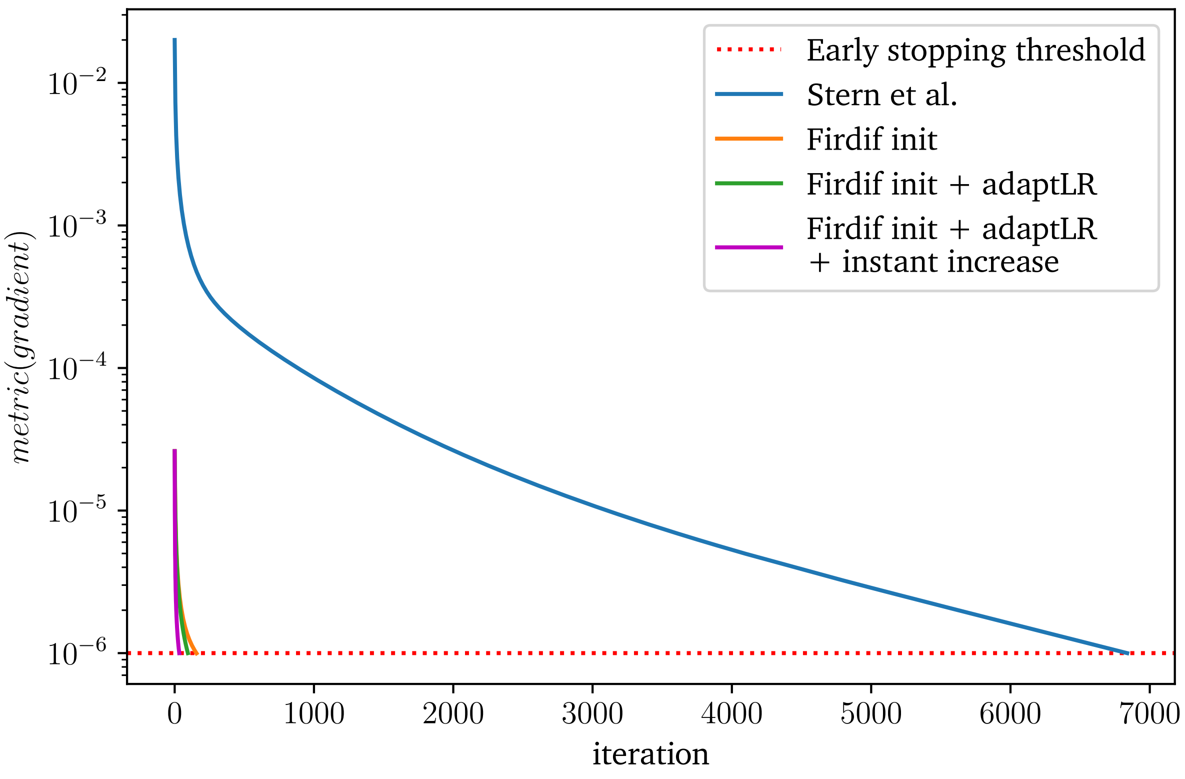


Figure 12: Similar to Figure 11, this figure shows $metric(gradient)$ over iterations with different optimization features on **Convar** deconvolution runs using Stern et al.'s[1] example dataset with dimensions $T \times P : 400 \times 50$. Expanding on Figure 11, the early stopping threshold is highlighted by cutting lines off when they intersect. Additionally we used our ω-**Search** algorithm (explained in Chapter 3.4, built on the idea of Stern et al.'s[1] λ-**Search** algorithm) beforehand to find the best window size ω (similar to a degree of smoothing, like λ) for **Firdif**. This cost about a tenth of a second but reduces the number of iterations until early stopping down to almost one third (as seen when comparing Table 11 with Table 12).

Optimization features	**Early stopped at i**	**Time total [s]**
Stern et al.[1]	(10000)	13.86
Stern et al.[1] with early stopping	6831	9.56
Firdif init	157	0.28
Firdif init + adaptLR	95	0.18
Firdif init + adaptLR + instant increase	33	0.09

Data in the third column (Time total [s]) is rounded to the second decimal place.
Each benchmark contains about 0.04 s of constant overhead.

Table 5: Statistics of Figure 12 including the exact iteration i of early stopping and execution time.

