Fast Approximations to Structured Sparse Coding and Applications to Object Classification

Arthur Szlam[1], Karol Gregor[2], and Yann LeCun[3]

[1] City College of New York
[2] Howard Hughes Medical Institute
[3] New York University

Abstract. We describe a method for fast approximation of sparse coding. A given input vector is passed through a binary tree. Each leaf of the tree contains a subset of dictionary elements. The coefficients corresponding to these dictionary elements are allowed to be nonzero and their values are calculated quickly by multiplication with a precomputed pseudoinverse. The tree parameters, the dictionary, and the subsets of the dictionary corresponding to each leaf are learned. In the process of describing this algorithm, we discuss the more general problem of learning the groups in group structured sparse modeling. We show that our method creates good sparse representations by using it in the object recognition framework of [1,2]. Implementing our own fast version of the SIFT descriptor the whole system runs at 20 frames per second on 321×481 sized images on a laptop with a quad-core cpu, while sacrificing very little accuracy on the Caltech 101, Caltech 256, and 15 scenes benchmarks.

1 Introduction

Sparse modeling [3,4] has proven to be a useful framework for signal processing. Each point from a dataset consisting of vectors in a Euclidean space is represented by a vector with only a few nonzero coefficients. Sparse modeling has lead to state of the art algorithms in image denoising, inpainting, supervised learning, and of particular interest here, object recognition. The systems described in [1,2,5,6,7,8] use sparse coding as an integral element. Since the coding is done densely in an image with relatively large dictionaries, this is a computationally expensive part of the recognition system, and a barrier to real time application. The main contribution of this paper is a fast approximate algorithm for finding sparse representations; we use this algorithm to build a system with near state of the art recognition performance that runs in real time. During inference the algorithm uses a tree to assign an input to a group of allowed dictionary elements and then finds the corresponding coefficient values using a cached pseudoinverse. We give an algorithm for learning the tree, the dictionary and the dictionary element assignment, and along the way discuss methods for the more general problem of learning the groups in group structured sparse modeling.

One standard formulation of sparse coding is to consider N d-dimensional real vectors $X = \{x_1, \ldots, x_N\}$ and represent them using N K-dimensional real vectors $Z = \{z_1, \ldots, z_N\}$ using a $d \times k$ dictionary matrix W by solving

$$\text{argmin}_{Z,W} \sum_k ||Wz_k - x_k||^2, \text{ s.t. } ||z_k||_0 \leq q, \tag{1}$$

A. Fitzgibbon et al. (Eds.): ECCV 2012, Part V, LNCS 7576, pp. 200–213, 2012.
© Springer-Verlag Berlin Heidelberg 2012

where $||\cdot||_0$ measures the number of nonzero elements of a vector; each input vector x is thus represented as a vector z with at most q nonzero coefficients. While this problem is not convex, and in fact the problem in the Z variable is NP-hard, there exist algorithms for solving both the problem in Z (e.g. Orthogonal Matching Pursuit, OMP [9]) and the problem in both variables (e.g. K-SVD [4]) that work well in many practical situations.

It is sometimes appropriate to enforce more structure on Z than just sparsity. For example, many authors have noted that the solution to the z minimization in (1) (and its l_1 relaxation) is very unstable in the sense that nearby inputs can have very different coefficients, in part because of the combinatorially large number of possible active sets (i.e. sets of nonzero coordinates of z). This can be a problem in classification tasks. Other times we may know in advance some structure in the data that the coefficients should preserve. Various forms of structured sparsity are explored in [10 11 12 13].

A simple form of structured sparsity is given by specifying a list of L allowable active sets of coefficients (coefficients that are allowed to be nonzero), and some function $g : \mathbb{R}^d \mapsto \{1, ..., L\}$ associating each x to one of the L configurations. An example of this is the output of many subspace clustering algorithms. There each of the L configurations corresponds to the subspace spanned by a basis B_j, $j = 1, \ldots, L$. We can treat the sparse coding dictionary W as concatenation of the bases $W = [B_1 \ldots B_L]$. If each B is of size q then the allowed active sets of coefficients are $\{1, ..., q\}$, $\{q + 1, ..., 2q\}$, etc. This sort of method is used in object recognition in [6].

In this work we will try to learn the L configurations as well as the dictionary. We introduce a LLoyd-like algorithm [14] that alternates between updating the dictionary, updating the assignments of each data point to the groups, and updating the dictionary elements associated to a group via simultaneous OMP (SOMP) [15].

At inference time, we need a fast method for determining which group an x belongs to. This is computationally expensive if there is a large number of groups and one needs check the projection onto each group. However, by specializing the Lloyd type algorithm to the case when each group is composed of a union of (perhaps only one) leaves of a binary decision tree, we will build a fast inference scheme into the learned dictionary. The key idea is that by using SOMP, we can learn which leaves should use which dictionary elements as we train the dictionary. To code an input, we march it down the tree until we arrive at the appropriate leaf. In addition to the decision vectors and thresholds, we will store a lookup table with the active set of each leaf as learned above, and the pseudoinverse of the columns of W corresponding to that active set. Thus after following x down the tree we need only make one matrix multiplication to get the coefficients.

