Learning Feature Representations with K-means

Adam Coates and Andrew Y. Ng

Stanford University, Stanford CA 94306, USA
{acoates,ang}@cs.stanford.edu

Originally published in: G. Montavon, G. B. Orr, K.-R. Müller (Eds.), Neural Networks: Tricks of the Trade, 2nd edn, Springer LNCS 7700, 2012.

Abstract. Many algorithms are available to learn deep hierarchies of features from unlabeled data, especially images. In many cases, these algorithms involve multi-layered networks of features (e.g., neural networks) that are sometimes tricky to train and tune and are difficult to scale up to many machines effectively. Recently, it has been found that K-means clustering can be used as a fast alternative training method. The main advantage of this approach is that it is very fast and easily implemented at large scale. On the other hand, employing this method in practice is not completely trivial: K-means has several limitations, and care must be taken to combine the right ingredients to get the system to work well. This chapter will summarize recent results and technical tricks that are needed to make effective use of K-means clustering for learning large-scale representations of images. We will also connect these results to other well-known algorithms to make clear when K-means can be most useful and convey intuitions about its behavior that are useful for debugging and engineering new systems.

1 Introduction

A major goal in machine learning is to learn deep hierarchies of features for other tasks. For instance, given a set of unlabeled images, many current algorithms seek to greedily learn successive layers of features that will make subsequent classification tasks (e.g., object recognition) easier to accomplish. A typical approach taken in the literature is to use an unsupervised learning algorithm to train a model of the unlabeled data and then use the results to extract interesting features from the data [35,21,31]. Depending on the choice of unsupervised learning scheme, it is sometimes difficult to make these systems work well. There can be many hyper-parameters and not much intuition for how to tune them. More recently, we have found that using K-means clustering as the unsupervised learning module in these types of "feature learning" pipelines can lead to excellent results, often rivaling state-of-the-art systems [11]. In this chapter, we will review some of this work with added notes on useful tricks and observations that are helpful for building large-scale feature learning systems.

K-means has already been identified as a successful method to learn features from images by computer vision researchers. The popular "bag of features" model [13,28] from the computer vision community is very similar to the pipeline that we will use in this chapter, and many conclusions here are similar to those identified by vision researchers [18,1]. In this chapter, however, we will focus on the ingredients needed to make K-means work well in a setting similar to that used by other deep learning and feature learning systems: learning directly from raw inputs (pixel intensities) and building multi-layered hierarchies, as well as connecting K-means to other well-known feature learning systems.

The classic K-means clustering algorithm finds cluster centroids that minimize the distance between data points and the nearest centroid. Also called "vector quantization", K-means can be viewed as a way of constructing a "dictionary" $\mathcal{D} \in \mathcal{R}^{n \times k}$ of k vectors so that a data vector $x^{(i)} \in \mathcal{R}^n$, $i = 1, \ldots, m$ can be mapped to a code vector that minimizes the error in reconstruction. In this chapter, we will use a modified version of K-means (sometimes called "gain shape" vector quantization [41], or "spherical K-means" [14]) that finds $\mathcal{D}$ according to:

$$\begin{aligned} \underset{\mathcal{D},s}{\text{minimize}} & \sum_i ||\mathcal{D}s^{(i)} - x^{(i)}||_2^2 \\ \text{subject to } & ||s^{(i)}||_0 \leq 1, \forall i \\ \text{and } & ||\mathcal{D}^{(j)}||_2 = 1, \forall j \end{aligned}$$