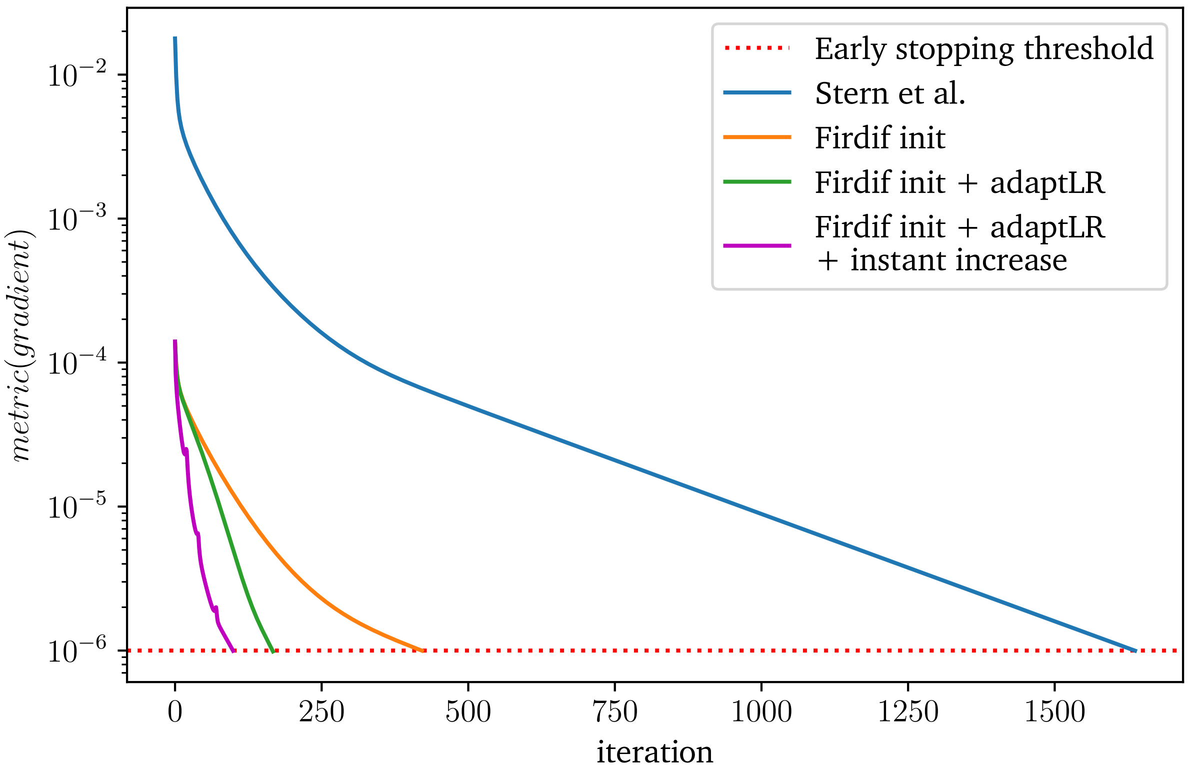


Figure 13: This figure shows $metric(gradient)$ over iterations with different optimization features on **Convar** deconvolution runs using one of our own datasets with dimensions $T \times P : 400 \times 10000$. The ω-**Search** algorithm has been executed beforehand and cost about 1.62 seconds. The notches seen in the purple line are a product of the learning rate increasing too fast, bordering on divergence, when the learning rate decrease inhibits this behaviour (as described in Chapter 3.5).

Optimization features	**Early stopped at i**	**Time total [s]**
Stern et al.[1]	(10000)	916.41
Stern et al.[1] with early stopping	1637	150.11
Firdif init	421	38.86
Firdif init + adaptLR	167	17.19
Firdif init + adaptLR + instant increase	99	10.23

Data in the third column (Time total [s]) is rounded to the second decimal place.
Each benchmark contains about 0.09 s of constant overhead.

Table 6: Statistics of Figure 13 including the exact iteration i of early stopping and execution time.

et al.'s[1] original implementation while our implementation intersects with that specific threshold in less than 500 iterations. Although, changes at this level of granularity might not matter to the end user.

Figure 13 shows **Convar** performance on one of our own datasets. Looking at the statistics in Table 6 we see a speedup of about 90x between Stern et al.'s[1] and our implementation. This example also displays how powerful the speedup can be in terms of absolute time gain, saving us about 15 minutes of processing time. The general difference in processing time between Stern et al.'s[1] and our dataset can be explained through the much higher number of fluorescence traces $P = 10000$ included. While the speedup factor with this dataset is lower compared to Stern et al.'s[1] example dataset, the absolute time gain is still very significant in making **Convar** a performant algorithm.