Finally, we would like use these algorithms to build an accurate real time recognition system. We focus on a particular architecture studied in [1 2 6 8]. First, SIFT descriptors are calculated densely over the image. Then (a form of) sparse coding is used to calculate a sparse vector at every location from the corresponding SIFT vector. Then each feature is pooled over a small number of spatial regions and the results are concatenated. Finally the labels are obtained using linear SVM or logistic regression.

We use this pipeline with two modifications. First we write our own fast implementation of the SIFT descriptor. Second we use our fast algorithm for the sparse coding step. The resulting system achieves nearly the same performance as exact sparse coding

calculation but processes 321×481 size images at the rate of 20 frames per second on a laptop computer with a quad core cpu.

The rest of this paper is organized as follows: Section 2, we discuss greedy structured sparse modeling, and describe in depth how to train a model that learns the structure, and that respects a given set of groups given by a tree. In section 3, we show experiments on image patches to qualitatively demonstrate what learned groups look like and then we apply our methods to object recognition.

2 Hashing and Dictionary Learning

2.1 A Simple Form of Structured Dictionary Learning

In this subsection assume we are given a list of L perhaps overlapping groups $G_1, ..., G_L$ of the coefficients of Z. Suppose that there are K dictionary vectors, so that each column of Z is of length K, then $G_i \subset \mathcal{P}(\{1, ..., K\})$, where $\mathcal{P}$ is the set of all subsets of its argument. Our goal is to assign each input vector x to one of the groups and learn the dictionary that would minimize overall energy (specified below). Our solution is a generalization of the LLoyd algorithm [14] for K means or K flats to this setting.

After initializing the dictionary W, we find the distance of each x in X to its projection $P_{G_i}x$ onto the span of W_{G_i} for each i. Each x is associated to the i with the smallest distance

$$x \mapsto \text{argmin}_{i \in \{1,...,L\}} ||P_{G_i}x - x||^2, \tag{2}$$

and we find the coefficients

$$z = (W_{G_i}^T W_{G_i})^{-1} W_{G_i}^T x.$$

Then we update W to be the minimum of the convex problem

$$\text{argmin}_W \sum_x ||Wz - x||^2,$$

and repeat. Each of the subproblems either has an explicit solution or is convex, and so the energy decreases. When the training is finished, we define g to be the function that maps each point in $x \in \mathbb{R}^d$ to the i minimizing the error of the projection of x onto the span of W_{G_i}.

We can also run the same sort of algorithm when in addition to each group G specifying a list of indices, it also specifies a cost for the use of the dictionary elements associated to each index. If we choose an l_2 cost for each of the coefficients, we still get explicit updates and the decrease of energy at each round.

Note that if the number of groups is very large, it may be too costly to find the best group for each x exhaustively. However, we can make a greedy approximation by running a modification of OMP[1]. Here, supposing at iteration s of the OMP we have an active set Ω, the available dictionary elements to add to Ω are the union of all groups containing Ω. It is not necessary to be able to enumerate all the groups to use this method, only to have a subroutine which given $\Omega \subset \{1, ..., k\}$ can return $\bigcup_{\Omega \subset G} G$. However, using this sort of greedy approximation removes the guarantee that the energy decreases at each iteration.

[1] See the next subsection for a more complete description of OMP.

Algorithm 1. SOMP [15]
Finds a single set of nonzero coefficients and their values for a group of inputs X.

function $Z = \mathbf{SOMP}(X, W, q)$
  Initialize: coefficients $Z = 0$, residual $R = X$, active set $\Omega = \emptyset$.
  repeat
    $j = \text{argmax}_i \sum_s |W_i^T R_s|$
    $\Omega = \Omega \bigcup j$
    $Z = \left(W_\Omega^T W_\Omega\right)^{-1} W_\Omega^T X$
    $R = X - WZ$
  until q iterations
end function

2.2 Learning the Groups with Simultaneous Orthogonal Matching Pursuit

In the previous section the groups were specified in advance. If we want to learn the groups, we can add a step in the algorithm. Now instead of taking the list of groups as input, we instead input just the number K of dictionary elements and the number of coefficients allowed per x. After associating to each x the group that best represents it, we can turn around and consider all the x associated to that group. Our task is then to choose a subset of the dictionary that best represents that group. A greedy approximation to this problem in the least squares sense is given by the Simultaneous Orthogonal Matching Pursuit algorithm (SOMP)[15]. This algorithm proceeds just as a standard OMP, but at each iteration, all the x associated to a given group have to choose the next dictionary element added to the group together. To elaborate: an OMP [9]) greedily proceeds towards a solution of the problem

$$\min_z ||Wz - x||^2, \text{ s.t. } |z|_0 \leq q \tag{3}$$

by initializing the residual $r = x$, the active set $\Omega = \emptyset$ and then updating $\Omega = \Omega \bigcup j$, where $j = \text{argmax}_i |W_i^T r|$; z is updated to $\left(W_\Omega^T W_\Omega\right)^{-1} W_\Omega^T x$ on the entries of z specified by Ω, and r is updated to $x - Wz$. The process is repeated q times. The SOMP just works on groups of x simultaneously, instead on each x individually; the j update is the one that is best for the whole group at that stage. See algorithm 1. Unfortunately, because neither OMP nor SOMP is guaranteed to find the optimal solution to the NP hard problems they address, the energy may not decrease at each iteration with this scheme; however, we have found that in practice these methods do usually lead to a decrease in the energy. As in K-means, it may happen that no group uses a dictionary element; in such a situation one can remove a dictionary element from one of the groups, find the residual, and replace the unused dictionary element by the principal component of the residual.

