Deep Residual Auto-Encoders for Expectation Maximization-based Dictionary Learning

Bahareh Tolooshams, *Student Member, IEEE,* Sourav Dey, *Member, IEEE,* and Demba Ba, *Member, IEEE*

***Abstract*—Convolutional dictionary learning has become a popular method for learning sparse representations from data. State-of-the-art algorithms perform dictionary learning through an optimization-based alternating-minimization procedure that comprises a sparse coding and a dictionary update step respectively. Here, we draw connections between convolutional dictionary learning and neural networks by proposing an architecture for convolutional dictionary learning termed the constrained recurrent sparse auto-encoder (CRsAE). We leverage the interpretation of the alternating-minimization algorithm for dictionary learning as an Expectation-Maximization algorithm to develop auto-encoders that, for the first time, enable the simultaneous training of the dictionary and regularization parameter. The forward pass of the encoder, which performs sparse coding, solves the E-step using an encoding matrix and a soft-thresholding nonlinearity imposed by the FISTA algorithm. The encoder in this regard is a variant of residual and recurrent neural networks. The M-step is implemented via a two-stage back-propagation. In the first stage, we perform back-propagation through the auto-encoder formed by the encoder and a linear decoder whose parameters are tied to the encoder. This stage parallels the dictionary update step in dictionary learning. In the second stage, we update the regularization parameter by performing back-propagation through the encoder using a loss function that includes a prior on the parameter motivated by Bayesian statistics. We leverage GPUs to achieve significant computational gains relative to state-of-the-art optimization-based approaches to convolutional dictionary learning. We apply CRsAE to spike sorting, the problem of identifying the time of occurrence of neural action potentials in recordings of electrical activity from the brain. We demonstrate on recordings lasting hours that CRsAE speeds up spike sorting by 900x compared to notoriously slow classical algorithms based on convex optimization.**

***Index Terms*—Dictionary Learning, Convolutional Sparse Coding, Auto-Encoders, Deep Residual Networks, Expectation-Maximization, Spike Sorting.**

I. Introduction

SPARSE coding has become a popular method for extracting features from data. Sparse coding problems assume the data $\mathbf{y}$ can be represented as a sparse linear combination of columns (features) of a matrix $\mathbf{H}$, termed a dictionary. Given the dictionary $\mathbf{H}$, methods such as orthogonal matching pursuit [1] and basis pursuit [2] find the sparse representation. The Lasso is an alternative method to estimate the sparse code by penalizing the ℓ_1-norm of the linear regression coefficients [3]. The performance of this regression highly depends on the choice of the regularization parameter. The sparse coding problem also has connections to Bayesian statistics, where Monte-Carlo Expectation-Maximization (EM) [4] can be used for estimation of hyper-parameters such as the regularization parameter [5], [6]. In this line of thought, treating the sparse codes as scale-mixtures of Gaussians makes it possible to obtain Gibbs samples from the missing data of interest, namely the sparse codes and the hidden variance of each of the Gaussians in the scale mixture. The M-step uses the Gibbs samples to approximate the intractable expectations necessary to update the hyper-parameters [6]. As with most Monte-Carlo methods, the Gibbs sampling step can be computationally expensive, particularly when the sparse codes are high-dimensional.

The dictionary in the sparse coding problem can be analytical (e.g. discrete wavelet transform [7]), have random structures (e.g. Gaussian random matrix), or be learned from data for a better adaptation. In the signal processing literature, dictionary learning (DL) is the de-facto method for learning the sparse representations in an unsupervised manner. The method of optimal directions [8] and K-SVD [9] are two examples of popular DL algorithms. In convolutional dictionary learning (CDL), which has drawn attention in signal and image processing [10] due to its ability to produce shift-invariant sparse representations [11], $\mathbf{H}$ has a block-Toeplitz structure consisting of the concatenation of a finite number of Toeplitz matrices, each corresponding to a convolutional filter (atom) [12], [13]. CDL is a non-convex optimization problem. A popular method to circumvent this non-convexity is to alternate between a convolutional sparse coding (CSC) step and a convolutional dictionary update step, until a convergence criterion is met, a procedure referred to as *alternating minimization* [14]. The state-of-the-art algorithms solve the CSC step through variants of ADMM [15]. While efficient, these algorithms lack a scalable infrastructure to solve the CSC step in parallel for thousands of examples.

A recent line of work has sought to draw connections between auto-encoders (AE)s and DL [16]–[19]. Expressing sparse coding and DL as a feed-forward neural network lets us take advantage of the parallelism offered by GPUs to speed up learning. In this line of work, the *encoder* defines a mapping from the input data space to a sparse vector similar to the sparse coding step in DL, and the *decoder*, imitating the dictionary update step, uses a set of filters (dictionary) to reconstruct the input data from the approximated sparse codes. We showed in our previous work [20] that, unless the weights of the encoder and decoder are tied, the architectures from [16] and [17] do not, strictly speaking, perform DL. That is, in order for the weights of an AE to be interpretable as convolutional filters in a CDL problem, encoder and decoder

B. Tolooshams and D. Ba is with the School of Engineering and Applied Sciences, Harvard University, Cambridge, MA, 02138 USA e-mail: (btolooshams@seas.harvard.edu; demba@seas.harvard.edu)

S. Dey is with Manifold AI, Oakland, CA e-mail:(sdey@manifold.ai)

Manuscript received April, 2019.

weights must necessarily be tied. In addition, we showed using simulated data that the AE introduced in [16] is not able to learn the true underlying convolutional dictionary because of a) encoder and decoder weights are not tied, and b) the codes produced by the encoder are not sparse enough to enable DL. In the context of dense (as opposed to convolutional) DL, the k-sparse AE [18] imposes the afore-mentioned constraint and uses a variant of iterative hard-thresholding [21] in its encoder to ensure that the encoder outputs sparse codes. The efficacy of limiting the code to be k-sparse through hard-thresholding is not clear in the convolutional case.

The next section, Section II, summarizes our contributions. In Section III, we introduce notation and the generative model that underlies CDL. We review classical CDL in Section IV. In Section V, we discuss the EM approach to the DL problem. We introduce a deep residual AE architecture motivated by the EM algorithm in Section VI and provide guidance on how to choose the hyper-parameters necessary for training the architecture. This is followed by Section VII in which we compare the EM-inspired deep residual AE to existing algorithms for CDL and apply it to spike sorting. We conclude in Section VIII.

II. Summary of Contributions

The contributions of this paper are as follows

A deep residual AE for CDL: We propose an AE architecture for convolutional dictionary learning (Figure 2). The encoder in this architecture is a variant of residual [22] and unrolled recurrent networks [17], [23] (Figure 1).

A training algorithm inspired by EM: We provide a prescription for simultaneously training the filters and the regularization parameter from the CDL objective function. This procedure is motivated by EM and Bayesian statistics. The encoder, which performs sparse coding, mimics the E-step of EM i.e. the sparse coding step in DL. The M-step corresponds to a two-stage back-propagation procedure (Figure 3). In the first stage, we perform back-propagation through the deep residual AE formed by the encoder and a linear decoder, a step that parallels the dictionary update step in DL. In the second stage, we perform back-propagation through the encoder using a loss function motivated by Bayesian statistics. This latter does not have an equivalent in DL, where the regularization parameter is typically not trained (Figure 5). To our knowledge, this two-stage procedure is novel.

An architecture that is interpretable and robust: We demonstrate, for the first time, that, when trained on data simulated from a convolutional generative model with known filters, this AE architecture motivated by CDL can a) successfully learn ground-truth filters (Figures 6 and 7), and b) is robust to noise at a range of signal-to-noise ratio (SNR) (Figure 4). That is, ours is a deep residual AE architecture for which the filters are interpretable as coming from a convolutional generative model.

An application to spike sorting: We show that the architecture can perform source separation when the support of the sources is much smaller than the observed data. We demonstrate this ability by separating the activity of single neurons in recordings of electrical activity from the brain of rats (Figure 9). The encoder from the architecture successfully performs spike sorting, namely identifies the location of action potentials from individual neurons in the recordings (Figure 10).

A demonstration of significant computational gains: We demonstrate that this approach, which can readily employ off-the-shelf tools for training networks on GPUs, is 5x faster than a state-of-the-art CDL algorithm based on convex optimization (Figure 8), and demonstrate that it can be used to perform spike sorting on hours of raw electrophysiological recordings 900x faster than existing optimization-based methods (Table IV).

III. Generative Model

A. Notation

The rest of our treatment will follow the notation and conventions summarized in Table I.

Symbol	Description
$\mathbf{H}$	Matrix (upper-case bold letters)
$\mathbf{h}$	Vector (lower-case bold letters)
$h_c[i]$	i^{th} element of the vector $\mathbf{h}_c$
$\mathbf{z}_t$	Vector $\mathbf{z}$ at the t^{th} iteration of an iterative algorithm
$z_t[n]$	n^{th} element of $\mathbf{z}_t$
$\mathbf{H}^{\text{T}}$	Transpose of $\mathbf{H}$
$\mathbf{h}^{\text{T}}$	Transpose of $\mathbf{h}$
$\mathbf{y}^j$	j^{th} training example (window of data)
$\|\|\mathbf{x}\|\|_p$	The ℓ_p-norm of vector $\mathbf{x}$
$\sigma_{\max}(\mathbf{H})$	Maximum singular value of the matrix $\mathbf{H}$
$\mathbf{I}$	The identity matrix
$\mathbb{I}$	The indicator function
$*$	Linear convolution
$i.i.d$	Independent and identically distributed
$\mathcal{N}(m, \sigma^2)$	Gaussian distribution with mean m and variance σ^2

TABLE I
Notation and conventions

B. Continuous-time Model

Consider a set of continuous-time filters $(h_c(t))_{c=1}^C$ localized in time where $C \in \mathbb{N}^+$ is the number of filters. Let $y(t)$ be the continuous-time signal that is the linear mixture of time-shifted and scaled versions of the filters $(h_c(t))_{c=1}^C$, and formally defined as

$$y(t) = \sum_{c=1}^{C} \sum_{i=1}^{N_c} x_{c,i} h_c(t - \tau_{c,i}) + v(t) \quad \text{for } t \in [0, T_0) \quad (1)$$

where $v(t) \overset{i.i.d}{\sim} \mathcal{N}(0, \sigma^2)$ is additive noise, $N_c \in \mathbb{N}$ is the number of appearances of filter c in the signal, and $x_{c,i} \in \mathbb{R}$ and $\tau_{c,i} \in \mathbb{R}^+$ encode, respectively, the amplitude and the position of the i^{th} appearance of filter c in the signal.

C. Discrete-time Model

Assuming $y(t)$ is sampled at the rate f_s, its discrete-time version $y[n]$ is given by

$$y[n] = \sum_{c=1}^{C}\sum_{i=1}^{N_c} x_{c,i}h_c[n-n_{c,i}] + v_n = \sum_{c=1}^{C} h_c[n] * x_c[n] + v[n] \quad \text{for } n = 1, \cdots, \left\lfloor \frac{T_0}{f_s} \right\rfloor, n_{c,i} = \left\lfloor \frac{\tau_{c,i}}{f_s} \right\rfloor \tag{2}$$

where, $x_c[n] = \sum_{i=1}^{N_c} x_{c,i}\delta[n-n_{c,i}]$ and $(h_c[n])_{n=0}^{K-1}$ is the discrete-time analog of $h_c(t)$, for $c = 1, \cdots, C$. We can express Eq. 2 in linear-algebraic form as follows

$$\mathbf{y} = \begin{bmatrix}\mathbf{H}_1 | \cdots | \mathbf{H}_C\end{bmatrix} \begin{bmatrix}\mathbf{x}_1 \\ \vdots \\ \mathbf{x}_C\end{bmatrix} + \mathbf{v} = \mathbf{H}\mathbf{x} + \mathbf{v} \tag{3}$$

where $\mathbf{y} \in \mathbb{R}^M$, and for $c = 1, \cdots, C$, $\mathbf{x}_c = [x_c[0], \cdots, x_c[M-K]]^{\mathrm{T}} \in \mathbb{R}^{N_e = M-K+1}$, $\mathbf{h}_c = [h_c[0], \cdots, h_c[K-1]]^{\mathrm{T}} \in \mathbb{R}^K$, $\mathbf{H}_c \in \mathbb{R}^{M \times N_e}$ is the matrix whose columns consist of delayed versions of the filter $\mathbf{h}_c$. The matrix $\mathbf{H} \in \mathbb{R}^{M \times N_e C}$ is the concatenation of convolutional operators that, in practice, we do not need to store explicitly, and $\mathbf{x} \in \mathbb{R}^{N_e C}$.

