Learning Fast Approximations of Sparse Coding

Karol Gregor and Yann LeCun {KGREGOR,YANN}@CS.NYU.EDU
Courant Institute, New York University, 715 Broadway, New York, NY 10003, USA

Abstract

In Sparse Coding (SC), input vectors are reconstructed using a sparse linear combination of basis vectors. SC has become a popular method for extracting features from data. For a given input, SC minimizes a quadratic reconstruction error with an L_1 penalty term on the code. The process is often too slow for applications such as real-time pattern recognition. We proposed two versions of a very fast algorithm that produces approximate estimates of the sparse code that can be used to compute good visual features, or to initialize exact iterative algorithms. The main idea is to train a non-linear, feed-forward predictor with a specific architecture and a fixed depth to produce the best possible approximation of the sparse code. A version of the method, which can be seen as a trainable version of Li and Osher's coordinate descent method, is shown to produce approximate solutions with 10 times less computation than Li and Osher's for the same approximation error. Unlike previous proposals for sparse code predictors, the system allows a kind of approximate "explaining away" to take place during inference. The resulting predictor is differentiable and can be included into globally-trained recognition systems.

1. Introduction

Sparse coding is the problem of reconstructing input vectors using a linear combination of an overcomplete family basis vectors with sparse coefficients (Olshausen & Field, 1996; Chen et al., 2001; Donoho & Elad, 2003). This paper introduces a very efficient method for computing good approximations of optimal sparse codes.

Sparse coding has become extremely popular for extracting features from raw data, particularly when the dictionary of basis vectors is learned from unlabeled data. Several such unsupervised learning methods have been proposed to learn the dictionary. There have been applications of sparse coding in many fields including visual neuroscience (Olshausen & Field, 1996; Hoyer, 2004; Lee et al., 2007) and image restoration (Elad & Aharon, 2006; Ranzato et al., 2007b; Mairal et al., 2008). Recently, these methods have been the focus of a considerable amount of research for extracting visual features for object recognition (Ranzato et al., 2007a; Kavukcuoglu et al., 2008; Lee et al., 2009; Yang et al., 2009; Jarrett et al., 2009; Yu et al., 2009). A major problem with sparse coding for applications such as object recognition is that the inference algorithm is somewhat expensive, prohibiting real-time applications. Given an input image the inference algorithm must compute a sparse vector for each and every patch in the image (or for all local collections of low-level features, if sparse coding is used as a second stage of transformation (Yang et al., 2009)). Consequently, a large amount of research has been devoted to seeking efficient optimization algorithms for sparse coding (Daubechies et al., 2004; Lee et al., 2006; Wu & Lange, 2008; Li & Osher, 2009; Mairal et al., 2009; Beck & Teboulle, 2009; Hale et al., 2008; Vonesch & Unser, 2007).

The main contribution of this paper is a highly efficient learning-based method that computes good approximations of optimal sparse codes in a fixed amount of time. Assuming that the basis vectors of a sparse coder have been trained and are being kept fixed, the main idea of the method is to *train a parameterized non-linear "encoder" function to predict the optimal sparse code, by presenting it with examples of input vectors paired with their corresponding optimal sparse codes* obtained through conventional optimization methods. After training, the encoder function has a pre-determined complexity (though it is adjustable before training), and can be used to predict approximate sparse codes with a fixed computational cost and a prescribed expected error.

The basic idea of using encoders for sparse code prediction has been proposed by others. Particularly relevant to our approach is the "predictive sparse decomposition" method (Kavukcuoglu et al., 2008; Jarrett et al., 2009), but their predictor is very simplistic and produces crude approximations to the sparse codes. Our contribution is to propose a particular form and particular parameterization of the

Appearing in *Proceedings of the 27^{th} International Conference on Machine Learning*, Haifa, Israel, 2010.

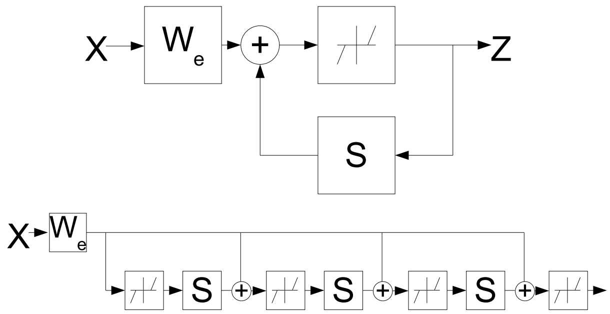

Figure 1. **Top:** block diagram of the ISTA algorithm for sparse coding. The optimal sparse code is the fixed point of $Z(k+1) = h_\alpha(W_e X - SZ(k))$ where X is the input, h_α is a coordinate-wise shrinking function with threshold α, W_e is the transpose of the dictionary matrix W_d (whose columns are the basis vectors), and S is $W_d^T W_d$. **Bottom:** The proposed approximator "Learned ISTA", uses a time-unfolded version of the ISTA block diagram, truncated to a fixed number of iterations (3 here). The matrices W_e and S, are learned, so as to minimize the approximation error to the optimal sparse code on a given dataset. The method allows us to impose restrictions on S so as to further reduce the computational burden (e.g. keeping many terms at 0, or using a low-rank factorized form). Another similar trainable encoder architecture based on the Coordinate Descent algorithm is also proposed.