We note that the model presented here can be thought of as a greedy sparse coding version of a "topic model". The dictionary elements act as the words, the x as the documents, and the groups are the topics. The algorithm learns the topics and the dictionary simultaneously.

Algorithm 2. Learning a dictionary and groups

Require: data X, number of dictionary elements K, number of active coefficients per data point q, number of iterations I, and if desired, $g : \mathbb{R}^d \mapsto \{1, ..., M\}$.
repeat
1: Each x chooses a group in $\{1, ..., L\}$ via (2) or by the modified OMP as in Section 2.1. If g is given, all the x in a hash bucket are forced to choose the same group.
2: Each group in $\{1, ..., L\}$ chooses subset of $\{1, ..., K\}$ using $Z = \mathbf{SOMP}(X, W, k)$.
3: Update W, either via K-SVD, or a least squares solve.
until I iterations

2.3 Hashing, Quantization, and Dictionary Learning

The main focus of this work will be choosing a g that can be computed rapidly and learning a dictionary that respects g. We will consider g to be a hash function on $\mathbb{R}^d$ ($g : \mathbb{R}^d \to \{0, 1\}^n$), and hash buckets will be the atomic units of the groups; that is, the groups will either be the hash buckets or will be glued together from the hash buckets. This can be considered a sort of geometric regularization of the sparse coding problem: the active set will be forced to remain constant on the region of $\mathbb{R}^d$ corresponding to each hash bucket.

Once g is chosen, we will learn the dictionary (and perhaps groups) as above, but instead of allowing each x to choose the group that best represents it individually, the x in a hash bucket will need to choose the group that best represents them together on average. We will also try to approximate standard greedy dictionary learning; in this case, there will be one group for every hash bucket. As above, and as with K-means, it may happen that no spatial bucket uses a particular group; in that case we can just pick a bucket at random and use the output of SOMP on that bucket to regenerate the unused group.

Learning how to quantize $\mathbb{R}^d$ is a much studied (but still not completely understood) problem. One common motivation is to build a data structure allowing nearest neighbors from a given data set to be quickly computed. Another common motivation is to use the buckets of the quantization as words to build bag of words feature representations. The relationship between vector quantization and sparse coding has been studied before by many authors, for example [16]. In particular, K-means is simply l_0 sparse coding with only the coefficients 0 and 1 allowed, and only 1 nonzero per x[2].

In this work we will use a 2-means tree [3] with subdivisions along medians to define g. We start by taking the entire data set and running 2-means, obtaining centers c_1 and c_2. We take each data point $x \in X$ and find the angle between x and $c_1 - c_2$; X is divided at the median. We then repeat on each of the pieces, continuing until each

[2] "shape gain coding" allows a non-binary coefficient.

[3] Although perhaps not exactly standard usage, we will call the data structure obtained from binary partitions of $\mathbb{R}^d$ a hash.

piece is within a given distance to its mean, or a set depth p, whichever comes first. We initialize the 2-means with farthest insertion, as in [17]. Note that our experience is that very few iterations are necessary, and really the farthest insertion is sufficient; in fact cutting in random directions (with some additional tricks and randomizations) has been shown to lead to good partitions when the underlying data has a "manifold" structure, see [18]. The number of buckets at the bottom of the tree is upper bounded by 2^p; we will choose p small enough so that it is simple to store a lookup table with the indices into the dictionary for each bucket, as well as the decision vectors for each branch in the tree.

We also could use mappings of the form $g(x) = s(h(Hx + b))$, where H is a $p \times d$ matrix, h is some sort of nonlinearity (e.g. $\tanh$, or $\sin$), b is an offset, and s is a thresholding function [19]. These mappings require less storage and are somewhat simpler to compute for the same bit depth, but on the data sets we work on, they have the disadvantage that many of the buckets are often empty or have very few entries for reasonable p. While this can be remedied by simply gluing (nearly) empty buckets to nearby full buckets and updating the lookup table, we have found the trees to work better. Note also that unlike in nearest neighbor data structures, it is unnecessary for leaf nodes to keep track of spatially nearby leaves that are far away in the tree metric, because all we care about is which dictionary atoms are used at that node.