The model in Eq. 3 has promising applications in signal and image processing, where the signal of interest is the linear superposition of translated discrete-time filters or templates. Examples are DL [12] and image representation by learned features [24]. Similar to [24], we assume that the support of the filters are much smaller than the size of the signal ($K << M = \left\lfloor \frac{T_0}{f_s} \right\rfloor$ (Assumption I)). In addition, in DL [12] or sparse coding problems [25], $\mathbf{x}$ is assumed to be a sparse vector (Assumption II), $\sum_{c=1}^{C} N_c << N_e C$, resulting in $\mathbf{x}_c$ being an N_c-sparse vector for each $c = 1, \cdots, C$. Both of these are plausible assumptions in spike sorting [26], where $y[n]$ is a recording of extracellular voltage from the brain, the filters represent action potentials from an ensemble of neurons, and the firing rate of the neurons is assumed to be much smaller than the duration of the recording M.

In the case of recordings from an array of sensors, we assume that each of the sensors follows the model of Eq. 1. In practice, due to memory constraints, we divide $\mathbf{y}$ into J windows, each of size N. As we detail in Section VII, this also lets us take advantage of batch-based methods for training neural networks. Following Assumptions I & II, we let $K << N << M$. For each window (example) j, Eq. 3 becomes

$$\mathbf{y}^j = \mathbf{H}\mathbf{x}^j + \mathbf{v}^j \quad \text{for } j = 1, \cdots, J. \tag{4}$$

where now $\mathbf{H} \in \mathbb{R}^{N \times N_e C}$, $N_e = N - K + 1$, and $\mathbf{x}^j$ is N_c-sparse.

IV. Classical Dictionary Learning

Sparse coding problems assume that the data $\mathbf{y}^j \in \mathbb{R}^N$ can be represented as a linear combination of a few columns of the matrix $\mathbf{H}$, termed a dictionary. Given a set of signals $(\mathbf{y}^j)_{j=1}^J$ and the filters $\mathbf{H}$, sparse coding attempts to find the sparsest representation for each example $\mathbf{x}^j$ by solving the following problem [25]

$$(P_0): \quad \min_{\mathbf{x}^j} \|\mathbf{x}^j\|_0 \text{ s.t. } \mathbf{H}\mathbf{x}^j = \mathbf{y}^j \tag{5}$$

(P_0) is a non-convex optimization problem. Basis pursuit [2] is a convex relaxation of (P_0) that instead solves

$$(P_1): \quad \min_{\mathbf{x}^j} \|\mathbf{x}^j\|_1 \text{ s.t. } \mathbf{H}\mathbf{x}^j = \mathbf{y}^j \tag{6}$$

In the case of a signal $\mathbf{y}^j = \mathbf{H}\mathbf{x}^j + \mathbf{v}^j$ observed in the presence of bounded noise $\mathbf{v}^j$ such that $\|\mathbf{v}^j\|_2 \leq \epsilon$, (P_1) can be extended as follows [27]

$$(P_2): \quad \min_{\mathbf{x}^j} \|\mathbf{x}^j\|_1 \text{ s.t. } \|\mathbf{y}^j - \mathbf{H}\mathbf{x}^j\|_2 \leq \epsilon \tag{7}$$

Given the dictionary, (P_2) finds the sparse codes given noisy observations. In source separation problems such as spike sorting, the filters are unknown, and must therefore be learned along with the sparse codes. CDL assumes that the dictionary $\mathbf{H} \in \mathbb{R}^{N \times N_e C}$ has block-Toeplitz structure and is a linear operator that performs convolution with the filters $(\mathbf{h}_c)_{c=1}^C$, where C is the number of filters, and N_e is the length of each sparse code corresponding to a filter. The goal is to find these filters by solving an extension of (P_2) given by

$$\min_{(\mathbf{x}^j)_{j=1}^J, (\mathbf{h}_c)_{c=1}^C} \sum_{j=1}^{J} \|\mathbf{x}^j\|_1 \text{ s.t. } \|\mathbf{y}^j - \mathbf{H}\mathbf{x}^j\|_2 \leq \epsilon \quad \|\mathbf{h}_c\|_2 = 1 \quad \text{for } c = 1, \cdots, C, \tag{8}$$

where the constraint on the filters is to avoid scaling ambiguities. Solving for the sparse codes $(\mathbf{x}^j)_{j=1}^J$ and the filters $(\mathbf{h}_c)_{c=1}^C$ simultaneously in Eq. 8 is a non-convex optimization problem. The alternating-minimization method solves Eq. 8 in two stages: given an initial estimate of the filters, the algorithm alternates between a CSC step to estimate sparse codes $(\mathbf{x}^j)_{j=1}^J$ and a dictionary update step to estimate the filters $(\mathbf{h}_c)_{c=1}^C$ given the newly-estimated sparse codes [14]. In the case of dense (as opposed to convolutional) dictionary, [14] shows that the alternating minimization algorithm converges to the true dictionary whenever the dictionary satisfies RIP [28].

A. Convolutional Sparse Coding Update

Given the filters $\mathbf{H}$, the CSC step is separable over the J examples. We can solve for the j^{th} sparse code $\mathbf{x}^j$ using the unconstrained form of (P_2) given by

$$\min_{\mathbf{x}^j} \frac{1}{2}\|\mathbf{y}^j - \mathbf{H}\mathbf{x}^j\|_2^2 + \lambda\|\mathbf{x}^j\|_1, \tag{9}$$

where $\lambda > 0$ is a regularization parameter that depends on ϵ and encourages sparsity. State-of-the-art CSC algorithms use ADMM to solve Eq. 9 [29]. ADMM, however, is inherently an iterative algorithm that lacks the infrastructure that would enable the computation of the solution to Eq. 9 for all J examples in parallel. In section VI, we introduce an architecture that allows us to solve the CSC problem for all J examples in parallel using GPUs.

B. Convolutional Dictionary Update

Given the sparse codes, the filters are updated as follows

$$\min_{(\mathbf{h}_c)_{c=1}^C} \sum_{j=1}^{J} \frac{1}{2}\|\mathbf{y}^j - \mathbf{H}\mathbf{x}^j\|_2^2 \text{ s.t. } \|\mathbf{h}_c\|_2 = 1 \quad \text{for } c = 1, \cdots, C. \tag{10}$$

Unlike the CSC step, the dictionary update is *not* parallelizable over the J examples, which makes it computationally expensive. The work in [12] has proposed various methods to solve Eq. 6, of which we briefly describe two. The first solves Eq. 6 using ADMM by introducing a consensus variable [12], [15]. The second takes advantage of the symmetry of the convolution and solves a problem equivalent to Eq. 6 using gradient-based methods in the DFT-domain [12]. In both of these methods for updating the filters, the regularization parameter is treated as a hyper-parameter to be tuned through cross-validation. In the next section, we propose an approach based on Bayesian statistics and EM to estimate both the filters $\mathbf{H}$ and regularization parameter λ.

V. Expectation-Maximization-based Dictionary Learning and Parameter Estimation

The objective is to estimate the sparse codes, filters, and regularization parameter. We explain how this would be achieved using the EM algorithm in a Bayesian generative setting. The EM algorithm motivates the deep residual AE architecture proposed in section VI that is able to learn both the filters and regularization parameter in classical CDL.

A. Bayesian Generative Model

In the remainder of the treatment, we drop the superscript j from Eq. 4 to simplify the notation. This yields

$$\mathbf{y} = \mathbf{H}\mathbf{x} + \mathbf{v}, \tag{11}$$

where we assume $\mathbf{v} \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$. Hence, given the sparse codes, the filters, and σ^2, the data are distributed according to a multivariate Gaussian distribution. That is

$$\mathbf{y} \mid \mathbf{x}, \mathbf{H}, \sigma^2 \sim \mathcal{N}(\mathbf{H}\mathbf{x}, \sigma^2\mathbf{I}), \tag{12}$$

resulting in the data likelihood

$$P_{\mathbf{Y}}(\mathbf{y} \mid \mathbf{x}, \mathbf{H}, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{\frac{N}{2}}}\, e^{\frac{-1}{2\sigma^2}||\mathbf{y}-\mathbf{H}\mathbf{x}||_2^2} \tag{13}$$

To encourage sparsity, we assume that conditioned on λ, each sparse code $\mathbf{x}_c$, $c = 1, \cdots, C$, is drawn according to distribution whose density is the product of one-dimensional $i.i.d.$ Laplace probability density functions. That is

$$P(\mathbf{x}_c \mid \lambda) = \prod_{k=1}^{N_e} \frac{\lambda}{2} e^{-\lambda|x_c[k]|} = \left(\frac{\lambda}{2}\right)^{N_e} e^{-\lambda\|\mathbf{x}_c\|_1} \tag{14}$$

We assume the marginal prior on the filters $P(\mathbf{H})$ is non-informative, and that λ follows a Gamma prior

$$P(\lambda) = \frac{\delta^r}{\Gamma(r)}\lambda^{r-1}e^{-\delta\lambda} \tag{15}$$

where r, and δ are the shape and rate parameters of the density, respectively. Hence, $E[\lambda] = \frac{r}{\delta}$.

We are interested in the so-called complete-data likelihood, i.e. the joint distribution $P(\mathbf{y}, \mathbf{x}; \mathbf{H}, \lambda, \sigma^2)$ under the prior

$$P(\mathbf{x}, \lambda; \sigma^2) = \prod_{c=1}^{C} \left(\frac{\lambda}{2}\right)^{N_e} e^{-\lambda\|\mathbf{x}_c\|_1} P(\lambda). \tag{16}$$

The log complete-data likelihood is of the form

$$\begin{aligned} \log P(\mathbf{y}, \mathbf{x}; \mathbf{H}, \lambda, \sigma^2) &\propto \log P(\mathbf{y} \mid \mathbf{x}, \mathbf{H}, \lambda; \sigma^2) \\ &\quad + \log P(\mathbf{x}, \lambda; \sigma^2) \\ &\propto -\frac{1}{2\sigma^2}\|\mathbf{y} - \mathbf{H}\mathbf{x}\|_2^2 - \lambda\|\mathbf{x}\|_1 \\ &\quad + C\log P(\lambda) + N_e C \log\lambda, \end{aligned} \tag{17}$$

B. EM Algorithm

The EM algorithm is an algorithm to maximize the marginal log likelihood $\log P(\mathbf{y}; \mathbf{H}, \lambda, \sigma^2)]$ with respect to (w.r.t.) $\mathbf{H}$ and λ. Because the marginal likelihood is typically not available in closed form, EM operates instead on the complete-data likelihood $\log P(\mathbf{y}, \mathbf{x}; \mathbf{H}, \lambda, \sigma^2)$ by iteratively alternating between an E-step and M-step.