encoder which, unlike previous proposals, implements an approximate "explaining away" (or competition) between code components, so that if two sets of basis vectors could reconstruct the input equally well, one set will be picked by the algorithm, while the other one will be suppressed. Our first encoder architecture essentially implements a truncated form of the Iterative Shrinkage and Thresholding Algorithm (ISTA) (Daubechies et al., 2004; Rozell et al., 2008; Beck & Teboulle, 2009). While ISTA uses two matrices that are computed from the basis vectors, our method, dubbed LISTA (Learned ISTA) learns those two matrices so as to produce the lowest possible error in a given number of iterations. Our second encoder architecture is based on a truncated form of the Coordinate Descent algorithm (CoD) (Li & Osher, 2009), again with learned matrices instead of pre-computed ones.

1.1. Sparse Coding

In the most popular form of sparse coding, the inference problem is, for a given input vector $X \in \mathbb{R}^n$, to find the optimal sparse code vector $Z^* \in \mathbb{R}^m$ that minimizes an energy function that combines the square reconstruction error and an L_1 sparsity penalty on the code:

$$E_{W_d}(X, Z) = \frac{1}{2}||X - W_d Z||_2^2 + \alpha||Z||_1 \qquad (1)$$

Algorithm 1 ISTA

function ISTA(X, Z, W_d, α, L)
  Require: $L >$ largest eigenvalue of $W_d^T W_d$.
  Initialize: $Z = 0$,
  repeat
    $Z = h_{(\alpha/L)}(Z - \frac{1}{L} W_d^T(W_d Z - X))$
  until change in Z below a threshold
end function

where W_d is an $n \times m$ dictionary matrix whose columns are the (normalized) basis vectors, α is a coefficient controlling the sparsity penalty. The overcomplete case corresponds to $m > n$. The optimal code for a given X is defined as $Z^* = \text{argmin}_Z E(X, Z)$.

The dictionary matrix is often learned by minimizing the average of $\min_z E_{W_d}(X, Z)$ over a set of training samples using a stochastic gradient method (Olshausen & Field, 1996). Training such a system on natural image patches produces Gabor-like filters covering the space of locations, frequencies, and orientations. Experiments reported in this paper are conducted on datasets of natural image patches and handwritten digits.

2. Iterative Shrinkage Algorithms

This section describes baseline iterative shrinkage algorithms for finding sparse codes. The ISTA and FISTA methods (Beck & Teboulle, 2009) update the whole code vector in parallel, while the more efficient Coordinate Descent method (CoD) (Li & Osher, 2009) updates the components one at a time and carefully selects which component to update at each step. Both methods refine the initial guess through a form of mutual inhibition between code component, and component-wise shrinkage.

2.1. ISTA and Fast ISTA

A popular algorithm for sparse code inference is the Iterative Shrinkage and Thresholding Algorithm (see for example (Daubechies et al., 2004; Beck & Teboulle, 2009), and (Rozell et al., 2008) for a continuous-time, biologically relevant form of ISTA). The method is given in Algorithm 1, and the block diagram of the method is represented in figure 1(a). Given an input vector X, ISTA iterates the following recursive equation to convergence:

$$Z(k+1) = h_\theta\left(W_e X + SZ(k)\right) \quad Z(0) = 0 \qquad (2)$$

The elements of the equation are described below. First, we define a constant L which must be an upper bound on the largest eigenvalue of $W_d^T W_d$. The "backtracking" form of ISTA (not described here) automatically adjusts this constant as part of the algo-

Algorithm 2 Coordinate Descent (Li & Osher, 2009)

function CoD(X, Z, W_d, S, α)
  Require: $S = I - W_d^T W_d$
  Initialize: $Z = 0$; $B = W_d^T X$
  repeat
    $\bar{Z} = h_\alpha(B)$
    $k =$ index of largest component of $|Z - \bar{Z}|$
    $\forall j \in [1, m]: \quad B_j = B_j + S_{jk}(\bar{Z}_k - Z_k)$
    $Z_k = \bar{Z}_k$
  until change in Z is below a threshold
  $Z = h_\alpha(B)$
end function

rithm. The other elements in eq. 2 are:

$$\text{filter matrix: } W_e = \frac{1}{L} W_d^T$$
$$\text{mutual inhibition matrix: } S = I - \frac{1}{L} W_d^T W_d$$
$$\text{shrinkage function: } [h_\theta(V)]_i = \text{sign}(V_i)(|V_i| - \theta_i)_+ \tag{3}$$

The function $h_\theta(V)$ is a component-wise vector shrinkage function with a vector of thresholds θ. In standard ISTA, all the thresholds are set to $\theta_i = \alpha/L$.