where $s^{(i)}$ is a "code vector" associated with the input $x^{(i)}$, and $\mathcal{D}^{(j)}$ is the j'th column of the dictionary $\mathcal{D}$. The goal here is to find a dictionary $\mathcal{D}$ and a new representation, $s^{(i)}$, of each example $x^{(i)}$ that satisfies several criteria. First, given $s^{(i)}$ and $\mathcal{D}$, we should be able to reconstruct the original $x^{(i)}$ well; in particular, we aim to minimize the squared difference between $x^{(i)}$ and its corresponding reconstruction $\mathcal{D}s^{(i)}$. This goal is optimized under two constraints. The first constraint, $||s^{(i)}||_0 \leq 1$, means that each $s^{(i)}$ is constrained to have at most one non-zero entry. Thus we are searching not only for a new representation of $x^{(i)}$ that preserves it as well as possible, but also for a very simple or parsimonious representation. The second constraint requires that each dictionary column have unit length, preventing them from becoming arbitrarily large or small. Otherwise we could arbitrarily rescale $\mathcal{D}^{(j)}$ and the corresponding $s_j^{(i)}$ without effect.

This algorithm is very similar in spirit to other algorithms for learning efficient coding schemes, such as sparse coding [34,17]:

$$\begin{aligned} \underset{\mathcal{D},s}{\text{minimize}} & \sum_i ||\mathcal{D}s^{(i)} - x^{(i)}||_2^2 + \lambda||s^{(i)}||_1 \\ \text{subject to } & ||D^{(j)}||_2 = 1, \forall j. \end{aligned}$$

Sparse coding optimizes the same type of reconstruction objective, but constrains the complexity of $s^{(i)}$ by adding a penalty $\lambda||s^{(i)}||_1$ that encourages $s^{(i)}$ to be sparse. This is similar to the constraint used by K-means ($||s^{(i)}||_0 \leq 1$), but allows more than one non-zero entry in each $s^{(i)}$, enabling a much more accurate representation of each $x^{(i)}$ while still requiring each $s^{(i)}$ to be simple.

From their descriptions above, it is no surprise that K-means and more sophisticated dictionary-learning schemes like sparse coding are often interchangeable—differing in their optimization objectives, but producing code vectors $s^{(i)}$ and dictionaries $\mathcal{D}$ that accomplish similar goals. Empirically though, sparse coding appears to be a better performer in many applications. For instance, replacing K-means with sparse coding in the classic bag-of-features model has been shown to significantly improve image recognition results [39]. Despite its simplicity, however, K-means is still a very useful algorithm for learning features due to its speed and scalability. Sparse coding requires us to solve a convex optimization problem [36,15,32] for every $s^{(i)}$ repeatedly during the learning procedure and thus is very expensive to deploy at large scale. For K-means, by contrast, the optimal $s^{(i)}$ used in the algorithm above is simply:

$$s_j^{(i)} = \begin{cases} \mathcal{D}^{(j)\top} x^{(i)} & \text{if } j == \arg\max_l |\mathcal{D}^{(l)\top} x^{(i)}| \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

Because this can be done very quickly (and solving for $\mathcal{D}$ given s is also easy), we can train very large dictionaries rapidly by alternating optimization of $\mathcal{D}$ and s. As well, K-means does not have any parameters requiring tuning other than k, the number of centroids, making it relatively easy to get working. The surprise is that large dictionaries learned by K-means often work very well in practice provided we mix in a few other ingredients that are less commonly documented in other works. This chapter is about what these ingredients are as well as some intuition about why they are needed and how they affect results. For most of this work, we will use images (or image patches) as input data to the algorithm, but the basic principles are applicable to other types of data as well.