E-step: At iteration l, the E-step computes the expectation of complete-data likelihood w.r.t. the posterior of $\mathbf{x}$ given the data $\mathbf{y}$ and the parameters $\mathbf{H}^{(l-1)}$ and $\lambda^{(l-1)}$ from the previous iteration. Assuming the posterior is concentrated around the maximum a posteriori (MAP) estimate of $\mathbf{x}$, we can approximate this expectation by evaluating the complete-data log-likelihood of Eq. 17 around the posterior mode given by

$$\begin{aligned} \mathbf{x}^{(l)} &= \arg\max_{\mathbf{x}} \log P(\mathbf{x} \mid \mathbf{y}, \mathbf{H}^{(l-1)}, \lambda^{(l-1)}, \sigma^2) \\ &= \arg\min_{\mathbf{x}} \ \frac{1}{2\sigma^2}\|\mathbf{y} - \mathbf{H}^{(l-1)}\mathbf{x}\|_2^2 + \lambda^{(l-1)}\,||\mathbf{x}||_1 \end{aligned} \tag{18}$$

Stated otherwise, the mode of the posterior is the sufficient statistics of the E-step, assuming the posterior concentrates around it.

M-step: In M-step, the EM algorithm uses the sufficient statistic from the E-step to maximize the expectation of the log complete-data likelihood (Eq. 17) w.r.t. $\mathbf{H}$ and λ. Hence, given $\mathbf{x}^{(l)}$, we update $\mathbf{H}$ and λ at iteration l as follows

$$\mathbf{H}^{(l)} = \arg\min_{\mathbf{H}} \ \frac{1}{2}\|\mathbf{y} - \mathbf{H}\mathbf{x}^{(l)}\|_2^2 \tag{19}$$

$$\lambda^{(l)} = \arg\min_{\lambda} \ \lambda(\|\mathbf{x}^{(l)}\|_1 + C\delta) - (N_e + r - 1)C\log\lambda \tag{20}$$

Both objectives are convex w.r.t. to the parameter of interest. Moreover, in this setting, the updates for both parameters are available in closed-form. We also note that $\mathbf{H}$ is a convolutional filter and that, to avoid scaling ambiguities, a constraint on the norm of the filters can be enforced similar to Eq. 8.

The E-step parallels the CSC step in CDL. Eq. 19 parallels the dictionary update step in CDL, while Eq. 20 does not have an analogue in CDL. In principle, given J independent examples $\mathbf{y}^j$, both the CSC step and the E-step are parallelizable. However, both CDL and EM lack the infrastructure that would make it possible to solve the respective steps in parallel. The interpretation of CDL as EM motivates the AE architecture we introduce in the next section. This architecture, called constrained recurrent sparse auto-encoder (CRsAE), leverages the parallelism offered by GPUs to learn $\mathbf{H}$ by stochastic gradient descent at a fraction of the time required by state-of-the-art CDL algorithms. In addition, being motivated by EM, this architecture is also able to learn λ, unlike existing CDL algorithms.

VI. Constrained Recurrent Sparse Auto-encoders

When we first introduced CRsAE [20], it was as an interpretable deep residual AE architecture for CDL. Here, we leverage the connection between CDL and EM to further extend the architecture in such a way that both $\mathbf{H}$ and λ can be learned. In the new formulation of CRsAE, the forward pass of the architecture plays the role of E-step giving an estimate of the sparse code $\mathbf{x}$, and in the training stage, the backward pass updates the filters $\mathbf{H}$ and regularization parameter λ through a two-stage back-propagation.

A. Network Architecture for EM-based Dictionary Learning

CRsAE employs linear operators $\mathbf{H}$ and $\mathbf{H}^{\mathrm{T}}$ and a non-linear shrinkage operator η_ϵ. In our convolutional setting, $\mathbf{H}$: $\mathbb{R}^{N_e C} \to \mathbb{R}^N$ is a linear mapping from encoded sparse codes space to the data space performing a sum of convolutions as in Eq. 4. $\mathbf{H}^{\mathrm{T}}$: $\mathbb{R}^N \to \mathbb{R}^{N_e C}$ maps the input data into sparse code space while computing correlation between its input and filters $(\mathbf{h}_c)_{c=1}^C$. The shrinkage operator, $\eta_\epsilon : \mathbb{R}^{N_e C} \to \mathbb{R}^{N_e C}$, applies element-wise shrinkage to its input and is defined as

$$(\eta_\epsilon(\mathbf{z}))_n = (|z[n]| - \epsilon)_+ \mathrm{sgn}(z[n]) \tag{21}$$

E-step (encoder): The encoder imitates the E-step by mapping $\mathbf{y}$, through some encoding matrices and a non-linearity imposed by the FISTA algorithm [30], to a sparse code. To solve Eq. 18, one iteration of FISTA algorithm proceeds as follows

$$\begin{aligned} \mathbf{x}_t &= \eta_{\frac{\lambda\sigma^2}{L}}\left(\mathbf{w}_t + \frac{1}{L}\mathbf{H}^{\mathrm{T}}(\mathbf{y} - \mathbf{H}\mathbf{w}_t)\right) \\ s_t &= \frac{1 + \sqrt{1 + 4s_{t-1}^2}}{2} \\ \mathbf{w}_{t+1} &= \mathbf{x}_t + \frac{s_{t-1} - 1}{s_t}(\mathbf{x}_t - \mathbf{x}_{t-1}) \end{aligned} \tag{22}$$

where the constant $L \geq \sigma_{\max}(\mathbf{H}^{\mathrm{T}}\mathbf{H})$. Let $\mathbf{x}_t$ represent the sparse code after t iterations of FISTA. Assume $\mathbf{x}_{-1} = \mathbf{x}_0$, and let the "state" vector $\mathbf{z}_t = \begin{bmatrix} \mathbf{z}_t^{(1)} & \mathbf{z}_t^{(2)} \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} \mathbf{x}_t & \mathbf{x}_{t-1} \end{bmatrix}^{\mathrm{T}}$. The encoder follows a recurrence relation $\mathbf{z}_t = f(\mathbf{y}, \mathbf{z}_{t-1}, \mathbf{H})$ where $f(.)$ represents the operation in a single FISTA iteration as in Eq. 22. Indeed, the encoder performs deep unfolding [31] and is built by unfolding this recurrent relation T times, as depicted in the dashed box from Figure 2. The output of the encoder is the code at the last iteration, namely $\mathbf{x}_T = \mathbf{z}_T^{(1)}$. Similar to the network in [17], all the time steps of this recurrent neural network share the same input $\mathbf{y}$, unlike classical recurrent architectures in which the input is fed in a sequence. This recurrent behavior, summarized in Algorithm 1, allows us to produce approximately sparse codes, which are essential for DL [14]. It is important to note that, while not in the standard form of Eq. 22, the reader can verify that Algorithm 1 indeed implements FISTA.

Encoder and ResNet: The encoder in CRsAE resembles a deep residual network [22]. In fact, when unfolded, both the ISTA and FISTA algorithms can be interpreted as variants of deep residual networks. Residual networks owe their success to the principle of residual learning. The goal of residual learning is to learn an underlying mapping $\mathcal{H}(\mathbf{x})$ by approximating it using a cascade of residual functions $\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x}$, as depicted in Figure 1(a). As depicted in the figure, ISTA (Figure 1(b)) and FISTA (Figure 1(c)) each impose a specific structure on the form of the residual mapping $\mathcal{F}(\mathbf{x})$. The structure comes from the recurrence relationship that defines the specific algorithm (ISTA/FISTA).

Algorithm 1: $\mathrm{CRsAE}_{\mathrm{encoder}}(\mathbf{y}, \mathbf{h}, \lambda, \sigma, L)$: Encoder of CRsAE algorithm for producing sparse codes.

Input: $\mathbf{y}, \mathbf{h}, \lambda, \sigma, L \geq \sigma_{\max}(\mathbf{H}^{\mathrm{T}}\mathbf{H})$
Output: $\mathbf{z}_T^{(1)}$

$\mathbf{z}_0 = \mathbf{0}, s_0 = 0$
for $t = 1$ *to* T **do**
  $s_t = \frac{1+\sqrt{1+4s_{t-1}^2}}{2}$
  $\mathbf{w}_t = \mathbf{z}_{t-1}^{(1)} + \frac{s_{t-1}-1}{s_t}\left(\mathbf{z}_{t-1}^{(1)} - \mathbf{z}_{t-1}^{(2)}\right) = \left[\left(1 + \frac{s_{t-1}-1}{s_t}\right)\mathbf{I}_{N_e C}| - \frac{s_{t-1}-1}{s_t}\mathbf{I}_{N_e C}\right]\mathbf{z}_{t-1}$
  $\mathbf{c}_t = \mathbf{w}_t + \frac{1}{L}\mathbf{H}^{\mathrm{T}}(\mathbf{y} - \mathbf{H}\mathbf{w}_t)$
  $\mathbf{z}_t = \begin{bmatrix} \eta_{\frac{\lambda\sigma^2}{L}}(\mathbf{c}_t) & \mathbf{z}_{t-1}^{(1)} \end{bmatrix}^{\mathrm{T}}$

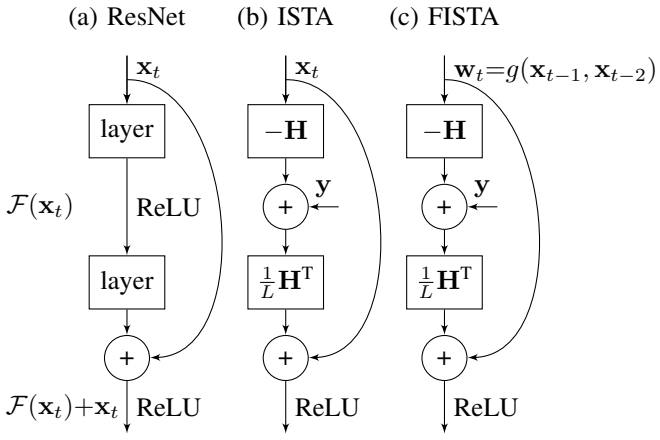

Fig. 1. ISTA and FISTA as ResNets. The building blocks of a ResNet, ISTA, and FISTA are shown in (a), (b), and (c), respectively. FISTA performs a residual learning similar to ISTA on a vector $\mathbf{w}_t$ which is a function of the sparse code $\mathbf{x}$ in unfolded layers t-1 and t-2 given by: $g(\mathbf{x}_{t-1}, \mathbf{x}_{t-2}) = \mathbf{x}_{t-1} + \frac{s_{t-1}-1}{s_t}(\mathbf{x}_{t-1} - \mathbf{x}_{t-2})$.

M-step: The M-step is a two-stage back-propagation procedure (Figure 3) that is the subject of the next sub-section. Here, we argue that the two-stage back-propagation procedure is the natural consequence of using Eqs. 19 and 20 to augment the encoder described previously.

M-step/H update: The least-squares objective from Eq. 19 motivates a linear decoder that applies $\mathbf{H}$ to the output of the encoder to reconstruct $\mathbf{y}$: the weights in the encoder and decoder are tied to each other. This is the CRsAE architecture previously introduced in [20]. Hence, the AE is constructed as

$$\mathrm{CRsAE}(\mathbf{y}, \mathbf{h}, \lambda, \sigma, L) = \mathbf{H}\ \mathrm{CRsAE}_{\mathrm{encoder}}(\mathbf{y}, \mathbf{h}, \lambda, \sigma, L). \tag{23}$$

Figure 2 is the block diagram of CRsAE. We refer to $\mathbf{z}_T^{(1)}$ as the sparse code (output of the encoder). Constraining the

encoder and decoder makes the filters interpretable in terms of the model in Eq. 2 and also the DL optimization problem in Eq. 8. Moreover, it reduces the number of trainable parameters by a factor of $\frac{1}{3}$ compared to the AE in [16]. Eq. 19 suggests the following loss function to learn $\mathbf{H}$:

$$\mathcal{L}_H(\mathbf{y},\mathbf{h},\lambda,\sigma,L) = \tfrac{1}{2}\|\mathbf{y} - \underbrace{\text{CRsAE}(\mathbf{y},\mathbf{h},\lambda,\sigma,L)}_{\hat{\mathbf{y}}}\|_2^2. \quad (24)$$

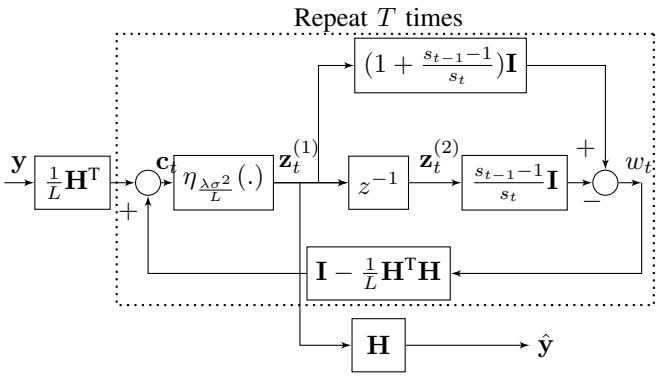

Fig. 2. Block Diagram of CRsAE. Given a convolutional filter, the encoder is a recurrent network performing T iterations of FISTA. The decoder applies the filters to the output of the encoder to reconstruct the input. The operator z^{-1} refers to the delay in discrete-time.