After building g and training the dictionary, in order compute the coefficients of a new data point x, we pass it though the tree, obtaining $g(x)$. We lookup $g(x)$ in a table, and this gives an index of m columns Ω of W; at this point we solve the linear system $W_\Omega z - x$ to get the outputs. Alternatively, for each group, we can store $(W_\Omega^T W_\Omega)^{-1}$ (or some stable factorization), or $(W_\Omega^T W_\Omega)^{-1} W_\Omega^T$, and just do the requisite matrix multiplications

2.4 Discussion of Related Work

There is now a large literature on structured sparsity. Like this work, [13, 20] use a greedy approach for structured sparse coding based on OMP or CoSaMP. Unlike this work, they have provable recovery properties when the true coefficients respect the structure, and when the dictionaries satisfy certain incoherence properties. On the other hand, those works do not attempt to learn the dictionary, and only discuss the forward problem of finding z from x and W.The works in [10, 11, 12] use an approach to structured sparsity that allows for convex optimization in z. In these works the coefficients are arranged into a predetermined set of groups, and the sparsity term penalizes the number of active groups, rather than the number of active elements; the dictionary is trained to fit the data. None of these works attempt to learn the group structure along with the dictionary.

The idea of clustering the input space and then using a different dictionary for each cluster has appeared many times before. As mentioned in the introduction, a simple example is the K-flats algorithm, or other subspace clustering algorithms [21]. In [22, 23] each cluster gets a representative dictionary, as opposed to a representative subspace; the cluster dictionaries are augmented by a shared dictionary that points in any cluster can use. In all of these works, the subdictionaries serve the dual purpose of determining the

clusters and also finding the coefficients for the data points associated to them. Using multiple dictionaries associated to data clusterings has been successfully applied to object recognition by [6,8]. In those works, the clusters are determined by K-means (or a Gaussian mixture model); in the first, there is a different dictionary for each cluster, and the code is the size of the union of all the subdictionaries, but only the blocks corresponding to the centroids near the input are nonzero. In the second work, the dictionaries for each centroid are the same, but the code is still a concatenation of the codes associated to each centroid (and are set to zero if the input does not belong to that centroid). The current work differs from all of these in two ways. The first is the use of a fast method for associating points to the clusters at test time. The second is allowing richer parts sharing; in our work any cluster can be allowed to use any dictionary element, but must choose only a small number.

In [24] the authors construct a dictionary on the backbone of a hierarchical clustering with fast evaluation. They also use shared parts. However, in that work the part sharing is determined by the tree structure of the clustering, and not learned along with the dictionary elements.

This work also uses ideas similar to those in [25]. In that work, patches from an image were clustered based on patch similarity and pixel location distance, and a simultaneously sparse decomposition was learned so that each cluster was encouraged to use the same active set. In this work, we propose learning the clustering simultaneously with the groups. Our fast algorithm, that we use for object recognition, preclusters the data, as in [25], but does so in such a way that the cluster assignment of new, unseen data can be quickly determined.

Finally we note that other works have explored the idea of accelerating sparse coding by training the dictionary along with an approximation method, e.g. [5,26]. In the first, the approximation is via a single layer feed forward network, and in the second, via a multilayer feed forward network with a shrinkage nonlinearity. This work uses a tree and lookup table instead.

3 Experiments

3.1 What Do the Groups Look Like?

To get a sense of what kind of groups learned from algorithm 2 look like, we train a dictionary on 500,000 8×8 image patches, and view the results. The image patches are drawn from the PASCAL dataset, and their means are removed. We train a dictionary with 256 elements and 512 groups; each group has 5 dictionary elements in it. We train using the batch method, with a K-SVD update for the dictionary.

After training, some of the dictionary elements are used by many groups, and others are used by only a few. The median number of groups using a given element is 6; 47 elements are in exactly 1 group, and 15 are in more than 30. In Figure 1 we display the dictionary ordered by the number of groups containing each element; this number increases in each column and moving to the right. Unsurprisingly, "popular" elements that belong to many groups are low frequency. In this figure we also show the groups containing a few chosen atoms.

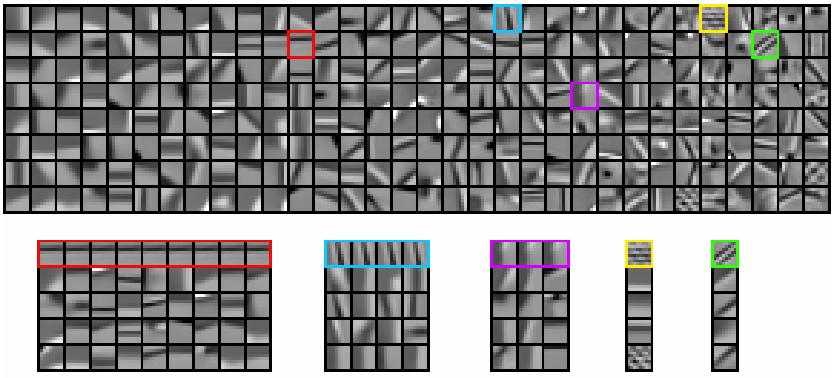

Fig. 1. 256 dictionary atoms in 512 groups trained by algorithm 2 on 500,000 8×8 image patches. The group structure and dictionary were trained simultaneously. The dictionary elements, shown on top, are ordered by popularity (the number of groups they belong to). Underneath, for each dictionary atom in a colored square, we show all of the groups containing it. These groups can be thought of as "topics". Less popular atoms tend to be more specialized.