Depending on the overcompleteness and hardware implementation, one can use either factorized form shown in Algorithm 1 with computational complexity $O(mn)$, pre-computed matrix S with complexity $O(m^2)$ or pre-computed matrix S where only nonzero coefficients of the code vector are propagated, with complexity $O(mk)$ where k is the sparsity averaged over iterations and samples.

FISTA (Beck & Teboulle, 2009) is a version of this algorithm that converges more rapidly, both in theory and in practice. The main difference is the introduction of a "momentum" term in the dynamics. The new code vector is equal to the shrinkage function applied to the previous code vector, plus a coefficient times the difference of the last two outputs of the shrinkage function (akin to a momentum effect). See (Beck & Teboulle, 2009) for details. While it is fast by most standards, FISTA may require several dozen iterations to produce accurate sparse codes.

2.2. Coordinate Descent (CoD)

In (F)ISTA all the code components are updated simultaneously, which requires $O(mn)$, $O(m^2)$ or $O(mk)$ operations per iteration, as discussed. The idea of the Coordinate Descent algorithm (CoD) is to change only one carefully chosen coordinate at a time, a step which takes $O(m)$ operations. Repeating this $O(m)$ or $O(n)$ times produces a better approximation than updating all the coordinates at the same time for the same amount of computation. The algorithm is as follows. At any given step, pick a code component, and minimize the energy with respect to that component, keeping all other components constant. Repeat until convergence. The code component being chosen at any given point is the one that will be subject to the largest modification through this update. The steps are detailed in Algorithm 2. Since only one component is modified, only one column of S is used to propagate the changes for the next shrinkage operation, and the cost is $O(m)$. The max operation required to pick the next best component is also $O(m)$. With infinitely many iterations it converges to the optimal sparse code, but consistently produces better approximations with considerably fewer operations than FISTA.

This algorithm can also be interpreted in the framework of the Figure 1. Similarly to ISTA, we set $W_e = W_d^T$ and $\theta_i = \alpha$. The feedback mechanism is slightly more complicated than in ISTA, but can be expressed as a linear operation with a very sparse, iteration-dependent matrix, which takes only $O(m)$ operations.

The common wisdom is that CoD is the fastest algorithm in existence for sparse code inference (or at least, it is in the leading pack).

3. Trainable Sparse Code Predictors

We now come to the main focus of the paper, which is to propose fast encoders that can be trained to compute approximate sparse codes. An important desired characteristic of encoders is that they must be continuous and almost-everywhere differentiable functions with respect to their parameters and with respect to their input. Differentiability with respect to the parameters will ensure that we can use gradient-based learning methods to train them, while differentiability with respect to the input will ensure that gradients can be back-propagated through them, enabling their use as components of larger globally-trainable systems (Jarrett et al., 2009).

The basic idea is to design a non-linear, parameterized, feed-forward architecture with a fixed depth that can be trained to approximate the optimal sparse code. The architecture of our encoders will de denoted $Z = f_e(X, W)$, where W collectively designates all the trainable parameters in the encoder. Training the encoder will be performed using stochastic gradient descent to minimize a loss function $\mathcal{L}(W)$, defined as the squared error between the predicted code and the optimal code averaged over a training set $(X^1, \ldots, X^p)$:

$$\mathcal{L}(W) = \frac{1}{P} \sum_{p=0}^{(P-1)} L(W, X^p) \;\text{ with} \tag{4}$$

$$L(W, X^p) = \frac{1}{2} ||Z^{*p} - f_e(W, X^p)||^2 \tag{5}$$

where $Z^{*p} = \text{argmin}_z E_{W_d}(X^p, Z)$ is the optimal code for sample X^p, as obtained with the CoD method. The learning algorithm is a simple stochastic gradient descent:

$$W(j+1) = W(j) - \eta(j)\frac{dL(W, X^{(j \bmod P)})}{dW} \quad (6)$$

where $\eta(j)$ decays as $1/j$ to ensure convergence.

3.1. Baseline Encoder Architecture

As a baseline of comparison, we now discuss a simple class of encoder architectures of the form

$$Z = g(W_e X) \quad (7)$$

where W_e is an $m \times n$ trainable matrix, g is a coordinate-wise non-linearity, as proposed by (Kavukcuoglu et al., 2008). They proposed to use a non-linearity of the form $g = D \tanh$ where D is a trainable diagonal matrix (gain). This architecture has two major shortcomings. First, it makes it very difficult for the system to produce code values close to 0, since *tanh* has a high derivative near 0. Second, this single-layer feed-forward architecture cannot possibly exhibit any kind of *competition* between different subsets of code components whose corresponding basis functions could reconstruct the input equally well. We address the first shortcoming, and will address the second in detail in the next section.