M-step/λ update: The objective function from Eq. 20 suggests that we can update λ by augmenting the encoder with the loss function from Eq. 20. Therefore, we use the following loss function to learn λ:

$$\begin{aligned}\mathcal{L}_\lambda(\mathbf{y},\mathbf{h},\lambda,\sigma,L) =& \lambda(\underbrace{\|\text{CRsAE}_{\text{encoder}}(\mathbf{y},\mathbf{h},\lambda,\sigma,L)\|_1}_{\|\mathbf{z}_T^{(1)}\|_1} + C\delta) \\ & -(N_e + r - 1)C\log\lambda\end{aligned} \quad (25)$$

B. Back-propagation

We train CRsAE by a two-stage back-propagation procedure. Figure 3 illustrates the steps involved in this procedure. Minimizing the losses in this two-stage back-propagation corresponds to maximizing the expectation of the log complete-data likelihood using the approximate sufficient statistics, i.e. the MAP estimate of the sparse codes, implicitly computed in the forward pass of CRsAE.

In the first stage, we perform back-propagation through the AE using the loss function from Eq. 24. Algorithm 2 is the back-propagation algorithm for computing the gradient of the loss function in Eq. 24 w.r.t. $\mathbf{H}$. The output of this algorithm is the gradient of the loss function in Eq. 24, denoted by $\delta\cdot$ [23], w.r.t. the set of filters $(\mathbf{h}_c)_{c=1}^C$ shared across all layers. The number of trainable parameters is only $K \times C$ and is equal to the number of filter parameters in one layer of the network.

In the second stage, we perform back-propagation through the encoder using the loss function from Eq. 25 motivated by Bayesian statistics. This latter stage is unique to our approach and does not have an equivalent in DL, where the regularization parameter is typically not trained. The back-propagation algorithm for computing the gradient of the loss function in Eq. 25 w.r.t. λ is given in Algorithm 3. The term $-(N_e+r-1)C\log\lambda$ plays an important role in the successful estimation of λ as it prevents its convergence to zero during learning. In other words, training λ naïvely through back-propagation by only minimizing the first component of Eq. 25 may result in convergence of λ to zero. If λ were to converge to zero, the encoder would fail to produce sparse codes. This implies that, in the absence of the second term from Eq. 25, if the AE were fed simulated data generated by known filters $\mathbf{H}$ and sparse codes, the overall two-stage procedure would fail to learn $\mathbf{H}$ because the success of DL relies on the encoder producing sparse codes [14].

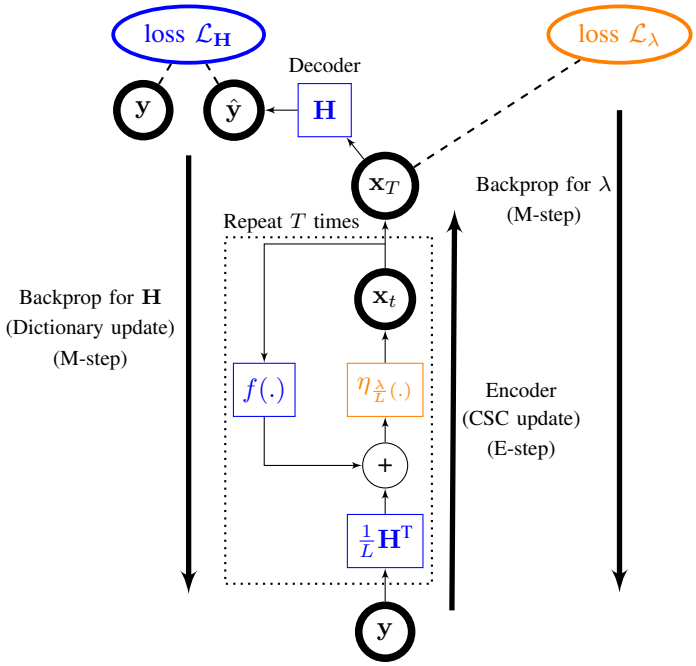

Fig. 3. CRsAE architecture and two-stage back-propagation training procedure. Given the input data $\mathbf{y}$, the encoder, which mimics the E-step of EM, outputs a sparse code $\mathbf{x}_T$ after T iterations. Given $\mathbf{x}_T$, in a stage that mimics the first stage of the M-step, CRsAE updates $\mathbf{H}$ by back-propagation through the reconstruction loss of the decoder. In the second stage, CRsAE back-propagates through the encoder to update λ using a loss function motivated by Bayesian statistics. In the figure, $f(\mathbf{x}_t) = (\mathbf{I} - \frac{1}{L}\mathbf{H}^{\mathrm{T}}\mathbf{H})(\mathbf{x}_t + \frac{s_t-1}{s_{t+1}}(\mathbf{x}_t - \mathbf{x}_{t-1}))$.

The derivations for both Algorithm 2 and 3 are provided in detail in the Appendix. Algorithm 4 summarizes the two-stage back-propagation procedure which, to our knowledge, is novel. Any gradient-based method can be used for training. For simplicity, Algorithm 4 uses gradient descent to learn the parameters. Table II summarizes the parallel between the steps in classical CDL, EM-based CDL, and CRsAE.

C. Training

We implemented CRsAE on Keras with TensorFlow as a backend and trained it on a GPU (NVIDIA Tesla V100) provided by AWS. The number of trainable parameters is $K \times C + 1$. We used the ADAM optimizer [32] to solve for the parameters by mini-batch gradient descent where the gradients are computed using the back-propagation algorithms described in the previous section. For faster convergence, we used the AMSGrad variant of ADAM [33]. The learned parameters are the ones minimizing the validation loss over all epochs.

Gamma Hyper-prior Parameters: Given a dense dictionary, a suggested value for the regularization parameter motivated by theory [2] is $\lambda = \frac{\sqrt{2\log(C\times N_e)}}{\sigma}$. We set the rate and shape

	Sparse Coding $\mathbf{x}$	Dictionary Learning $\mathbf{H}$	Regularizing λ
DL	$\mathbf{x}^{(l)} = \arg\min_{\mathbf{x}} \frac{1}{2}\|\mathbf{y} - \mathbf{H}^{(l-1)}\mathbf{x}\|_2^2 + \lambda\|\mathbf{x}\|_1$	$\mathbf{H}^{(l)} = \arg\min_{(\mathbf{h}_c)_{c=1}^C} \frac{1}{2}\|\mathbf{y} - \mathbf{H}\mathbf{x}^{(l)}\|_2^2$ s.t. $\|\mathbf{h}_c\|_2 = 1$	Hyper-parameter
EM	$\mathbf{x}^{(l)} = \arg\max_{\mathbf{x}} \log P(\mathbf{x} \mid \mathbf{y}, \mathbf{H}^{(l-1)}, \lambda^{(l-1)}, \sigma^2)$	$\mathbf{H}^{(l)} = \arg\min_{(\mathbf{h}_c)_{c=1}^C} \frac{1}{2}\|\mathbf{y} - \mathbf{H}\mathbf{x}^{(l)}\|_2^2$ s.t. $\|\mathbf{h}_c\|_2 = 1$	$\lambda^{(l)} = \arg\min_{\lambda} \lambda(\|\mathbf{x}^{(l)}\|_1 + C\delta) - (N_e + r - 1)C\log\lambda$
CRsAE	$\text{CRsAE}_{\text{encoder}}(\mathbf{y}, \mathbf{h}^{(l-1)}, \lambda^{(l-1)}, \sigma, L)$	$\mathbf{h}^{(l)} -= \alpha\nabla_{\mathbf{h}}\mathcal{L}_H(\mathbf{y}, \mathbf{h}^{(l-1)}, \lambda^{(l-1)}, \sigma, L)$	$\lambda^{(l)} -= \beta\nabla_{\lambda}\mathcal{L}_{\lambda}(\mathbf{y}, \mathbf{h}^{(l-1)}, \lambda^{(l-1)}, \sigma, L)$

TABLE II
COMPARISON OF CLASSICAL DL, EM-BASED DL, AND CRSAE.

Algorithm 2: $\nabla_{\mathbf{h}}\mathcal{L}_H(\mathbf{y}, \mathbf{h}, \lambda, \sigma, L)$: Back-propagation of CRsAE for filters.

Input: $\mathbf{y}, \lambda, \sigma, L, \mathbf{h}$, Variables $s_t, \mathbf{w}_t, \mathbf{c}_t, \mathbf{z}_T$ from forward propagation.
Output: $\delta\mathbf{h}$

$\delta\hat{\mathbf{y}} = \hat{\mathbf{y}} - \mathbf{y}, \delta\mathbf{h} = \mathbf{0}_K$
$\delta\mathbf{c}_{T+1} = \frac{\partial\hat{\mathbf{y}}}{\partial\mathbf{c}_{T+1}}\delta\hat{\mathbf{y}}$
$\delta\mathbf{h} = \delta\mathbf{h} + \frac{\partial\mathbf{c}_{T+1}}{\partial\mathbf{h}}\delta\mathbf{c}_{T+1}$
$\delta\mathbf{z}_T = \frac{\partial\mathbf{c}_{T+1}}{\partial\mathbf{z}_T}\delta\mathbf{c}_{T+1}$
for $t = T$ *to* 1 **do**
  $\delta\mathbf{c}_t = \frac{\partial\mathbf{z}_t}{\partial\mathbf{c}_t}\delta\mathbf{z}_t$
  $\delta\mathbf{h} = \delta\mathbf{h} + \frac{\partial\mathbf{c}_t}{\partial\mathbf{h}}\delta\mathbf{c}_t$
  $\delta\mathbf{z}_{t-1} = \frac{\partial\mathbf{c}_t}{\partial\mathbf{z}_{t-1}}\delta\mathbf{c}_t$

Algorithm 3: $\nabla_{\lambda}\mathcal{L}_{\lambda}(\mathbf{y}, \mathbf{h}, \lambda, \sigma, L)$: Back-propagation of CRsAE for regularization parameter.

Input: $\mathbf{y}, \lambda, \sigma, L, \mathbf{h}$, Variables $s_t, \mathbf{w}_t, \mathbf{c}_t, \mathbf{z}_T$ from forward propagation.
Output: $\delta\lambda$

$(\frac{\partial\|\mathbf{z}_T^{(1)}\|_1}{\partial\mathbf{z}_T})_i = \begin{cases} \text{sign}(\mathbf{z}_T[i]) \text{ if } \mathbf{z}_T[i] \neq 0 \\ 0 \quad \text{,otherwise} \end{cases}$
$\delta\lambda = \|\mathbf{z}_T^{(1)}\|_1 + C\delta - \frac{(N_e+r-1)C}{\lambda} + \lambda\frac{\partial\mathbf{z}_T}{\partial\lambda}\frac{\partial\|\mathbf{z}_T^{(1)}\|_1}{\partial\mathbf{z}_T}$
for $t = T$ *to* 1 **do**
  $\frac{\partial\mathbf{z}_t}{\partial\mathbf{z}_{t-1}} = \frac{\partial\mathbf{z}_t}{\partial\mathbf{c}_t}\frac{\partial\mathbf{c}_t}{\partial\mathbf{z}_{t-1}}$
  $\frac{\partial\|\mathbf{z}_T^{(1)}\|_1}{\partial\mathbf{z}_{t-1}} = \frac{\partial\mathbf{z}_t}{\partial\mathbf{z}_{t-1}}\frac{\partial\|\mathbf{z}_T^{(1)}\|_1}{\partial\mathbf{z}_t}$
  $\delta\lambda = \delta\lambda + \lambda\frac{\partial\mathbf{z}_{t-1}}{\partial\lambda}\frac{\partial\|\mathbf{z}_T^{(1)}\|_1}{\partial\mathbf{z}_{t-1}}$

Algorithm 4: Training CRsAE.