3.2 Review of the Image Classification Pipeline

Here we will review a standard pipeline for object recognition [1,2], while giving details about our implementation, which streamlines certain components. It consists of the following parts: 1) Calculation of SIFT vectors at every location (SIFT grid). 2) Calculation of the feature vectors for every SIFT vector using the "tree sparse coding" described above. 3) Spatial pyramidal max pooling. 4) Logistic regression or SVM classification. Care is taken to calculate each of these parts efficiently.

SIFT Grid. We run tests with two different implementations of dense SIFT. The first is matlab code by L. Lazebnik [1]. We also use a fast, approximate c++ version that we coded ourselves. The details are as follows:

The x and y Derivatives. We convolve the image with two 5×5 filters that are the x and y derivatives of a Gaussian. This results in an approximation of the values of the x and y derivatives $I_y = dI/dy, I_x = dI/dx$ of the image intensity at every location of the image.

Orientation Histogram. This operation takes the two gradient values I_y, I_x from the previous step and smoothly bins them into histogram of eight orientations. First we calculate the orientation angle $\phi = \arctan(I_y/I_x) + \pi(1 - \text{sign}(x))/2$ and magnitude $m = \sqrt{I_x^2 + I_y^2}$. Let $\phi_h(n) = n\pi/4$, $n = 0, \ldots, 7$. The output of this module is $v(n) = m * \cos(\phi - \phi_h(n))_+^9$ where the $x_+ = x$ if $x > 0$ and 0 otherwise. Most of these operations are computationally expensive and therefore we precompute as much as possible: we bin the I_y and I_x values into 500 bins each and for every combination (500^2 values) we calculate $v(n)$, $n = 0, \ldots, 7$. The bin range is chosen so that the values of I_y and I_x never fall outside the range of the binning so no checks are needed. The output of this module is a feature array with 8 values at every location of the image.

Smooth Subsampling. We subsample the resulting features by two in each direction. Specifically let $v_{n,y,x}$ be the input value obtained from the previous step, where n is the feature number and y, x is the location. The output value will be $u_{n,y,x} = v_{n,2y,2x} + v_{n,2y,2x+1} + v_{n,2y+1,2x} + v_{n,2y+1,2x+1}$. This is efficient since it only involves additions. Note that it results in output values that are essentially four times larger the input values at each location.

Smoothing. We convolve each feature with the filter $[[1,1],[1,1]]$. This is calculated using $u_{n,y,x} = v_{n,y,x} + v_{n,y,x+1} + v_{n,y+1,x} + v_{n,y+1,x+1}$ again resulting in essentially four times larger output values then input values.

Combining and Normalizing into SIFT Vectors. Now we produce the 128 component SIFT vectors from every location of the feature maps from the previous step. At every location (x, y) (of the subsampled features) we first obtain a 128 vector s by concatenating the 8-component vectors at the locations $(x + 2i, y + 2j)$, $i = 1, 2, 3, 4$ and $j = 1, 2, 3, 4$. Then we normalize this vector as follows (as in the code of [1]): if the norm of s is smaller than the threshold $t_h = 1$ we keep s unchanged. If it is larger, we set all of its components that are greater than $.2||s||$ to $.2||s||$, and then scale the result to have norm t_h. The result is placed into the appropriate location of the final $m_y \times m_x \times 128$ vector, where $m_{y,x} \approx n_{y,x}/2$, and where $n_{x,y}$ are the dimensions of the original image. The dimensions are slightly smaller due to boundary effects. This last operation (combining and normalizing) is the most expensive operation in the SIFT grid calculation and we took care to implement it efficiently. Note that in Lazebnik's (and Lowe's original) SIFT the smoothing is done over a larger neighborhood with inputs near the center weighted more then those further. This makes the output more smoothly varying under translations; in our case we used equal weighting over small neighborhoods for computational efficiency.

Hashed Sparse Coding. We used the main procedure of this paper to calculate a sparse feature vector for each SIFT vector. Each such computation consists of depth=16 multiplications of SIFT and tree decision vectors (16 128 dimensional vector multiplies) followed by multiplication of the SIFT vector by the appropriate pseudo-inverse matrix (a 5×128 or 10×128 matrix multiplying a 128 vector) resulting a total of approximately 30 128 vector multiplications. For 2048 dimensional feature vector this compares to $(2048 + 4)$ vector multiplications that are needed for an OMP resulting in an almost 100-fold reduction. Our model was trained on 2×10^6 randomly selected SIFT vectors from the Pascal 2011 dataset.

Spatial Pyramidal Pooling. We used the same spatial pyramidal max pooling as in [2]. Since the feature vectors are in the sparse format the resulting computation is very efficient and negligible compared to either SIFT or tree sparse coding. We calculate the maximum over the features in 1×1, 2×2 and 4×4 regions of the feature vector obtained in the previous step, starting with the 4×4 set of pools, and then pooling the pools for the coarser features. Since calculation is done by looping over all feature vectors and indices and filling the pooled feature vector, the number of computations is of the order of the total number of nonzero features. The final output vector is concatenation of these vectors, resulting in $n_f \times 21$ vector.

Classification. Subsequently a logistic regression classifier is trained on the feature vectors using the liblinear package [27].