5 Discussion

Stern et al.'s[1] proposed algorithm for deconvolution of wide-field calcium imaging recordings is the first work to deal with spiking rate inference in the specific context of wide-field microscopy. It allows approximation of underlying neural activity, producing great results especially when comparing it to using methods not optimized for wide-field microscopy on these recordings[1]. That being said, its performance struck our attention as something we could improve on.

We resettled Stern et al.'s[1] original MATLAB implementation into Python, adopting the algorithms functionality while establishing a number of optimization features. It is noteworthy that we did not alter the original implementation's core functionality and still completely rely on Stern et al.'s[1] models, building our optimizations on top of the existing algorithm.

Something important to bear in mind when comparing implementations (Table 2 & 3) is that although the bottom five rows of Table 2 are Python implementations, which is considered to be slower than a high performance language like C/C++, the libraries used for linear algebra operations are written in C. NumPy uses native C datatypes and merely acts as an *application programming interface* (API), allowing us to utilize wrapped C procedures and datatypes from Python. SciPy takes advantage of NumPy data structures while implementing additional procedures, including portable precompiled wrappers for the highly performant BLAS library, saving us the trouble of linking and compiling BLAS with, for instance, NumPy on every system. We originally intended to use PyTorch with CUDA, calculating on the GPU instead of CPU. The involved cost of transporting data to GPU memory however outweighs the benefit of potentially faster matrix multiplication since, being able to chunk both T and P (Chapter 3.2), we do not work with big matrices which is where GPUs would potentially outperform CPUs. Still, PyTorch poses a viable option as its runtime on CPU is satisfactory and is a suitable solution if one would want to easily switch between CUDA and CPU implementations by changing only a few arguments.

Stern et al.[1] use a proximal gradient descent algorithm[18] to deconvolve the measured fluorescence and infer the desired neural activity. In this, they approximate a gradient in each iteration of the algorithm's main loop. This process is lengthy because iterations are both costly and manifold. Most of our speedup in general cases originates from cutting down on the number of necessary iterations by accelerating convergence of gradient descent. We accomplished this in essence by meaningfully initializing the output set as opposed to randomizing it (Chapter 3.4) and adapting the learning rate each iteration dependent on gradient trend (Chapter 3.5). Paired with early stopping the algorithm when the gradient has become sufficiently small (Chapter 3.3), crossing a pre-set threshold, we achieved satisfactory runtimes. It is noteworthy that our changes do not deteriorate the results of the deconvolution but rather improve them by accelerating gradient convergence. Since every local optimum is also a global optimum[17] we effectively improved the accuracy of approximated spiking rates. We also provided a way to deal with long datasets (Chapter 3.2) and proposed an alternative to Stern et al.'s[1] tuning parameter search algorithm involved in the deconvolution. In addition to all this we examined high performance linear algebra procedures, written in C and wrapped in Python (Chapter 3.1), which we employ in our finished program.

In direct comparison to Stern et al.'s[1] MATLAB implementation our program is almost two times faster from a mere implementation perspective and another 100 times faster on average when using the optimization features aimed at gradient descent, totaling an increase in speed of roughly 200 times on average. All in all, we do not see a disadvantage to using our implementation since we offer a wide range of customization (that even allows to completely disable every one of our optimization features), a highly portable, lightweight implementation using a free programming environment and several features aimed at accelerating the deconvolution of wide-field calcium imaging data.

6 Conclusion

The goal of this work was to optimize the deconvolution algorithm proposed by Stern et al.[1] aimed at retrieving underlying neural activity from recordings that utilize wide-field calcium imaging techniques. As a result we were able to achieve a speedup of about 100 times on average with universally usable optimization measures. The system-specific optimizations we made net us another speedup of almost two times on our machine by employing high performance libraries and utilizing given resources to the fullest, raising our factor of total speedup to around 200 times. Furthermore we presented a way to handle large data, eliminating another bottleneck posed by the complex nature of the algorithm. The final product of this work is an optimized version of Stern et al.'s[1] MATLAB release, written in Python and packed into module structure, with full customizability over presented features and a high degree of portability including only two dependencies.