2 Data, Pre-processing and Initialization

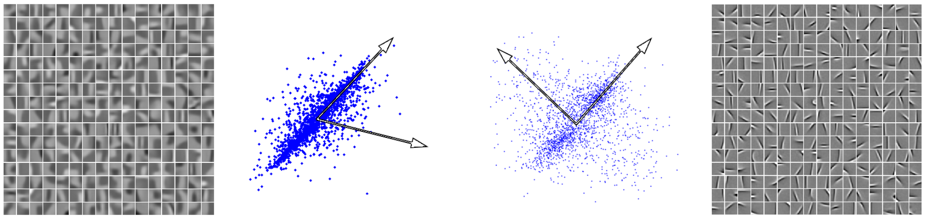
We will begin with a dataset composed of small image patches. For concreteness, we will work with 16-by-16 pixel grayscale patches represented as a vector of 256 pixel intensities (i.e., $x^{(i)} \in \mathcal{R}^{256}$), but color patches can also be used similarly. These patches can be collected from unlabeled imagery by cropping out random 16-by-16 chunks. In order to build a "complete" dictionary (i.e., a dictionary with at least 256 centroids), we should ensure that there will be enough patches so that each cluster can claim a reasonable number of inputs. For 16-by-16 gray patches, $m = 100,000$ patches is enough. In practice, we will often need *more* data to train a K-means dictionary than is necessary for other algorithms (e.g., sparse coding), since each data point contributes to just 1 centroid in the end. Usually the added expense is easily offset by the speed of training. For notational convenience, we will assume that our data points are packed into the columns of a matrix $X \in \mathcal{R}^{n\times m}$. (Similarly, we will denote by S the matrix whose columns are the code vectors $s^{(i)}$ from Eq. (1).)

2.1 Pre-processing

Before running a learning algorithm on our input data points $x^{(i)}$, it is useful to normalize the brightness and contrast of the patches. That is, for each $x^{(i)}$ we subtract out the mean of the intensities and divide by the standard deviation. A small value is added to the variance before division to avoid divide by zero and also suppress noise. For pixel intensities in the range $[0, 255]$, adding 10 to the variance is often a good starting point:

$$x^{(i)} = \frac{\tilde{x}^{(i)} - \text{mean}(\tilde{x}^{(i)})}{\sqrt{\text{var}(\tilde{x}^{(i)}) + 10}}$$

where $\tilde{x}^{(i)}$ are unnormalized patches and "mean" and "var" are the mean and variance of the elements of $\tilde{x}^{(i)}$.



(a) (b) (c)

Fig. 1: (a) Centroids learned by K-means from natural images without whitening. (b) A cartoon depicting the effect of whitening on the K-means solution. Left: unwhitened data, where the centroids tend to be biased by the correlated data. Right: whitened data, where centroids are more orthogonal. (c) Centroids learned from whitened image patches.

After normalization, we can try to run K-means on the new input patches. The centroids that are obtained (i.e., the columns of the dictionary $\mathcal{D}$) are visualized as patches in Figure 1a. It can be seen that K-means tends to learn low-frequency edge-like centroids. This result has been reproduced many times in the past [16,37,2]. Unfortunately, it turns out that these centroids tend to work poorly in recognition tasks [11]. One explanation for this result is that the correlations between nearby pixels (i.e., low-frequency variations in the images) tend to be very strong even after brightness and contrast normalization. In the presence of these correlations, K-means tends to generate many highly correlated centroids rather than spreading the centroids out to span the data more evenly. A cartoon depicting this problem is shown on the left of Figure 1b. To remedy this situation, one should use whitening (also called "sphering") to rescale the input data to remove these correlations [22]. This tends to cause K-means to allocate more centroids in the orthogonal directions, as shown on the right of Figure 1b.

A simple choice of whitening transform is the ZCA whitening transform. If $VDV^\top = \mathrm{cov}(x)$ is the eigenvalue decomposition of the covariance of the data points x, then the whitened points are computed as $V(D+\epsilon_{\mathrm{zca}}I)^{-1/2}V^\top x$, where ϵ_{zca} is a small constant. For contrast-normalized data, setting ϵ_{zca} to 0.01 for 16-by-16 pixel patches, or 0.1 for 8-by-8 pixel patches is a good starting point. Note that setting this number too small can cause high-frequency noise to be amplified and make learning more difficult. Since rotating the data does not alter the behavior of K-means, one can also use other whitening transforms such as PCA whitening (which differ from ZCA only by a rotation).