The first shortcoming is addressed by using a shrinkage function similar to the one used in ISTA instead of tanh. Three different such non-linearities have been tested. The first one is simply h_θ, where the vector of thresholds θ is subject to training. One can show that $h_\alpha(x)$ is the best possible function in the situation when the dimensionalities of the input and code are both one, by solving (1). This function is not differentiable at the points $\pm\alpha$. Furthermore, it has a strictly zero region, and if a value falls there, the gradient will be zero, causing no parameter update. The second one is the so-called "double tanh" $g(X) = D(\tanh(X+U) + \tanh(X-U))$, where U is trainable vector, and D a trainable diagonal matrix. This function has the advantage of being smooth and differentiable unlike the shrinkage function (see Figure 2). Thirdly, the ideal form of non-linearity can be obtained through learning: by using a flexible parameterization of the non-linear function training it. We tested this in two ways. In the first case, the non-linear function was parameterized as a weighted sum of univariate Gaussian RBFs with trainable centers and widths. In the second case, the function is piecewise linear, with trainable control points spaced at regular intervals. After training, we find that the learned non-linearity is very similar to the shrinking function (3) with slight roundings of the corners near α, similar to that of $D(\tanh(X+U)+\tanh(X-U))$. The code prediction error obtained by the trainable non-linearities are only slightly better than with the shrinkage function. Hence, we decided to simply use the shrinkage function h_θ in all experiments.

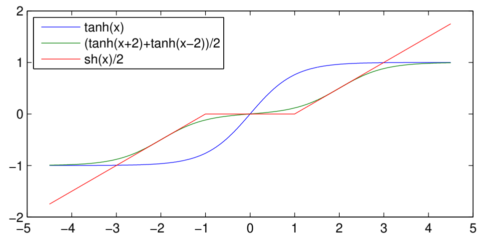

Figure 2. Non-linearities for the baseline encoder described in section 7. The "double tanh" function and the shrinkage function g_θ give the best results. The shrinkage function is used for most experiments described in the paper.

3.2. Mutual Inhibition and Explaining Away

In overcomplete situations, it is necessary for the inference algorithm to allow *the code components to compete* to explain the input. The baseline encoders have no such capability, which is the reason behind their second shortcoming as mentioned above. To illustrate the problem, imagine that two rows of W_e contain two very similar filters, for example two oriented Gabors filters with similar orientations. If the input image contains an edge with an orientation very close to the first filter, the input would be well reconstructed by turning on the code component of the first filter and turning off the second. Unfortunately, the simple encoder will activate both code components equally, and will be incapable of *explaining away* the second component with the first, because *with the baseline encoder, the activations of the two code components are independent given the input.*

The following sections introduce two encoders that include interaction terms between code components, inspired by ISTA and CoD. The basic idea is to use feed-forward networks whose structure correspond to a few steps of ISTA, or a fixed number of steps of CoD. The matrices W_e, S, and the vector θ, instead of being computed from W_d and α will be trained so as to minimize the loss of Equation 4. The methods are dubbed Learning ISTA (LISTA) and Learning CoD (LCOD). Laying out the operations for a few iteration of the ISTA algorithm results in the block diagram (or data flow graph) of Figure 1(b). The encoder architecture can be seen as a sort of time-unfolded recurrent neural network.

Naturally, instead of using learning, we could simply use the matrices and parameters specified by ISTA and CoD and simply terminate the algorithm after a small number of steps, but there is no guarantee that we will obtain the best approximation for the given number of operations. conversely, there will be no guarantee

Algorithm 3 LISTA::fprop

```
LISTA :: fprop(X, Z, W_e, S, θ)
;; Arguments are passed by reference.
;; variables Z(t), C(t) and B are saved for bprop.
B = W_e X; Z(0) = h_θ(B)
for t = 1 to T do
   C(t) = B + S Z(t − 1)
   Z(t) = h_θ(C(t))
end for
Z = Z(T)
```

Algorithm 4 LISTA::bprop

```
LISTA :: bprop(Z*, X, Z, W_e, S, θ, δX, δW_e, δS, δθ)
;; Arguments are passed by reference.
;; Variables Z(t), C(t), and B were saved in fprop.
Initialize: δB = 0; δS = 0; δθ = 0
δZ(T) = (Z(T) − Z*)
for t = T down to 1 do
   δC(t) = h'_θ(C(t)).δZ(t)
   δθ = δθ − sign(C(t)).δC(t)
   δB = δB + δC(t)
   δS = δS + δC(t) Z(t − 1)^T
   δZ(t − 1) = S^T δC(t)
end for
δB = δB + h'_θ(B).δZ(0)
δθ = δθ − sign(B).h'_θ(B) δZ(0)
δW_e = δB X^T; δX = W_e^T δB
```

that the encoders obtained with LISTA and LCoD will converge to the true optimal code if one runs them for more steps than they were trained for.

At first glance, hoping that LISTA and LCoD will beat ISTA and CoD for a small number of steps seems like wishful thinking. But one must keep in mind that we are not seeking to produce approximate sparse code for all possible input vectors, but *only for input vectors drawn from the same distribution as our training samples*. With learning we are carefully carving the solution of a restricted problem of interest, not the general problem. The next two sections describe the LISTA and LCoD architectures and learning procedures in detail.