Input: $\mathbf{y}, \mathbf{h}^{\text{init}}, \lambda^{\text{init}}, \sigma, L \geq \sigma_{\max}(\mathbf{H}_{\text{init}}^{\mathrm{T}}\mathbf{H}_{\text{init}}), \alpha, \beta, I$
Output: $\mathbf{h}^{(I)}, \lambda^{(I)}$

$\mathbf{h}^{(0)} = \mathbf{h}^{\text{init}}$
$\lambda^{(0)} = \lambda^{\text{init}}$
for $l = 1$ *to* I **do**
  $\mathbf{h}^{(l)} -= \alpha\nabla_{\mathbf{h}}\mathcal{L}_H(\mathbf{y}, \mathbf{h}^{(l-1)}, \lambda^{(l-1)}, \sigma, L)$
  $\lambda^{(l)} -= \beta\nabla_{\lambda}\mathcal{L}_{\lambda}(\mathbf{y}, \mathbf{h}^{(l-1)}, \lambda^{(l-1)}, \sigma, L)$
  $\|\mathbf{h}_c^{(l)}\|_2 = 1$

of the gamma prior on λ such that the prior is centered around the suggested estimate. In CDL, we do not have access to the true filters, which results in an additional source of noise: $\mathbf{y} = \mathbf{H}_0\mathbf{x} + \underbrace{(\mathbf{H} - \mathbf{H}_0)\mathbf{x} + \mathbf{v}}_{\sigma_n}$ where σ_n contains observation noise and noise from the lack of knowledge of $\mathbf{H}$. Hence, it is reasonable to expect λ to deviate from the suggested estimate during training. To allow for this, we chose the rate parameter such that the distribution is wide enough for λ to deviate from its mean during training while maintaining stability by avoiding the convergence of λ to 0 or ∞. For the simulated data from Section VII-B, σ is known. For the real data from Section VII-C, we can estimate σ from "silent" periods in the signal.

Encoder Hyper-Parameters: The hyper-parameters of the encoder are L and number of FISTA iterations T. We choose L to be greater than $\sigma_{\max}(\mathbf{H}^{\mathrm{T}}\mathbf{H})$ [30]. For neural data, we can estimate L from an existing collection of action potentials. The value of T does not affect the trainable parameters or memory consumption as the encoder is a recurrent network with shared parameters. However, for the encoder to produce sparse codes, T should be set to a large number.

Optimizer Hyper-Parameters: The learning rate of the optimizer depends on the smoothness of the loss function which is the function of the model and dataset. Tuning the learning rate is important as training with a small learning rate not only requires a large number of epochs to converge but can also lead to getting trapped in local minima close to the initial value of the parameters. For the filters, we found an optimal learning rate range by varying it from 10^{-5} to 10^{-1} while monitoring the validation loss function. As suggested in [34], we picked the optimal learning rate as the one that leads to the sharpest drop in the validation loss. We tuned the learning rate for λ to a value in the range between 1 and 5. Choosing a larger learning rate for λ compared to $\mathbf{H}$ allows the value of λ, within each epoch, to stabilize faster than $\mathbf{H}$ within that epoch. In other words, within each epoch, λ converges quickly to a stable value, while the filters $\mathbf{H}$ keep improving. As discussed in Section VII, where we apply CRsAE to simulated and real data, we found that this behavior was crucial in allowing the two-stage back-propagation procedure to converge to the true filters $\mathbf{H}$ that underlie the data.

The batch size B of the optimizer corresponds to the number of examples used in every gradient update step of the back-propagation algorithm. For time-series dataset consisting of recordings of extracellular voltage from neurons, we found

that it was important to relate B to the expected number of spikes in each batch. We can estimate the expected number of spikes in a window given its length N and the firing rate of neurons. In turn, this calculation yields the expected number of spikes in each batch. In our experiments, we chose B so as to have enough spikes in each mini-batch but still have a large number of batches in each epoch to take advantage of stochasticity during training.

Augmentation: We used two forms of augmentation: flipping and circular rotation. Neither flipping nor circular rotation change the generative model (Eq. 2). However, these forms of data augmentation can be useful as the training is done through mini-batch gradient descent on noisy data [35]. In our model, augmentation by flipping, which changes the sign of the sparse codes, does not increase the complexity of the model, as the encoder uses two-sided shrinkage (Eq. 21), i.e. as opposed to one-sided ReLU [36]. In augmentation by circular rotation, the sparse code positions are shifted. This augmentation is done by circularly rotating each example by an integer delay randomly generated between 1 to N.

VII. Data Anlaysis

In this section, we train CRsAE to learn the filters that underlie two different datasets. The first is a dataset of simulated recordings of extracellular voltage from neurons with known ground-truth filters. We use this example to compare the ability of CRsAE to learn the ground-truth filters to the AE called learned convolutional sparse coding (LCSC) from [16]. The encoder of LCSC uses 3 ISTA iterations. Unlike in CRsAE, the decoder is *unconstrained*, i.e. it is not tied to the encoder. In LCSC, each filter is associated with its own regularization parameter. The overall objective of LCSC is to minimize the least-squares reconstruction loss from input to output. We also use the simulated example to compare the computational efficiency of CRsAE to the state-of-art optimization-based CDL algorithm from [12], which is implemented in the Sporco library [37] (we refer to this method by Sporco). Sporco solves the CSC update through ADMM [29], and solves the dictionary update by an accelerated proximal gradient method constraining unit norm on the filters. The second dataset consists of recordings of extracellular voltage from neurons in the rat Hippocampus [38]. We use this dataset to compare CRsAE to continuous basis pursuit (CBP) [39], the state-of-the-art optimization-based algorithm for spike sorting, which is the process of identifying the location of action potentials (sparse codes) and their corresponding neurons in extracellular recordings.

A. Evaluation Criteria

We use two criteria to evaluate the performance of CRsAE:

(a) Its ability to recover underlying convolutional filters.
(b) Its ability to perform spike sorting.

Let $\hat{\mathbf{h}}_c$ be one of the learned filters, we quantify (a) using the following standard measure [14]

$$\text{err}(\mathbf{h}_c, \hat{\mathbf{h}}_c) = 10 \log \left(\sqrt{1 - \frac{\langle \mathbf{h}_c, \hat{\mathbf{h}}_c \rangle^2}{\|\mathbf{h}_c\|_2^2 \|\hat{\mathbf{h}}_c\|_2^2}} \right) \qquad (26)$$

	Simulated	Real
Length of data T_0 [min]	17	4
Firing rate of each neuron [Hz]	30	-
Sampling frequency f_s [Hz]	10,000	10,000
Sparseness of code $\|\mathbf{x}^j\|_0$	3×4=12	-
σ	set based on SNR	0.058
Number of filters C	4	2
Length of each filter K	18	35
Length of each example N	1,000	60,000
Number of examples	10,100	24
Number of training examples	9,000	63 augmented
Number of validation examples	1,000	3
Number of testing examples	100	-
Number of trainable parameters	(18×4)+1=73	(35×2)+1=71
FISTA iterations T	180	600
Batch size B	1024	8
L	13.5	15
λ_{init}	$\frac{\sqrt{2\log(C\times N_e)}}{\sigma}$	$\frac{\sqrt{2\log(C\times N_e)}}{\sigma}$
δ	50	-
r	$\delta\lambda_{\text{init}}$	-
$\text{err}(\mathbf{h}_c, \mathbf{h}_c^{\text{init}})$	-3 to -4	-
Learning rate for $\mathbf{h}$	10^{-5} to 10^{-1}	10^{-5} to 10^{-1}
Learning rate for λ	1 to 5	-

TABLE III
Details of Datasets and Training Parameters

This measure ranges from 0 to $-\infty$. The closer the learned filter to the true one, the smaller $\text{err}(\mathbf{h}_c, \hat{\mathbf{h}}_c)$. In the context of (a), we characterize the sensitivity of CRsAE to noise by computing this measure for a range of SNR values.

We quantify (b) by comparing the estimated spike trains to the true spikes (sparse codes). In case of the real data, the true spikes are provided through intracellular voltage recordings. Similar to [39], for a given threshold, we compute the proportion of true (intracellular) spikes that are not identified correctly by CRsAE (true missed), as well as the proportion of spikes identified by CRsAE for which there is no true matching spike (false alarm). In practice, to identify a spike, given the neuron of interest c, we reconstruct the component of the data corresponding only to the neuron of interest. Then, we use a threshold and identify any non-zero spiking activity greater than the threshold to be a spike. A low threshold value would result in identifying background noise or other artifacts as spikes (false alarm), whereas a high threshold may result in missing spikes (true missed).

B. Simulated Data

Simulated extra-cellular data: We simulated a recording (Eq. 1) of length approximately $T_0 = 17$ minutes consisting of the sum electrical-voltage activity from $C = 4$ neurons, each with an average firing rate of 30 Hz. The unit of the recording was in mV. Consistent with the biophysics of neurons [26], we picked filters (action potentials) of length 1.8 ms. We chose the amplitudes $(x_{c,i})_{i=1}^{N_c}$ and times of occurrence $(\tau_{c,i})_{i=1}^{N_c}$ of the

filters independently for each neuron, respectively, according to a $\mathcal{N}(180, 30)$ distribution and a Poisson process with rate 30 Hz. The cross-correlation between the filters ranged from 0.5 to 0.95.

We divided the voltage recording into windows of length 100 ms, resulting in a total number $J = 10{,}100$ windows (examples). Assuming a sampling rate of $f_s = 10$ kHz, the length of each window was $N = 1{,}000$, and each filter was $K = 18$ samples long. Therefore, $\mathbf{y}^i \in \mathbb{R}^{1,000}$ and $\mathbf{h}_c \in \mathbb{R}^{18}$. For each example j, each of the vectors $(\mathbf{x}_c^j)_{c=1}^C$ was 3-sparse. Because of the refractory period of neurons, whose length is on the same order as the filter length, occurrences of filters from a given neuron cannot overlap within the signal. This implies that, in each sparse code $\mathbf{x}_c^j$, the non-zero entries were at least K samples apart. Filters from different neurons were allowed to overlap [26]. We added Gaussian noise $\mathbf{v}^j \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$ to each example $\mathbf{y}^j$, where the noise variance σ^2 was set according to a specified SNR. Finally, as is standard practice [40], we normalized the dataset by its maximum absolute value. Note that this scaling did not affect the generative model, as all the examples were re-scaled by the same constant. The first column of Table III summarizes the data and training parameters of the simulation.