Implementation. Each the following operations we implemented using a multicore processing: all steps of the SIFT, finding the group using the tree, and multiplying by pseudo-inverses. In each of these steps separately the image/feature image was split in n_{cores} parts and send to different core. The system was implemented in C++.

3.3 Accuracy on Caltech 101, 15 Scenes, and Caltech 256

We test the accuracy of the standard pipeline with the hashed dictionary and with standard l_0 sparse coding on three object recognition benchmarks, Caltech 101 [28], Caltech 256 [29] and 15 scenes [30]. As mentioned before, for all data sets, we train the hashed dictionary on 2×10^6 randomly selected SIFT vectors from the Pascal 2011 dataset. Caltech 101 consists of 101 image categories and approximately 50 images per category, and Caltech 256 expands this to 256 categories; many classes in these two data sets have more training examples than others and we do the usual normalization of error by class size. We use 30 training examples per class. The 15 scenes database contains 15 categories and 4485 images, and between 200 to 400 images per category. We use 100 training images per class on this data set. For each data set, we run 10 random splits and record the mean and standard deviation of the test error. We record the results in Tables 1 and 2. The first two columns of each table correspond to the hashed sparse coding run with 5 or 10 nonzero entries on Lazebnik's SIFT. The next two columns correspond to the "real time" system, hashed sparse coding run on our approximate SIFT, and the last two columns correspond to OMP, trained and coded with SPAMS[31] on Lazebnik's SIFT. Each row corresponds to the number of atoms in the dictionary. As far as we know, state of the art with single features on grayscale images on Caltech 101 with 30 training examples per category is .773, in [8], and .898 for the 15 scenes, in [32]. Both of these methods use the same basic pipeline as this work, but with variations on the sparse coding and pooling stages; note that our methods are complementary to and can be used in conjunction with the improvements they propose to the coding and pooling without losing speed at test time.

As has been observed by other authors, increasing the size of the dictionary only seems to increase the accuracy. Note that for our method, the only places that the size

Table 1. Caltech 101 accuracies and standard deviations over 10 random splits. The first two columns of each table correspond to the hashed sparse coding run with 5 or 10 nonzero entries, on Lazebnik's SIFT. The next two columns correspond to the "real time" system, hashed sparse coding run on our approximate SIFT, and the last two columns correspond to OMP, trained and coded with SPAMS[31] on Lazebnik's SIFT. Each row corresponds to the number of atoms in the dictionary.

	hashed $m = 5$	hashed $m = 10$	hashed $m = 5$, R.T.	hashed $m = 10$ R.T.	OMP $m = 5$	OMP $m = 10$
$K = 1024$	.722 ± .011	.704 ± .010	.710 ± .007	.697 ± .010	.725 ± .008	.721 ± .010
$K = 2048$	.735 ± .007	.731 ± .011	.723 ± .007	.716 ± .005	.747 ± .008	.738 ± .008
$K = 4096$	.741 ± .011	.740 ± .006	.736 ± .005	.724 ± .004	.754 ± .008	.757 ± .010
$K = 8092$	.751 ± .009		.739 ± .003			

Table 2. 15 scenes accuracies and standard deviations over 10 random splits. The first two columns of each table correspond to the hashed sparse coding run with 5 or 10 nonzero entries on Lazebnik's SIFT. The next two columns correspond to the "real time" system, hashed sparse coding run on our approximate SIFT, and the last two columns correspond to OMP, trained and coded with SPAMS[31] on Lazebnik's SIFT. Each row corresponds to the number of atoms in the dictionary.

	hashed $m = 5$	hashed $m = 10$	hashed $m = 5$, R.T.	hashed $m = 10$ R.T.	OMP $m = 5$	OMP $m = 10$
$K = 1024$	.792 ± .006	.789 ± .004	.786 ± .004	.770 ± .007	.801 ± .006	.802 ± .004
$K = 2048$	.807 ± .006	.800 ± .004	.796 ± .007	.788 ± .007	.814 ± .006	.813 ± .006
$K = 4096$	.810 ± .007	.810 ± .004	.807 ± .003	.804 ± .004	.826 ± .007	.822 ± .007
$K = 8092$	.811 ± .004		.815 ± .004			

Table 3. Caltech 256 accuracies and standard deviations over 10 random splits. The first column of the table corresponds to hashed sparse coding run with 5 nonzero entries, on Lazebnik's SIFT. The next column corresponds to the "real time" system, hashed sparse coding run on our approximate SIFT, and the last column corresponds to OMP, trained and coded with SPAMS[31] on Lazebnik's SIFT. Each row corresponds to the number of atoms in the dictionary.

	hashed $m = 5$	hashed $m = 5$, R.T.	OMP $m = 5$
$K = 1024$	.348 ± .003	.334 ± .002	.362 ± .003
$K = 2048$	.367 ± .003	.356 ± .002	.384 ± .002
$K = 4096$	.381 ± .002	.370 ± .001	.403 ± .004
$K = 8092$	.388 ± .002	.376 ± .003	

Table 4. Speeds of different parts of the system and different dictionary sizes on 321×481 pixel Berkeley dataset images and Caltech 101 images. The times are for single frame in seconds. Frame rates are the inverses and are in frames per second. The total time is the time to process the entire Caltech 101 datasets consisting of 9145 images (minutes:seconds). The Caltech 101 images were pre-resized so that largest side is 300 pixels. The last column is the recognition performance when trained on 30 training images per category. (The speeds vary probably due to disc access and are faster after one or more sweeps through the dataset).