3.3. Learned Iterative Shrinkage-Thresholding Algorithm (LISTA)

The LISTA encoder takes the precise form of Equation 2 with a fixed number of steps T. The pseudo-code for computing a sparse code using LISTA is given in Algorithm 3, and the block diagram in Figure 1(b).

Learning the parameters $W = (W_e, S, \theta)$ is performed by applying Equation 6 repetitively over the training samples. Computing the gradient $dL(W, X^p)/dW$ is achieved through the back-propagation procedure.

Algorithm 5 LCoD::fprop

```
LCoD :: fprop(X, Z, W_e, S, θ)
;; Arguments are passed by reference.
;; variables e(t), k(t), b(t) and B(T) are saved
B = W_e X; Z = 0;
for t = 1 to T − 1 do
   Z̄ = h_θ(B)
   k = index of largest component of |Z − Z̄|
   k(t) = k, b(t) = B_k; e(t) = Z̄_k − Z_k
   ∀j ∈ [1, m] :   B_j = B_j + S_jk e(t)
   Z_k = Z̄_k
end for
B(T) = B; Z = h_θ(B)
```

One can view the architecture as a time-unfolded recurrent neural network, to which one can apply the equivalent of back-propagation through time (BPTT). More simply, it can be viewed as a feed-forward network in which S is shared over layers. Computing the gradients consists in starting from the output and back-propagating gradients down to the input by multiplying by the Jacobian matrices of the traversed modules, which is a simple application of chain rule: $dL/dZ(t) = dL/dZ(t+1)dZ(t+1)/dZ(t)$, and $dL/dS = \sum_{t=1}^{T} dL/dZ(t)dZ(t)/dS$, since S is shared across time steps. Similar formulas can be applied to compute $dL/d\theta$ and dL/dW_e. The complete back-propagation pseudo-code is given in Algorithm 4. The δ prefix denotes the gradient of L with respect to the variable that follows it. Variables and their associated gradients have the same dimensions. $h'_{\theta(t)}$ denotes the jacobian of h with respect to its input (a square binary diagonal matrix).

3.4. Learned Coordinate Descent (LCoD)

The learned version of coordinate descent follows the same procedure as that described in section 2.2, with the same interpretation of the Figure 1, but with matrices W_e, S and vector θ learned. The code prediction portion of the pseudo-code is given in Algorithm 5. The procedure is identical to algorithm 2, except that some variables are saved in preparation for the subsequent back-propagation procedure. We also see how the last line is useful: if the number of iterations is zero, we get the baseline encoder described in the subsection 3.1. Adding iterations will improve the performance from there.

The pseudo-code of the gradient back-propagation procedure through this LCoD encoder is given in Algorithm 6. Note that what we will back-propagate are technically *sub-gradients*, as the operation that finds the index of the largest change in Z creates kinks in the function (though the function is still continuous, and the kinks have measure zero). Such kinks have little negative effects on stochastic gradient pro-

Algorithm 6 LCoD::bprop

LCoD :: **bprop**$(Z^*, X, Z, W_e, S, \theta, \delta X, \delta W_e, \delta S, \delta\theta)$
;; Arguments are passed by reference.
;; Variables $e(t)$, $k(t)$, $b(t)$ and $B(T)$ were saved
Initialize: $\delta S = 0$; $\delta Z = 0$
$B = B(T)$; $\delta B = h'_\theta(B).(Z - Z^*)$
for $t = T-1$ down to 1 **do**
  $k = k(t)$; $\delta e = \sum_j \delta B_j.S_{jk}$
  $\forall j \in [1,m]: \ \delta S_{jk} = \delta S_{jk} + \delta B_j.e(t)$
  $\delta B_k = \delta B_k + h'_{\theta_k}(b(t))\,(\delta Z_k + \delta e)$
  $\delta\theta_k = \delta\theta_k - \text{sign}(b(t))h'_{\theta_k}(b(t))\,(\delta Z_k + \delta e)$
  $\delta Z_k = -\delta e$
end for
$\delta W_e = \delta B.X^T$; $\delta X = W_e^T.\delta B$

cedures (though they sometimes are fatal to deterministic gradients methods). Explicitly deriving the back-propagation procedure is beyond the scope of this short paper, but suffices to say that writing down the block diagram of the fprop provides a mechanical way to write down the bprop. One must emphasize that each step in the bprop procedure requires only $O(m)$ operations. The amount of storage required for each iteration is also $O(m)$. Apart from $B(T)$, the other saved variables are scalars.

4. Results

In the first set of experiments we compare the performance of different methods on exact sparse code prediction. The data-set consists of a quasi infinite supply image patches of size 10×10 pixels, randomly selected from the Berkeley image database. Each patch is preprocessed to remove its mean and normalize its variance. The patches with small standard deviations were discarded. A sparsity coefficient $\alpha = 0.5$ was used in Equation 1.