SNR Analysis: We simulated data at four different equally-spaced SNRs in a log-scale ranging from 7 to 16 dB. We trained CRsAE and computed the measure from Eq. 26, which quantifies its ability to recover the filters used in the simulation, as a function of SNR. For training, we initialized the filters $(\mathbf{h}_c^{\text{init}})_{c=1}^C$ by adding random Gaussian noise to the true filters $(\mathbf{h}_c)_{c=1}^C$ such that $\text{err}(\mathbf{h}_c, \mathbf{h}_c^{\text{init}})$ was between -3 to -4 for all the filters. Figure 4 depicts $\text{err}(\mathbf{h}_c, \hat{\mathbf{h}}_c)$ as a function of SNR. The error was averaged over 20 independent experiments and is only shown for $c = 4$ as the curves for all the other filters were similar. The figure demonstrates that, compared to LCSC, CRsAE is able to recover the underlying filters and does so in a manner that is robust to noise. That is, as SNR increases, CRsAE learns filters that are closer to the true underlying ones. Interestingly, in spite of the fact that LCSC was able to reconstruct the noisy data very well, it failed to learn the underlying filters. Moreover, the filters that LCSC learned rarely correlated with the true filters as measured by Eq. 26. CRsAE, on the other hand, is able to perform denoising *and* learn the true filters. We attribute the success of CRsAE to two factors. First, it is well-known in the dictionary learning literature [14] that, given a noisy initial dictionary, the success of dictionary learning depends on the ability of the sparse coding step to produce sparse codes that are close to those that generated the examples. In CRsAE, we ran $T = 180$ FISTA iterations, compared to LCSC that only ran on the order of < 10 iterations of ISTA. That's why, the CRsAE encoder yielded sparse codes close to the true ones, while LCSC did not. Second, the fact that the encoder and decoder weights are tied in CRsAE means that it has fewer parameters to learn than LCSC with the same amount of data.

Dictionary Learning: Figure 6 shows a plot of the distance error from Eq. 26 as a function of epoch for one of the experiments from Figure 4 with 16 dB SNR. After 3 to 4 epochs, the distance between all learned and true filters decreases to less than -14. The learned dictionary corresponds to epoch 8 where the validation loss is minimized. The fast convergence of CRsAE is likely due to a) the low number of parameters to learn due to the sharing of parameters between the encoder and the decoder, and b) the large number of training examples. In the figure, the fact that the error increases slightly after it reaches a minimum is likely the effect of noise due to the stochastic nature of mini-batch gradient descent.

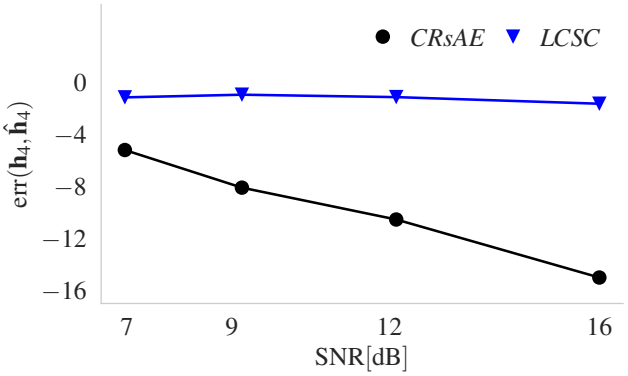

Fig. 4. Plot of $\text{err}(\mathbf{h}_c, \hat{\mathbf{h}}_c)$ as a function of SNR. Compared to LCSC, CRsAE is able to recover the true filters used in the simulations. Moreover, CRsAE performs better as SNR increases. Error $\text{err}(\mathbf{h}_c, \hat{\mathbf{h}}_c)$ is shown for only one of the filters as the curves for all the other filters were similar. The initial error is between -3 to-4. This result is the average of 20 independent experiments.

Learning the Regularization Parameter: The EM-based CRsAE architecture lets us automatically train and learn the regularization parameter using noisy data. Figure 5 shows the values of the regularization parameter for CRsAE as a function of epochs in a simulation with SNR of 16 dB. The figure shows that the regularization parameter for CRsAE converges after 4 to 5 epochs. We initialized λ as detailed in Section VI-C to a value of 168.04. The learned λ converged to 187.68, suggesting that the choice of prior on λ was flexible enough to allow it to deviate from its initial value. For LCSC, the regularization parameters for all filters converged to a very small value, resulting in over-fitting the noise in the dataset. To be fair, we found it difficult to optimize the very same loss function as in LCSC and therefore simply optimized the reconstruction loss. However, even if we had optimized the LCSC loss function, the regularization parameter could only converge to a very small value due to the absence of the regularization on the parameter as in the Bayesian loss function from Eq. 25, specifically the second term.

Figure 7 shows the filters corresponding to the error plots from Figure 6. In Figure 7, gray color (denoted by $\bigtriangledown$) indicates the initial filters. The true filters are shown in black. CRsAE is able to learn filters (red color denoted by $\star$) that are indistinguishable from the true filters, but LCSC (denoted by $\circ$ and depicted in green) fails to learn the underlying filters.

Together, Figures 4, 6 and 7 demonstrate the ability of CRsAE to perform convolutional dictionary learning, and its robustness to noise.

Speed Analysis: We compared the speed of CRsAE applied to CDL with Sporco. For fairness of comparison, we compared the speed of the algorithms given a fixed desired DL accuracy.

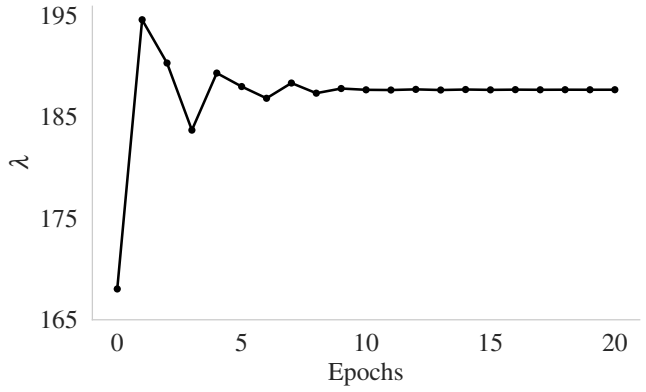

Fig. 5. The regularization parameter λ learned by CRsAE in each epoch of training for a trial for 16 dB SNR.

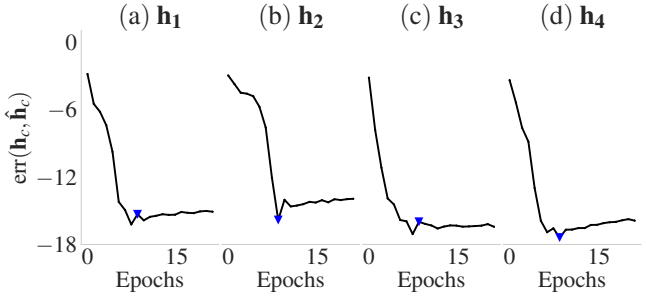

Fig. 6. Error err$(\mathbf{h}_c, \hat{\mathbf{h}}_c)$ as a function of epoch for the simulated dataset with 16 dB SNR. The error err$(\mathbf{h}_c, \hat{\mathbf{h}}_c)$ reaches a value below -14 after only 6 epochs, demonstrating that CRsAE converges fast. $\bigtriangledown$ denotes the epoch at which the validation loss is minimized.

We used simulated data at 16 dB SNR and set λ to the learned value from Section VII-B. We used the same regularization parameter ($\lambda\sigma^2 = 187.68\ (0.024)^2 \approx 0.11$ for SNR of 16 dB) for Sporco. Other hyper-parameters of Sporco (ρ, L) were tuned to $(4, 10^5)$ by random grid-search guided by hyper-parameters selection in [12]. We ran CRsAE and Sporco for 30 and 200 iterations, respectively. Here, 1 iteration refers to going over all of the dataset, i.e. an epoch in neural-networks terminology. Figure 8(a) plots the signal reconstruction error for CRsAE and Sporco as a function of time. The figure shows that, in this example, CRsAE performs CDL 5x faster than Sporco. Indeed, CRsAE took 69.27 s to train, and Sporco reached the same accuracy as CRsAE in 319.52 s. Figure 8(b)

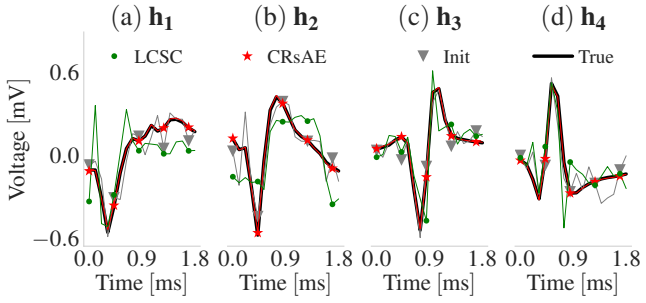

Fig. 7. Filters learned by CRsAE using simulated data with a 16 dB SNR. This highlights the ability of CRsAE to learn the true underlying filters (black). The gray curve ($\bigtriangledown$) corresponds to the initial filters. The red curve ($\star$) corresponds to the filter estimated by CRsAE, and the green curve ($\circ$) to the filters learned by LCSC.

		CRsAE	Sporco	CBP
Dictionary Learning	runtime	**69.27 s**	319.52 s	
	iterations	**10**	89	
Spike Sorting	runtime	**0.93 s**		17 hours

TABLE IV
DETAILS OF SPEED ANALYSIS

shows that CRsAE took 7 epochs to train, and the training loss reached its minimum in 10 epochs (denoted by $\bigtriangledown$). Sporco took 89 iterations to reach the same accuracy/loss as CRsAE. The first row of Table IV summarizes the speed comparison between CRsAE and Sporco. We chose to plot the signal reconstruction error, as opposed to Eq. 26, to demonstrate that CRsAE can also perform signal reconstruction. Both CRsAE and Sporco converged to the filters in Figure 7.

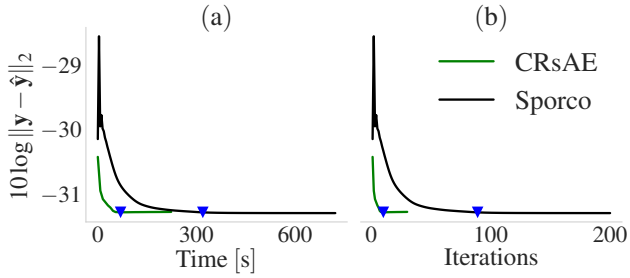

Fig. 8. Comparison of CRsAE to Sporco in terms of DL speed on simulated data with 16 dB SNR. In this example, CRsAE performs DL 5x faster than Sporco. $\bigtriangledown$ denotes the iteration/time at which the methods attain the minimum CRsAE validation loss.

C. Spike Sorting: Real Data

Extracellular data from rat Hippocampus: The dataset consists of extracellular tetrode recordings from the rat Hippocampus with simultaneous intracellular recording. The intracellular recording provides the ground-truth data that is used to assess the ability of CRsAE to successfully identify spikes. The reader can find additional details about these data in [38]. The goal is to sort the spikes that appear in the extracellular recordings. Spike sorting is an important source-separation problem in computational neuroscience [26]. It refers to the process of identifying the time of occurrence of action potentials (filters) in an extracellular voltage recording and their assignment to a neuron (source). Assuming the extracellular data follows the model of Eq. 2, we cast spike sorting as a CDL problem and apply CRsAE to it on the extracellular data.

We high-passed filter the raw data (Channel 0) at 400 Hz and whiten the noise. To minimize boundary effects, we picked a large window length of 6 s, resulting in a total of 24 windows (examples). We set the length of the filters to be learned to 3.5 ms. The sampling rate of the extracellular data was $f_s = 10$ kHz. Therefore, each window had length $N = 60{,}000$ samples, and each filter had length $K = 30$ samples. In the recording, there were mainly two neurons spiking [38], so we set $C = 2$. The data were divided by the maximum absolute value of the recording. Prior to normalization, we estimated the standard deviation of the noise from "silent" periods in the recording,

yielding a value of 1 in fractions of mV. We do not specify the units more precisely because the data are made available to us in an already-normalized form. Following normalization by the maximum absolute value of the signal, we obtained $\sigma \approx 0.058$. For training, we split the data into 21 examples for training and 3 examples for validation. To improve the learning performance, the training set was tripled by data augmentation (see section VI-C) resulting in 63 examples. The second column of Table III summarizes this dataset and the parameters used for training.