	321×481 pixel images				Caltech 101 (on 4 cores)		
	1 core (s)	4 cores (s)	1 core (fps)	4 cores (fps)	total time (m:s)	(fps)	performance
SIFT	0.039	0.017	25	59	-	-	-
SIFT+TreeSC+pyramid	0.143	0.045	7	22.5	-	-	-
full (1024)	0.145	0.0465	6.9	21	4:01	38	.710 ± .007
full (2048)	0.1473	0.050	6.8	20	4:45	32	.723 ± .007
full (4096)	0.1495	0.052	6.7	19	4:42	32	.736 ± .005
full (8092)	0.155	0.0565	6.4	18	5:35	27	.739 ± .003

of the dictionary affects the computational cost is in training, where we use an SOMP, and in the final linear classification stage. The last component is small for these experiments, but if we wanted to use the system for detection at many locations at an image, it would start to be significant.

3.4 Running Speed

We tested the speed of the full pipeline from image to classification. We show results on images from the Berkeley dataset and Caltech 101. The Berkeley images are 321×481, The Caltech 101 images were resized so that the largest size was at most 300, with the aspect ratio fixed. With 5 nonzero coefficients and depth 16 tree, we get the results in Table 4. The entire dataset of 9145 images in Caltech 101 was processed in 4 minutes and 48 seconds with 2048 features and in 5 minutes and 35 seconds with 8092 features. This corresponds to 31.75fps and 27.3fps respectively.

We also test the speed of just the sparse coding[4]. Coding 15000 SIFT vectors, about one image's worth at this resolution, using this pipeline, with a depth 16 tree and 5 nonzeros per x takes .034 seconds with one core, and .018 with four. In comparison, an OMP using SPAMS with a dictionary of size 1024 and 5 nonzeros costs .898 seconds using four cores. Simply multiplying the dictionary matrix by the data vectors takes .294 seconds. As the size of the dictionary increases, these will increase linearly, but our method will not get any slower. Note also that an OMP with a small number of nonzeros is almost always faster than a Lasso with the regularity parameter tuned to give roughly the same number of nonzeros (and faster still than a Lasso with a smaller regularity term and less sparse representations), which is more commonly used in this architecture [2, 7, 8].

The upshot is that with a few percent points of lost accuracy over the best methods in these works, we are able to run tens of times faster. Indeed, because we can efficiently use large dictionaries, we actually get better results than the original sparse coding/pyramid pooling systems in [1, 2]. Note also that we can use our coding technique in conjuction with improvements in the pipeline, like the macrofeatures in [7], and the local pooling in [8].

Finally we comment on the training time of our coding method. Because we need to compute the preferred active set for each group, it is difficult to use a stochastic gradient type method for the dictionary training. The alternating updates are considerably slower; for example training a 2048 element dictionary with the tree took about an hour, whereas using SPAMS to train dictionary using OMP it took only a few minutes.

4 Conclusion

In this paper we presented a fast approximate sparse coding algorithm and use it to build an accurate real time object recognition system. Our contributions can be summarized into four parts. 1) We describe a general method for learning the groups for greedy structured sparse coding using a generalization of LLoyd's algorithm and SOMP. 2) We use this method to design a fast approximation of greedy sparse coding that uses a tree structure for inference. 3) We give a fast approximate implementation of the SIFT descriptor. 4) These algorithms together allow as to build a real time object recognition system in the framework of [2]. It processes the entire Caltech 101 dataset in under 5 minutes (with images resized so that larger size is 300 pixels). As far as we know this

[4] This test was done on a quad core intel i5 running 64 bit Linux, with 4 gigs of ram; both our code and SPAMS were run as a mex file through Matlab.

is the first time that a fast implementation of this type of system has been put together with comparable accuracy.

We see many possible directions in the future both for improving the group sparse coding algorithm and applying our system to vision. We would like to learn the hash or tree, rather than build it before the dictionary training. We would like to train the system on larger datasets and work on real time object detection (as opposed to classification). At this speed the algorithm allows us to process around 2 million medium sized images (300×400) in 24 hours on a single computer. Object detection, that is, checking for and locating the presence of an object at multiple scales and locations should also be feasible, given that the expensive parts of the pipeline are fast.

Acknowledgments. A.S. would like to thank the NSF for generous support from DMS 1201666.