The dictionary of basis vectors W_d in (1) was trained by iterating the following standard procedure: (1) get an image patch from the training set X^p; (2) calculate the optimal code Z^{*p} using the CoD Algorithm 2; (3) update W_d with one step of stochastic gradient $W_d \leftarrow W_d - \eta dE_{W_d}E(X^p, Z^{*p})/dW_d$; (4) renormalize the column of W_d to unit norm; (5) iterate. The step size was decreased with a $1/t$ schedule. The procedure resulted in the usual gabor-like filters. We considered two cases, one with $m = 100$ (complete code) and one with $m = 400$ (4 times over-complete code).

Once W_d was trained, the LISTA and LCoD encoders were trained as follows: (1) get an image patch from the training set X^p; (2) calculate the optimal code Z^{*p} using the CoD Algorithm 2; (3) perform fprop through the encoder using either Algorithm 3 or 5 to predict a code; (4) perform bprop through the encoder using Algorithms 4 or 6; (5) update the encoder parameters using the gradient thereby obtained with equation 6; (6) iterate.

Table 1. Prediction error (squared error between the optimal code and the predicted codes) for different non-linearities of the baseline encoder (7), and for LISTA and FISTA. LISTA produces a much better estimate after one iteration than FISTA.

Non-linearity	100 units	400 units
$D\tanh(x)$	8.6	10.7
$D(\tanh(x+u)+\tanh(x-u))$	3.33	4.62
$h_\alpha(x)$	3.29	4.82
LISTA 1 iteration	1.50	2.45
LISTA 3 iterations	0.98	2.12
LISTA 7 iterations	0.52	1.62
FISTA 1 iteration	21.	22.

We compare the various encoders by measuring the squared error between the code predicted by the encoders and the optimal code Z^*. The code prediction error for different non-linearities with the baseline encoder 7 is shown in Table 1. The shrinkage function and the "double tanh" perform similarly, and are both considerably better than the $D\tanh$ non-linearity of (Kavukcuoglu et al., 2008). Interestingly, they are also much better than FISTA with 1 iteration, even though the computation is considerably less.

LISTA The prediction error for LISTA are shown in Table 1 and Figure 3 for a varying depth. We see that the higher the depth, the better the prediction error. A single iteration of LISTA reduces the error by almost a factor of 2 over a simple shrinkage encoder. More interestingly LISTA is stupendously better than FISTA: It takes 18 iterations of FISTA to reach the same error obtained with 1 iteration of LISTA for $m = 100$, and 35 iteration for $m = 400$.

One problem with LISTA is that the multiplication by S matrix takes $O(m^2)$ or $O(mk)$, as opposed to $O(mn)$ for FISTA. To reduce the computational burden at the expense of accuracy we experimented with reduced forms for S with two methods. In the first method, we express S as $S = U_1^T U_2$, where U_1 and U_2 are $q \times m$ matrices, U_1 with normalized rows. The amount of operations is thus reduced by $c_f = 2q/m$. We attempted to enforce $U_2 = U_1^T$ but the results were worse. In the second method, we simply constrain a fraction of elements of S to remain zero during training. The elements that are suppressed are the elements with the smallest absolute value in the FISTA S-matrix $I - W_d^T W_d$. The results are shown in Figure 4. We see that it is more efficient to remove elements than to reduce the rank. Removing about

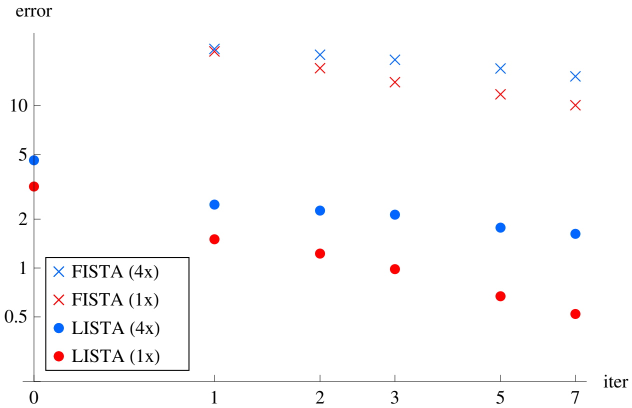

Figure 3. Code prediction error as a function of number of iterations for FISTA (crosses) and for LISTA (dots), for $m = 100$ (red) and $m = 400$ (blue). Note the logarithmic scales. iter $= 0$ corresponds to the baseline trainable encoder with the shrinkage function. It takes 18 iterations of FISTA to reach the error fo LISTA with just one iteration for $m = 100$, and 35 iteration for $m = 400$. Hence one can say that LISTA is roughly 20 times faster than FISTA for approximate solutions.

80% of connections ($c_f = 0.2$) causes a relatively small increase in prediction error from about 1.6 to about 2.0. Removing connections also allows efficient computation of the S matrix multiplication when only the nonzero code units are used.