To initialize the filters for training, we used the following standard procedure [39]. We first spotted the location of potential spikes using a pre-computed threshold and windowed the data around the spotted positions. Then, we computed the two first principal components of the matrix whose rows comprise the obtained windows. Finally, we performed k-means clustering on the windows in principal-component space and picked the initial filters as the center of the clusters. Figure 9 shows the results of training CRsAE on the extracellular data. The initial filters are in gray color (denoted by $\triangledown$) and learned filters are in red color (denoted by $\star$). The figure shows that CRsAE is able to learn filters that resemble true action potentials.

In this experiment, we did not have access to the ground-truth filters from the extracellular recordings. However, the intracellular voltage recording was available for the neuron corresponding to the filter $\mathbf{h}_1$. We used the intracellular data to assess how well CRsAE can perform spike sorting. Unlike traditional methods for spike sorting that use principal component analysis to identify distinct templates within the data, CRsAE uses raw extracellular data. We compared CRsAE to CBP. Unlike CRsAE, CBP does not perform CDL: it simply performs CSC to identify the time of occurrence of filters *given* the filters. Figure 10 shows the trade-off between the *false alarm* proportion and the *true missed* proportion as a function of threshold. We refer the reader to Section VII-A for a refresher on how we obtain this figure. The figure demonstrates that CRsAE was competitive with CBP in terms of the trade-off it achieves between false alarms and true missed when applied to spike sorting. We also compared the speed of CRsAE to that of CBP for spike sorting of the simulated data lasting 17 minutes. CRsAE learned the filters in 69.27 s and performed spike sorting in only 0.93 s. The implementation of CBP at our disposal is not efficient enough to be applied to a 17-minute long recording. We estimated that CBP would take 17 hours to perform spike sorting given the filters. We estimated this timing by running CBP on 1, 2, 4, and 8 s of the data and observed a linear increase in the timing CBP needed for spike sorting [39]. The second row of Table IV summarizes the comparison of speeds of the two algorithms applied to spike sorting.

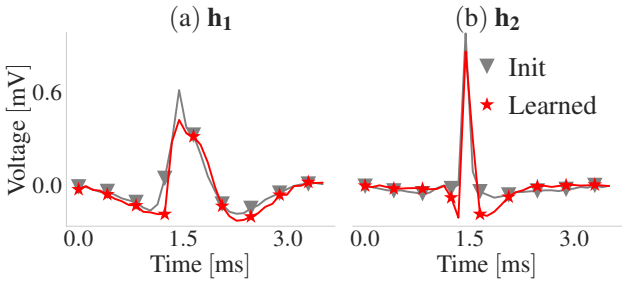

Fig. 9. Filters estimated by CRsAE applied to the extracellular tetrode recordings. The gray ($\triangledown$) curve corresponds to the initial filters. The filters learned by CRsAE are in red ($\star$). The figure shows that CRsAE is able to learn filters that resemble true action potentials.

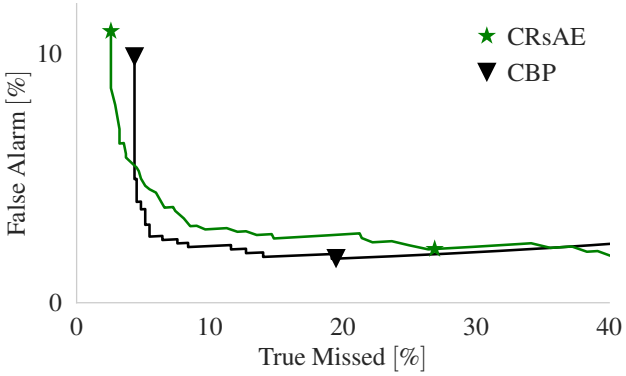

Fig. 10. Comparison between CRsAE and CBP applied to spike sorting. The trade-off curves, between *true missed* and *false alarm* proportions, obtained by using CRsAE (green $\star$) and CBP (black $\triangledown$) to sort the spikes from the neuron for which intracellular data are available. The figure demonstrates that CRsAE is competitive with CBP.

VIII. Conclusion

Training an auto-encoder for convolutional sparse coding to result in very low input/output prediction error is not a challenging task and has been done before [16]. A more challenging problem is to demonstrate that the weights learned by an auto-encoder are interpretable as convolutional filters in the context of convolutional dictionary learning or source separation, a problem that, to our knowledge, existing auto-encoder architectures for convolutional sparse coding have not solved. In this paper, we framed convolutional dictionary learning as training of an auto-encoder and proposed the constrained recurrent sparse auto-encoder (CRsAE). CRsAE is the first auto-encoder architecture to learn interpretable filters corresponding to a generative model. We highlighted this ability of CRsAE and its robustness to noise through simulation. The success of CRsAE in performing dictionary learning and source separation was owing to a) the deep recurrent and residual neural network architecture, inherited from FISTA, of its encoder, b) the tying of the weights to be learned in the encoder and decoder, and c) the Expectation Maximization-based training of the regularization parameter to ensure the sparseness of codes output by the encoder. The tying constraint on the encoder and decoder reduced the trainable parameters by a factor of $\frac{1}{3}$. In addition, the recurrent architecture of CRsAE with shared weights allowed us to produce sparse representations as the depth increases with a fix number of trainable parameters.

We highlighted the similarities and differences of methods such as classical dictionary learning, Expectation Maximization, and CRsAE for learning sparse representations from data. The architecture for CRsAE and the training procedure

were motivated by Expectation Maximization. The encoder in CRsAE produces sparse codes, a similar step to the sparse coding update in dictionary learning and the E-step of Expectation Maximization. CRsAE learns the filters and the regularization parameter using a two-stage back-propagation procedure, a step corresponding to the M-step in Expectation Maximization. In the first stage, the filters of interest are trained by back-propagation through the auto-encoder. This step parallels the dictionary update step in convolutional dictionary learning. In the second stage, the regularization parameter is updated by back-propagation through the encoder using a loss function motivated by Bayesian statistics. This two-stage back-propagation and the loss function for training the regularization parameter are the keys to ensuring the sparseness of the codes output by the encoder and preventing the convergence of the regularization parameter to zero.

We benchmarked CRsAE and showed that it performs dictionary learning 5x faster than the state-of-the-art algorithm [12] for convolutional dictionary learning based on alternating-minimization. We showed that the encoder of CRsAE can identify the location of action potentials in extracellular voltage recordings from the brain of rats. In particular, we showed that CRsAE can perform spike sorting as well as the state-of-the-art algorithm based on convex optimization known as continuous basis pursuit. At the same time, compared to continuous basis pursuit, CRsAE significantly speeds up the process of spike sorting by 900x.

APPENDIX

Here, we derive the back-propagation algorithm for CRsAE. This derivation is only for completeness. In practice, the gradients are computed through Tensorflow's autograd function while training. We assume, without loss of generality, number of examples $J = 1$. Let $\mathbf{h} = [\mathbf{h}_1^{\mathrm{T}}, \cdots, \mathbf{h}_C^{\mathrm{T}}]^{\mathrm{T}}$ where $\mathbf{h}_c = [h_c[0], h_c[1], \cdots, h_c[K-1]]^{\mathrm{T}}$, and $\mathbf{w}_t = [\mathbf{w}_{t,1}^{\mathrm{T}}, \cdots, \mathbf{w}_{t,C}^{\mathrm{T}}]^{\mathrm{T}}$ where $\mathbf{w}_{t,c} = [w_{t,c}[0], w_{t,c}[1], \cdots, w_{t,c}[N_e-1]]^{\mathrm{T}}$. Let $\mathbf{c}_{T+1} = \hat{\mathbf{y}} = \mathbf{H}\mathbf{c}_T$. The gradient of the loss function in Eq. 24 is

$$\frac{\partial \mathcal{L}_H}{\partial \mathbf{h}} = \delta \mathbf{h} = \sum_{t=1}^{T+1} \frac{\partial \mathbf{c}_t}{\partial \mathbf{h}} \frac{\partial \mathcal{L}_H}{\partial \mathbf{c}_t} = \sum_{t=1}^{T+1} \frac{\partial \mathbf{c}_t}{\partial \mathbf{h}} \delta \mathbf{c}_t, \tag{27}$$

where

$$\delta \mathbf{c}_t = \frac{\partial \mathbf{z}_t}{\partial \mathbf{c}_t} \frac{\partial \mathcal{L}_H}{\partial \mathbf{z}_t} = \frac{\partial \mathbf{z}_t}{\partial \mathbf{c}_t} \delta \mathbf{z}_t. \tag{28}$$

To evaluate Eq. 28, we first compute $\frac{\partial \mathbf{z}_t}{\partial \mathbf{c}_t}$ as

$$\frac{\partial \mathbf{z}_t}{\partial \mathbf{c}_t} = \left[\mathrm{diag}(\eta'_{\frac{\lambda\sigma^2}{L}}(\mathbf{c}_t)) | \mathbf{0}_{N_e C} \in \mathbb{R}^{N_e C \times 2N_e C} \right] \tag{29}$$

Then, we evaluate $\delta \mathbf{z}_{t-1}$ through the following recursion

$$\delta \mathbf{z}_{t-1} = \frac{\partial \mathbf{z}_t}{\partial \mathbf{z}_{t-1}} \frac{\partial \mathcal{L}_H}{\partial \mathbf{z}_t} = \frac{\partial \mathbf{c}_t}{\partial \mathbf{z}_{t-1}} \frac{\partial \mathbf{z}_t}{\partial \mathbf{c}_t} \frac{\partial \mathcal{L}_H}{\partial \mathbf{z}_t} = \frac{\partial \mathbf{c}_t}{\partial \mathbf{z}_{t-1}} \delta \mathbf{c}_t, \tag{30}$$

Having $\mathbf{c}_{T+1} = \hat{\mathbf{y}}$, we initialize the recursion with

$$\delta \mathbf{c}_{T+1} = \frac{\partial \mathcal{L}_H}{\partial \mathbf{c}_{T+1}} = \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{c}_{T+1}} \frac{\partial \mathcal{L}_H}{\partial \hat{\mathbf{y}}} = \mathbf{I}_N \delta \hat{\mathbf{y}} = \hat{\mathbf{y}} - \mathbf{y}. \tag{31}$$

We then evaluate $\frac{\partial \mathbf{c}_t}{\partial \mathbf{z}_{t-1}}$ needed for Eq. 38 as

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{z}_{t-1}} = \begin{cases} \begin{bmatrix} \mathbf{H}^{\mathrm{T}} \\ \mathbf{0}_{N_e C \times N} \end{bmatrix} \in \mathbb{R}^{2N_e C \times N}, \text{if } t = T+1 \\ \begin{bmatrix} \left(1 + \frac{s_{t-1}-1}{s_t}\right) \mathbf{I}_{N_e C} \\ -\frac{s_{t-1}-1}{s_t} \mathbf{I}_{N_e C} \end{bmatrix} \left[\mathbf{I}_{N_e C} - \frac{1}{L} \mathbf{H}^{\mathrm{T}} \mathbf{H}. \right], \text{otherwise.} \end{cases}$$