References

1. Lazebnik, S., Schmid, C., Ponce, J.: Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In: CVPR 2006 (2006) 1, 2, 8, 9, 11
2. Yang, J., Yu, K., Gong, Y., Huang, T.: Linear spatial pyramid matching using sparse coding for image classification. In: CVPR 2009 (2009) 1, 2, 8, 9, 11
3. Olshausen, B., Field, D.: Emergence of simple-cell receptive field properties by learning a sparse code for natural images. Nature 381, 607–609 (1996) 1
4. Aharon, M., Elad, M., Bruckstein, A.: K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation. IEEE Transactions on Signal Processing 54, 4311–4322 (2006) 1, 2
5. Kavukcuoglu, K., Ranzato, M., LeCun, Y.: 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) 1, 7
6. Yang, J., Yu, K., Huang, T.: Efficient Highly Over-Complete Sparse Coding Using a Mixture Model. In: Daniilidis, K., Maragos, P., Paragios, N. (eds.) ECCV 2010, Part V. LNCS, vol. 6315, pp. 113–126. Springer, Heidelberg (2010) 1, 2, 7
7. Boureau, Y., Bach, F., LeCun, Y., Ponce, J.: Learning mid-level features for recognition. In: Proc. International Conference on Computer Vision and Pattern Recognition (CVPR 2010). IEEE (2010) 1, 11
8. Boureau, Y., Roux, N.L., Bach, F., Ponce, J., LeCun, Y.: Ask the locals: multi-way local pooling for image recognition. In: International Conference on Computer Vision (2011) 1, 2, 7, 10, 11
9. Pati, Y.C., Rezaiifar, R., Rezaiifar, Y.C.P.R., Krishnaprasad, P.S.: Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In: Proceedings of the 27 th Annual Asilomar Conference on Signals, Systems, and Computers, pp. 40–44 (1993) 2, 4
10. Jenatton, R., Mairal, J., Obozinski, G., Bach, F.: Proximal methods for sparse hierarchical dictionary learning. In: International Conference on Machine Learning, ICML (2010) 2, 6
11. Kim, S., Xing, E.P.: Tree-guided group lasso for multi-task regression with structured sparsity. In: ICML, pp. 543–550 (2010) 2, 6
12. Jacob, L., Obozinski, G., Vert, J.P.: Group lasso with overlap and graph lasso. In: Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, pp. 433–440. ACM, New York (2009) 2, 6

13. Baraniuk, R.G., Cevher, V., Duarte, M.F., Hegde, C.: Model-Based Compressive Sensing (2009) 2, 6
14. Lloyd, S.P.: Least squares quantization in pcm. IEEE Transactions on Information Theory 28, 129–137 (1982) 2, 3
15. Gilbert, A.C., Strauss, M.J., Tropp, J.A.: Simultaneous Sparse Approximation via Greedy Pursuit. IEEE Trans. Acoust. Speech Signal Process. 5, 721–724 (2005) 2, 4
16. Tropp, J.: Topics in Sparse Approximation. PhD thesis, University of Texas at Austin, Computational and Applied Mathematics (2004) 5
17. Ostrovsky, R., Rabani, Y., Schulman, L., Swamy, C.: The effectiveness of lloyd-type methods for the k-means problem. In: FOCS 2006 (2006) 6
18. Dasgupta, S., Freund, Y.: Random projection trees and low dimensional manifolds. In: STOC 2008 (2008) 6
19. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing, pp. 518–529 (1999) 6
20. Huang, J., Zhang, T., Metaxas, D.N.: Learning with structured sparsity. In: ICML, p. 53 (2009) 6
21. Vidal, R.: Subspace clustering. IEEE Signal Processing Magazine 28, 52–68 (2011) 6
22. Wang, F., Lee, N., Sun, J., Hu, J., Ebadollahi, S.: In: Burgard, W., Roth, D. (eds.) AAAI. AAAI Press (2011) 6
23. Ramírez, I., Sprechmann, P., Sapiro, G.: Classification and clustering via dictionary learning with structured incoherence and shared features. In: CVPR, pp. 3501–3508 (2010) 6
24. Allard, W., Chen, G., Maggioni, M.: Multiscale geometric methods for data sets II: Geometric multi-resolution analysis. Applied and Computational Harmonic Analysis 32, 435–462 (2012) 7
25. Mairal, J., Bach, F., Ponce, J., Sapiro, G., Zisserman, A.: Non-local sparse models for image restoration. In: International Conference on Computer Vision (2009) 7
26. Gregor, K., LeCun, Y.: Learning fast approximations of sparse coding. In: International Conference on Machine Learning, ICML (2010) 7
27. Fan, R.E., Chang, K.W., Hsieh, C.J., Wang, X.R., Lin, C.J.: LIBLINEAR: A library for large linear classification. Journal of Machine Learning Research 9, 1871–1874 (2008) 10
28. Fei-Fei, L., Fergus, R., Perona, P.: Caltech 101, (http://www.vision.caltech.edu/Image_Datasets/Caltech101/) 10
29. Griffin, G., Holub, A., Perona, P.: Caltech-256 object category dataset. Technical Report 7694, California Institute of Technology (2007) 10
30. Lazebnik, S., Schmid, C., Ponce, J., Li, F., Oliva, A.: 15 scenes, (http://www-cvr.ai.uiuc.edu/ponce_grp/data/) 10
31. Mairal, J.: SPAMS sparse coding toolbox, (http://www.di.ens.fr/willow/SPAMS/) 10, 12
32. Gao, S., Tsang, I.W.-H., Zhao, L.T.C., Local, P.: Local features are not lonely-laplacian sparse coding for image classification. In: CVPR 2010 (2010) 10