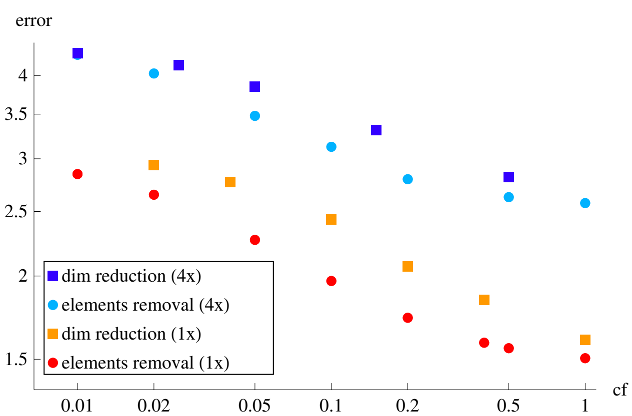

Figure 4. Prediction error for LISTA with one iteration as a function of fraction of operations c_f required relative to a full S matrix. The matrix is reduced using a low rang factorization, or by removing small elements.

LCoD: The prediction results for the learned CoD are shown in the Figure 5. Each iteration costs $O(m)$ operations as opposed to LISTA's $O(m^2)$ or $O(mk)$. The cost of the initial operation $W_e X$ is $O(nm)$. It is remarkable that with only 20 iterations, which adds a tiny additional cost to the initial calculation $W_e X$, and much smaller than a single iteration of FISTA or LISTA, the error is already below 2. It takes 100 iterations of CoD to reach the same error as 5 iterations of LCoD. For a large number of iterations, LCoD loses to CoD when the matrices are initialized randomly, but initializing the matrices with their CoD-prescribed values improves the performance significantly (open circles).

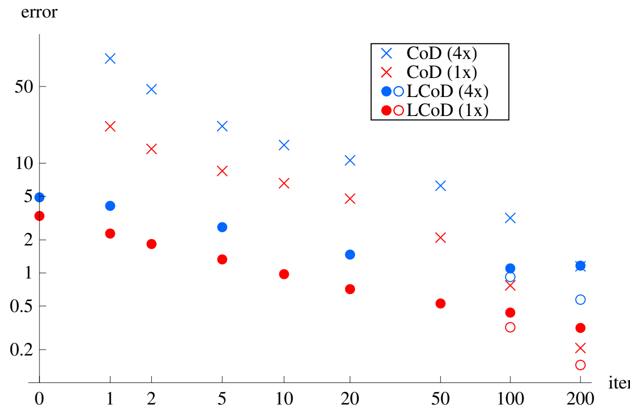

Figure 5. Code prediction errors for CoD and LCoD for varying numbers of iterations. LCoD is about 20 times faster than CoD for small numbers of iterations. Initializing the matrices with their LCoD values before training (open circles) improve the performance in the high iteration regime, but seems to degrade it in the low iteration regime (data not shown).

In the second set of experiments we investigated whether the improvement in prediction error leads to a better recognition performance using the MNIST dataset. In the first experiment, the CoD and LCoD methods with codes of size 784 were trained on the whole $28 \times 28 = 784$ pixel images. In the second one, the CoD and LCoD methods with 256 dimensional codes were trained on 16×16 pixel patches extracted from the MNIST digits. A complete feature vector consisted of 25 concatenated such vectors, extracted from all 16×16 patches shifted by 3 pixels on the input. The features were extracted for all digits using CoD with exact inference, CoD with a fixed number of iterations, and LCoD. Additionally a version of CoD (denoted CoD') used inference with a fixed number of iterations during training of the filters, and used the same number of iterations during test (same complexity as LCoD). A logistic regression classifier was trained on the features thereby obtained.

Classification errors on the test set are shown in Tables 2 and 3. While the error rate decreases with the number of iterations for all methods, the error rate of LCoD with 10 iterations is very close to the optimal (differences in error rates of less than 0.1% are insignificant on MNIST)[1].

[1]cpu times assume efficient implementation of the $W_e X$ that is not available for the argmax of (L)CoD: 1.6x speed up for Table 2 (vector) and 5x for Table 3 (batch).

Table 2. MNIST results with 784-D sparse codes.

iter	cpu	Pred. Error		Classification Error		
		CoD	LCoD	CoD	CoD'	LCoD
0	1	2143	1.07	6.74	-	2.65
1	1.02	370	0.99	5.53	-	2.55
5	1.05	31.7	0.78	6.71	5.04	2.33
10	1.10	12.0	0.66	5.24	4.82	2.32
20	1.21	5.81	0.55	3.77	4.17	2.39
50	1.50	2.14	0.51	2.57	3.54	2.29
conv	4.56	0	-	2.15	2.15	-

Table 3. MNIST results with 25 256-D sparse codes extracted from 16×16 patches every 3 pixels.