Running K-means on whitened image patches yields sharper edge features similar to those discovered by sparse coding, ICA, and others as seen in Figure 1c. This procedure of normalization, whitening, and K-means clustering is an effective "off the shelf" unsupervised learning module that can serve in many feature-learning roles. From this point forward, we will assume that whenever we apply K-means to new data that they are normalized and whitened as described here. But keep in mind that proper choices of the ϵ parameters for normalization and whitening can sometimes require adjustment for new data sources. Though these are likely best set by cross validation, they can often be tuned visually (e.g., to yield image patches with high contrast, not too much noise, and not too much low-frequency undulation).

2.2 Initialization

The usual K-means clustering algorithm is known to require a number of small tweaks to avoid common problems like empty clusters. One important consideration is the initialization of the centroids. Though it is common to initialize K-means to randomly-chosen examples drawn from the data, this has been found to be a poor choice. It is possible that images tend to group too densely in some areas, and thus initializing K-means with randomly chosen patches leads to a large number of centroids starting close together. Many of these centroids ultimately end up becoming near-empty clusters, as a single cluster accumulates all of the data points located within a dense area. Instead, it is better to randomly initialize the centroids from a Normal distribution and then normalize them to unit length. Note that because of the whitening stage, we expect that the important components of our data have already been rescaled to a more or less spherical distribution, so initializing to random vectors on a sphere is not a terrible starting point.

Other well-known heuristics for improving the behavior of K-means can be useful. For instance, heuristics for reinitializing empty clusters are commonly used in other implementations. In practice, the initialization scheme above works relatively well for image data. When empty clusters do occur, reinitializing the centroids with random examples is usually sufficient, but this is rarely necessary.[1] In fact, for a sufficiently scalable implementation, we can often train many centroids and simply throw away clusters that have too few data points.

Another minor tweak that improves behavior is to use damped updates of the centroids. Specifically, at each iteration we compute new centroids according to:

$$\begin{aligned}
\mathcal{D}_{\text{new}} &:= \arg\min_{\mathcal{D}} ||\mathcal{D}S - X||_2^2 + ||\mathcal{D} - \mathcal{D}_{\text{old}}||_2^2 \\
&= (SS^\top + I)^{-1}(XS^\top + \mathcal{D}_{\text{old}}) \\
&\propto XS^\top + \mathcal{D}_{\text{old}} \\
\mathcal{D}_{\text{new}} &:= \text{normalize}(\mathcal{D}_{\text{new}}).
\end{aligned}$$

Note that this form of damping does not affect "big" clusters very much (the j'th column of $XS^\top$ will be large compared to $D^{(j)}_{\text{old}}$) and only serves to prevent small clusters from being pulled too far in a single iteration.

Including the initialization and pre-processing, the full K-means training routine presented above is summarized here:

1. Normalize inputs:

$$x^{(i)} := \frac{x^{(i)} - \text{mean}(x^{(i)})}{\sqrt{\text{var}(x^{(i)}) + \epsilon_{\text{norm}}}}, \forall i$$

2. Whiten inputs:

$$\begin{aligned}
[V, D] &:= \text{eig}(\text{cov}(x));\ \ // \text{ So } VDV^\top = \text{cov}(x) \\
x^{(i)} &:= V(D + \epsilon_{\text{zca}} I)^{-1/2} V^\top x^{(i)}, \forall i
\end{aligned}$$

3. Loop until convergence (typically 10 iterations is enough):

$$\begin{aligned}
s_j^{(i)} &:= \begin{cases} \mathcal{D}^{(j)\top} x^{(i)} & \text{if } j == \arg\max_l |\mathcal{D}^{(l)\top} x^{(i)}| \\ 0 & \text{otherwise.} \end{cases} \quad \forall j, i \\
\mathcal{D} &:= XS^\top + \mathcal{D} \\
\mathcal{D}^{(j)} &:= \mathcal{D}^{(j)} / ||D^{(j)}||_2 \forall j
\end{aligned}$$

[1] Often, a large number of empty clusters indicates that the whitening or normalization parameters are improperly tuned, or the data is too high-dimensional for K-means to be successful.