Finally, to finish the full gradient propagation procedure, we compute $\frac{\partial \mathbf{c}_t}{\partial \mathbf{h}}$ needed for Eq. 27 as

$$\frac{\partial \mathbf{c}_{T+1}}{\partial \mathbf{h}} = \mathbf{Z}_{T*}^{(1)} \in \mathbb{R}^{KC \times N}. \tag{32}$$

where

$$\begin{bmatrix} \mathbf{Z}_{T*,1}^{(1)} \\ \mathbf{Z}_{T*,2}^{(1)} \\ \vdots \\ \mathbf{Z}_{T*,C}^{(1)} \end{bmatrix} \tag{33}$$

$$\mathbf{Z}_{T*,c}^{(1)} = \begin{bmatrix} x_{T,c}^{(1)}[0] & x_{T,c}^{(1)}[1] & x_{T,c}^{(1)}[2] & \cdots & \cdots \\ 0 & x_{T,c}^{(1)}[0] & x_{T,c}^{(1)}[1] & \cdots & \cdots \\ \vdots & \vdots & x_{T,c}^{(1)}[0] & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \cdots \end{bmatrix}_{K \times N} \tag{34}$$

and for $t = 1, \cdots, T$,

$$\left(\frac{\partial \mathbf{c}_t}{\partial \mathbf{h}}\right)_{(a-1)K+m,(b-1)N_e+n} = \frac{1}{L} \sum_{\{|k|<K\}} \sum_{g=1}^{C} w_{t,g}[n-1+k] \frac{\partial}{\partial h_a[m-1]} c_{\mathbf{h}_b \mathbf{h}_g}[k] + y[m+n-2] \mathbb{I}_{\{a=b\}} \tag{35}$$

for $a, b = 1, \cdots, C$, $m = 1, \cdots, K$, and $n = 1, \cdots, N_e$. $c_{\mathbf{h}_b \mathbf{h}_g}[k]$ is the deterministic cross-correlation function between $\mathbf{h}_b$ and k-delayed samples of $\mathbf{h}_g$ as follows

$$c_{\mathbf{h}_b \mathbf{h}_g}[k] = \sum_{n=0}^{K-k-1} \mathbf{h}_b[n+k] \mathbf{h}_g[n] \tag{36}$$

Next, we derive the back-propagation algorithm for CRsAE for training λ. Without loss of generality, we assume $\sigma = 1$. The gradient of the loss function in Eq. 25 w.r.t. λ is

$$\begin{aligned} \frac{\partial \mathcal{L}_\lambda}{\partial \lambda} &= (\|\mathbf{z}_T^{(1)}\|_1 + C\delta + \lambda \frac{\partial \|\mathbf{z}_T^{(1)}\|_1}{\partial \lambda}) - \frac{(N_e + r - 1)C}{\lambda} \\ &= \|\mathbf{z}_T^{(1)}\|_1 + C\delta - \frac{(N_e + r - 1)C}{\lambda} + \lambda \sum_{t=1}^{T} \frac{\partial \mathbf{z}_t}{\partial \lambda} \frac{\partial \|\mathbf{z}_T^{(1)}\|_1}{\partial \mathbf{z}_t} \end{aligned} \tag{37}$$

We evaluate $\frac{\partial \|\mathbf{z}_T^{(1)}\|_1}{\partial \mathbf{z}_{t-1}}$ through the following recursion

$$\frac{\partial \|\mathbf{z}_T^{(1)}\|_1}{\partial \mathbf{z}_{t-1}} = \frac{\partial \mathbf{z}_t}{\partial \mathbf{z}_{t-1}} \frac{\partial \|\mathbf{z}_T^{(1)}\|_1}{\partial \mathbf{z}_t} \tag{38}$$

We initialize the recursion with

$$\left(\frac{\partial \|\mathbf{z}_T^{(1)}\|_1}{\partial \mathbf{z}_T}\right)_i = \begin{cases} \mathrm{sign}(\mathbf{z}_T[i]) \text{ if } \mathbf{z}_T[i] \neq 0 \\ 0 \quad \text{,otherwise} \end{cases} \tag{39}$$

where $\frac{\partial \|\mathbf{z}_T^{(1)}\|_1}{\partial \mathbf{z}_T} \in \mathbb{R}^{2N_e C \times 1}$ (Note that for when $\mathbf{z}_T[i] = 0$, we have used the sub-gradient as the gradient is not defined). Then we evaluate

$$\frac{\partial \mathbf{z}_t}{\partial \mathbf{z}_{t-1}} = \frac{\partial \mathbf{z}_t}{\partial \mathbf{c}_t} \frac{\partial \mathbf{c}_t}{\partial \mathbf{z}_{t-1}} \tag{40}$$

where $\frac{\partial \mathbf{z}_t}{\partial \mathbf{c}_t}$ and $\frac{\partial \mathbf{c}_t}{\partial \mathbf{z}_{t-1}}$ are derived previously in the derivation of back-propagation for the filters.

Finally, to finish the full gradient propagation procedure, we compute $\frac{\partial \mathbf{z}_t}{\partial \lambda} \in \mathbb{R}^{1 \times 2N_e C}$ needed for Eq. 37 from $\mathbf{z}_t^{(1)} = \eta_{\frac{\lambda}{L}}(\mathbf{c}_t)$

$$\left(\frac{\partial \mathbf{z}_t}{\partial \lambda}\right)_i = \begin{cases} -\frac{1}{L}\ \text{sign}(c_t[i])\ \text{if}\ |c_t[i]| \geq \frac{\lambda}{L} \\ 0,\ \text{otherwise.} \end{cases} \tag{41}$$

ACKNOWLEDGMENTS

The authors gratefully acknowledge support from the Quantitative Biology Initiative at Harvard University. The authors would also like to acknowledge support from ARO grant number W911NF-16-1-0368. The authors would also like to thank AWS for their generous support.

REFERENCES

[1] S. Chen, S. A. Billings, and W. Luo, "Orthogonal least squares methods and their application to non-linear system identification," *International Journal of Control*, vol. 50, no. 5, pp. 1873–1896, 1989.

[2] S. S. Chen, D. L. Donoho, and M. A. Saunders, "Atomic decomposition by basis pursuit," *SIAM Review*, vol. 43, pp. 129–159, 1998.

[3] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[4] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 39, no. 1, pp. 1–22, 1977.

[5] X. F. Zhu, B. Li, and J. D. Wang, "L1-norm sparse learning and its application," *Applied Mechanics and Materials*, vol. 88-89, pp. 379–385, 2011.

[6] T. Park and G. Casella, "The bayesian lasso," *Journal of the American Statistical Association*, vol. 103, no. 482, pp. 681–686, 2008.

[7] S. Mallat, "A theory for multiresolution signal decomposition: The wavelet representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, pp. 674–693, 1989.

[8] K. Engan, S. Aase, and J. Hakon Husoy, "Method of optimal directions for frame design," in *Proc. 1999 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (ICASSP)*, vol. 5, 1999, pp. 2443–2446.

[9] M. Aharon, M. Elad, and A. Bruckstein, "K-svd: An algorithm for designing overcomplete dictionaries for sparse representation," *IEEE Transactions on Signal Processing*, vol. 54, no. 11, pp. 4311–4322, 2006.

[10] H. Bristow, A. Eriksson, and S. Lucey, "Fast convolutional sparse coding," in *Proc. 2013 IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 391–398.

[11] M. S. Lewicki and T. J. Sejnowski, "Coding time-varying signals using sparse, shift-invariant representations," in *Proc. Advances in Neural Information Processing Systems 11 (NIPS)*, 1999, pp. 730–736.

[12] C. Garcia-Cardona and B. Wohlberg, "Convolutional dictionary learning: A comparative review and new algorithms," *IEEE Transactions on Computational Imaging*, vol. 4, no. 3, pp. 366–381, Sep. 2018, there are errors in Equations (18) and (19) in the published version of the paper. These have been corrected in the most recent arXiv version.

[13] V. Papyan, Y. Romano, and M. Elad, "Convolutional neural networks analyzed via convolutional sparse coding," *Journal of Machine Learning Research*, vol. 18, pp. 1–52, 2017.

[14] A. Agarwal, A. Anandkumar, P. Jain, P. Netrapalli, and R. Tandon, "Learning sparsely used overcomplete dictionaries via alternating minimization," *SIAM Journal on Optimization*, vol. 26, pp. 2775–2799, 2016.

[15] S. P. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends in Machine Learning*, vol. 3, pp. 1–122, 2011.

[16] H. Sreter and R. Giryes, "Learned convolutional sparse coding," in *Proc. 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018, pp. 2191–2195.

[17] J. T. Rolfe and Y. LeCun, "Discriminative recurrent sparse auto-encoders," in *Proc. International Conference on Learning Representations (ICLR)*, 2013, pp. 1–15.

[18] A. Makhzani and B. J. Frey, "k-sparse autoencoders," in *Proc. International Conference on Learning Representations (ICLR)*, 2014, pp. 1–9.

[19] S. Venkataramani, Y. C. Sübakan, and P. Smaragdis, "Neural network alternatives toconvolutive audio models for source separation," in *Proc. 2017 IEEE 27th International Workshop on Machine Learning for Signal Processing (MLSP)*, 2017, pp. 1–6.

[20] B. Tolooshams, S. Dey, and D. Ba, "Scalable convolutional dictionary learning with constrained recurrent sparse auto-encoders," in *Proc. 2018 IEEE 28th International Workshop on Machine Learning for Signal Processing (MLSP)*, Sept. 2018, pp. 1–6.

[21] T. Blumensath and M. E. Davies, "Iterative hard thresholding for compressed sensing," *Applied and Computational Harmonic Analysis*, vol. 27, no. 3, pp. 265 – 274, 2009.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.

[23] K. Gregor and Y. LeCun, "Learning fast approximations of sparse coding," in *Proc. the 27th International Conference on Machine Learning (ICML)*, 2010, pp. 399–406.

[24] M. D. Zeiler, D. Krishnan, G. W. Taylor, and R. Fergus, "Deconvolutional networks," in *Proc. 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2010, pp. 2528–2535.

[25] D. L. Donoho and M. Elad, "Optimally sparse representation in general (nonorthogonal) dictionaries via l1 minimization," *Proc. the National Academy of Sciences*, vol. 100, no. 5, pp. 2197–2202, 2003.

[26] M. S. Lewicki, "A review of methods for spike sorting: the detection and classification of neural action potentials," *Network: Computation in Neural Systems*, vol. 9, no. 4, pp. R53–R78, 1998.

[27] E. J. Candès, J. K. Romberg, and T. Tao, "Stable signal recovery from incomplete and inaccurate measurements," *Communications on Pure and Applied Mathematics*, vol. 59, pp. 1–15, 2006.

[28] E. J. Candès, "The restricted isometry property and its implications for compressed sensing," *Comptes Rendus Mathematique*, vol. 346, no. 9, pp. 589 – 592, 2008.

[29] B. Wohlberg, "Efficient convolutional sparse coding," in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Florence, Italy, May 2014, pp. 7173–7177.

[30] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM journal on imaging sciences*, vol. 2, no. 1, pp. 183–202, 2009.

[31] J. R. Hershey, J. L. Roux, and F. Weninger, "Deep unfolding: Model-based inspiration of novel deep architectures," *arXiv:1409.2574 [cs.LG]*, pp. 1–27, 2014.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. the 3rd International Conference on Learning Representations (ICLR)*, 2014, pp. 1–15.

[33] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," in *Proc. International Conference on Learning Representations*, 2018, pp. 1–23.

[34] L. N. Smith, "Cyclical learning rates for training neural networks," in *Proc. 2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, vol. 1, 2017, pp. 464–472.

[35] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Advances in Neural Information Processing Systems 25 (NIPS)*, 2017, pp. 1097–1105.

[36] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. 14th International Conference on Artificial Intelligence and Statistics*, 2011, pp. 315–323.

[37] B. Wohlberg, "SPORCO: A Python package for standard and convolutional sparse representations," in *Proc. 15th Python in Science Conference*, Austin, TX, USA, July 2017, pp. 1–8.

[38] K. Harris, D. Henze, J. Csicsvari, H. Hirase, and G. Buzsaki, "Accuracy of tetrode spike separation as determined by simultaneous intracellular and extracellular measurements," *Journal Of Neurophysiology*, vol. 84, no. 1, pp. 401–414, 2000.

[39] C. Ekanadham, D. Tranchina, and E. P. Simoncelli, "A unified framework and method for automatic neural spike identification," *Journal of neuroscience methods*, vol. 222, pp. 47–55, 2014.

[40] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," *Neural Networks: Tricks Of The Trade*, vol. 1524, pp. 9–50, 1998.