iter	cpu	Pred. Error		Classification Error		
		CoD	LCoD	CoD	CoD'	LCoD
0	1	273	0.70	2.24	-	1.66
1	1.15	80.5	0.58	2.3	-	1.60
5	1.55	5.58	0.34	1.82	2.18	1.47
10	2.10	2.75	0.22	1.58	1.99	1.42
20	2.95	1.44	0.14	1.55	1.55	1.42
50	5.45	0.44	0.07	1.46	1.48	1.39
conv	37.25	0	-	1.33	1.33	-

5. Conclusions

We have shown that learning the filters and the mutual inhibition matrices of truncated versions of FISTA and CoD leads to dramatic reduction in the number of iterations to reach a given code prediction error, roughly by a factor of 20 for the low iteration regime. It seems that a small amount of data-specific mutual inhibition is all that is needed to explain away unnecessary components of the code vector. Even if accurate codes are needed, LCoD can be used advantageously to initialize CoD. The method opens the door to the use of sparse feature extraction in real-time vision and pattern recognition systems. In future work, the method will be applied to image restoration and object recognition tasks.

Acknowledgments: this work was supported in part by ONR contract N00014-09-1-0473.: Deep Belief Networks for Nonlinear Analysis.

References

Beck, A. and Teboulle, M. A fast iterative shrinkage-thresholding algorithm with application to wavelet-based image deblurring. *ICASSP'09*, pp. 693–696, 2009.

Chen, S.S., Donoho, D.L., and Saunders, M.A. Atomic decomposition by basis pursuit. *SIAM review*, 43(1): 129–159, 2001.

Daubechies, I, Defrise, M., and De Mol, C. An iterative thresholding algorithm for linear inverse problems with a sparsity constraint. *Comm. on Pure and Applied Mathematics*, 57:1413–1457, 2004.

Donoho, D.L. and Elad, M. Optimally sparse representation in general (nonorthogonal) dictionaries via ℓ^1 minimization. *PNAS*, 100(5):2197–2202, 2003.

Elad, M. and Aharon, M. Image denoising via learned dictionaries and sparse representation. In *CVPR'06*, 2006.

Hale, E.T., Yin, W., and Zhang, Y. Fixed-point continuation for l1-minimization: Methodology and convergence. *SIAM J. on Optimization*, 19:1107, 2008.

Hoyer, P. O. Non-negative matrix factorization with sparseness constraints. *JMLR*, 5:1457–1469, 2004.

Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. What is the best multi-stage architecture for object recognition? In *ICCV'09*. IEEE, 2009.

Kavukcuoglu, Koray, Ranzato, Marc'Aurelio, and LeCun, Yann. Fast inference in sparse coding algorithms with applications to object recognition. Technical Report CBLL-TR-2008-12-01, Computational and Biological Learning Lab, Courant Institute, NYU, 2008.

Lee, H., Battle, A., Raina, R., and Ng, A.Y. Efficient sparse coding algorithms. In *NIPS'06*, 2006.

Lee, H., Chaitanya, E., and Ng, A. Y. Sparse deep belief net model for visual area v2. In *Advances in Neural Information Processing Systems*, 2007.

Lee, H., Grosse, R., Ranganath, R., and Ng, A.Y. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *International Conference on Machine Learning*. ACM New York, 2009.

Li, Y. and Osher, S. Coordinate descent optimization for l1 minimization with application to compressed sensing; a greedy algorithm. *Inverse Problems and Imaging*, 3 (3):487–503, 2009.

Mairal, J., Elad, M., and Sapiro, G. Sparse representation for color image restoration. *IEEE T. Image Processing*, 17(1):53–69, January 2008.

Mairal, J., Bach, F., Ponce, J., and Sapiro, G. Online dictionary learning for sparse coding. In *ICML'09*, 2009.

Olshausen, B.A. and Field, D. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, 381(6583):607–609, 1996.

Ranzato, M., Huang, F.-J., Boureau, Y.-L., and LeCun, Y. Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *CVPR'07*. IEEE, 2007a.

Ranzato, M.-A., Boureau, Y.-L., Chopra, S., and LeCun, Y. A unified energy-based framework for unsupervised learning. In *AI-Stats'07*, 2007b.

Rozell, C.J., Johnson, D.H, Baraniuk, R.G., and Olshausen, B.A. Sparse coding via thresholding and local competition in neural circuits. *Neural Computation*, 20: 2526–2563, 2008.

Vonesch, C. and Unser, M. A fast iterative thresholding algorithm for wavelet-regularized deconvolution. In *IEEE ISBI*, 2007.

Wu, T.T. and Lange, K. Coordinate descent algorithms for lasso penalized regression. *Ann. Appl. Stat*, 2(1): 224–244, 2008.

Yang, Jianchao, Yu, Kai, Gong, Yihong, and Huang, Thomas. Linear spatial pyramid matching using sparse coding for image classification. In *CVPR'09*, 2009.

Yu, Kai, Zhang, Tong, and Gong, Yihong. Nonlinear learning using local coordinate coding. In *NIPS'09*, 2009.