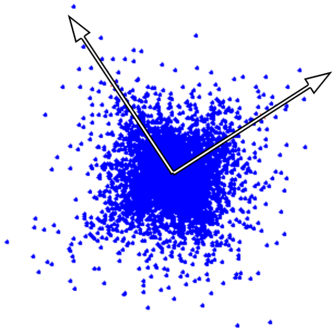


Fig. 2: The result of running spherical K-means on points sampled from a heavy-tailed distribution. The K-means "centroids" tend to point in the directions of the tails.

3 Comparison to Sparse Feature Learning

As was shown above, K-means learns oriented edge-like features when applied to natural images, much like ICA [23] or sparse coding [34]. An important practical question is whether this is accidental (e.g., because edges are so common that learning "exemplars" from images is enough to find them) or whether this implies that K-means can perform a type of sparse decomposition similar to ICA. When we attempt to apply K-means to other types of data such as audio or features computed by lower layers in a deep network, it is important to understand to what extent this clustering algorithm mimics well-known projection methods like ICA and what the limitations are. It is clear that because each $s^{(i)}$ is allowed to contain only a single non-zero entry, K-means tries to learn centroids that single-handedly explain an entire input image. It is thus not guaranteed that the learned centroids will always be like the filters produced by sparse coding or ICA. These algorithms learn genuine "distributed" representations where a single image can be explained jointly by multiple columns of the dictionary instead of just one. Nevertheless, empirically it turns out that K-means *does* tend to discover sparse projections of the data under the right conditions. Because of this property, we can often use the learned dictionary in a manner similar to the dictionaries or filters learned by other algorithms that explicitly search for sparse, distributed representations.

One intuition for why this tends to occur can be seen in a simple low-dimensional example. Consider the case where our data is sampled from two independent, heavy-tailed (sparse) random variables. After normalization, the data will be most dense near the coordinate axes, and less dense in the quadrants between them. As a result, while K-means will tend to represent many points very badly, training 2 centroids on this distribution will tend to yield a

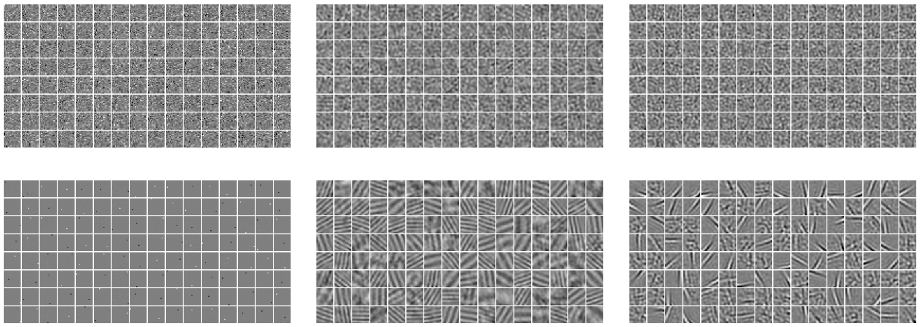


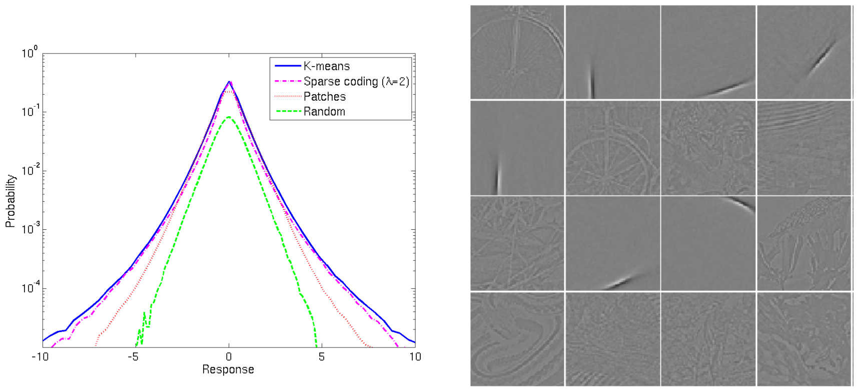
Fig. 3: Top: Three different sparse distributions of images. Bottom: Filters (centroids) identified by K-means with a complete (256-centroid) dictionary.

dictionary of basis vectors pointing in the direction of the tails (i.e., in the sparse directions). This result is illustrated in Figure 2.