References

[1] Merav Stern, Eric Shea-Brown, and Daniela Witten. Inferring neural population spiking rate from wide-field calcium imaging. *bioRxiv*, 2020.

[2] Christine Grienberger and Arthur Konnerth. Imaging calcium in neurons. *Neuron*, 73(5):862–885, 2012. ISSN 0896-6273.

[3] William Allen, Isaac Kauvar, Michael Chen, Ethan Richman, Samuel J. Yang, Ken Chan, Viviana Gradinaru, Ben Deverman, Liqun Luo, and Karl Deisseroth. Global representations of goal-directed behavior in distinct cell types of mouse neocortex. *Neuron*, 94:891–907.e6, 05 2017.

[4] Tsai-Wen Chen, Nuo Li, Kayvon Daie, and Karel Svoboda. A map of anticipatory activity in mouse motor cortex. *Neuron*, 94:866–879.e4, 05 2017.

[5] Hiroshi Makino, Chi Ren, Haixin Liu, An Na Kim, Neehar Kondapaneni, Xin Liu, Duygu Kuzum, and Takaki Komiyama. Transformation of cortex-wide emergent properties during motor learning. *Neuron*, 94(4):880–890.e8, 2017. ISSN 0896-6273.

[6] Gordon B. Smith, Bettina Hein, David Whitney, D. Fitzpatrick, and M. Kaschube. Distributed network interactions and their emergence in developing neocortex. *Nature neuroscience*, 21: 1600 – 1608, 2018.

[7] Tsai-Wen Chen, Trevor Wardill, Yi Sun, Stefan Pulver, Sabine Renninger, Amy Baohan, Eric Schreiter, Rex Kerr, Michael Orger, Vivek Jayaraman, Loren Looger, Karel Svoboda, and Douglas Kim. Ultrasensitive fluorescent proteins for imaging neuronal activity. *Nature*, 499:295–300, 07 2013.

[8] Joshua T. Vogelstein, Brendon O. Watson, Adam M. Packer, Rafael Yuste, Bruno Jedynak, and Liam Paninski. Spike Inference from Calcium Imaging Using Sequential Monte Carlo Methods. *Biophysical Journal*, 97(2):636–655, July 2009.

[9] Johannes Friedrich, Pengcheng Zhou, and Liam Paninski. Fast online deconvolution of calcium imaging data. *PLOS Computational Biology*, 13(3):1–26, 03 2017.

[10] Sean Jewell and Daniela Witten. Exact spike train inference via ℓ_0 optimization. *The Annals of Applied Statistics*, 12(4):2457 – 2482, 2018.

[11] Sean W Jewell, Toby Dylan Hocking, Paul Fearnhead, and Daniela M Witten. Fast nonconvex deconvolution of calcium imaging data. *Biostatistics*, 21(4):709–726, 02 2019.

[12] E. Pnevmatikakis, Daniel Soudry, Yuanjun Gao, Timothy A. Machado, J. Merel, David Pfau, T. Reardon, Yu Mu, Clay Lacefield, Weijian Yang, M. Ahrens, R. Bruno, T. Jessell, D. Peterka, R. Yuste, and L. Paninski. Simultaneous denoising, deconvolution, and demixing of calcium imaging data. *Neuron*, 89:285–299, 2016.

[13] Gergely Silasi, Dongsheng Xiao, Matthieu Vanni, Andrew Chen, and Timothy Murphy. Intact skull chronic windows for mesoscopic wide-field imaging in awake mice. *Journal of neuroscience methods*, 267, 04 2016.

[14] Aaron M. Kerlin, Mark L. Andermann, Vladimir K. Berezovskii, and R. Clay Reid. Broadly tuned response properties of diverse inhibitory neuron subtypes in mouse visual cortex. *Neuron*, 67 (5):858–871, 2010. ISSN 0896-6273.

[15] Kelly B. Clancy, Ivana Orsolic, and Thomas D. Mrsic-Flogel. Locomotion-dependent remapping of distributed cortical networks. *bioRxiv*, 2018.

[16] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996. ISSN 00359246.

[17] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. doi: 10.1017/CBO9780511804441.

[18] Neal Parikh and Stephen Boyd. Proximal algorithms. *Foundations and Trends® in Optimization*, 1(3):127–239, 2014. ISSN 2167-3888.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.