If the data dimensionality is not too high (e.g., a hundred or so) this "tail-seeking" phenomenon also shows up in more complicated scenarios. Figure 3 shows examples of three sets of images generated from sparse sources. On the left (at top), are 16-by-16 images with pixels sampled from independent Laplace distributions (sparse pixel intensities). In the middle (top) are images composed of a sparse combination of gratings and at right (top) a sparse combination of non-orthogonal gabor filters. The bottom row shows the result of learning 256 centroids with K-means from 500000 examples drawn from each distribution. It can be seen that K-means does, in fact, roughly recover the sparse projections we would expect. A similar experiment appears in [34] to demonstrate the source-separation ability of sparse coding—yet K-means tends to recover the same filters even though these filters are clearly not especially similar to individual images. That is, K-means can potentially do more than merely recover "exemplar" images from the input distribution.

When applied to natural images, it is evident that the learned centroids $\mathcal{D}^{(j)}$ (as in Figure 1c) are relatively sparse projections of the data. Figure 4a shows a histogram of responses resulting from projecting randomly chosen (whitened) image patches onto 4 different filters. The 4 filters used are: (i) a centroid learned by K-means, (ii) a basis vector trained via sparse coding, (iii) a randomly selected image patch, and (iv) a randomly initialized filter. In the figure, it can be seen that using the K-means-trained centroid as a linear filter gives us a very sparse projection of the data. Thus, it appears that relative to other simple choices K-means does tend to seek out very sparse projections of the data, even though its formulation as a clustering algorithm does not aim to do this explicitly.

Despite this empirical similarity to ICA and sparse coding, K-means does have a major drawback: it turns out that its ability to discover sparse directions in the data depends heavily on the dimensionality of the input and the quantity of data. In particular, as the dimensionality increases, we need increasingly large



(a) (b)

Fig. 4: (a) A comparison of the distribution of linear filter responses when filters obtained by 4 different methods are applied to natural image patches. K-means tends to learn filters with very sparse response characteristics similar to sparse coding—much more sparse than randomly chosen patches or randomly initialized filters. (b) "Centroids" learned from 64-by-64 pixel patches. At this scale, K-means becomes difficult to apply as many clusters become empty or singletons.

quantities of data to get clean results. For instance, to obtain the results above we had to use a very large number of examples. We can obtain similar results easily with sparse coding with just 10000 examples. For very large patches, we must use even more. Figure 4b shows the results of running K-means on 500000 64-by-64 patches—note that while we can capture a few edges, most of the clusters are composed of a single patch from the dataset. At this scale, empty or near-empty clusters become far more common and extremely large amounts of data are needed to make K-means work well. This tradeoff is the main driver in deciding when and how to employ K-means: depending on the dimensionality of the input, a certain amount of data will be required (typically much more than is needed for similar results from sparse coding). For modest dimensionalities (e.g., hundreds of inputs), this tradeoff can be advantageous because the additional data requirements do not outweigh the very large constant-factor speedup that is gained by training with K-means. For very high dimensionalities, however, it may well be the case that another algorithm like sparse coding works better or even faster.

4 Application to Image Recognition

The above discussion has provided the basic ingredients needed to turn K-means into a simple feature learning method. Given a batch of unlabeled data, we can now learn a dictionary $\mathcal{D}$ whose columns yield more or less sparse projections of the data points. Just as with sparse coding, we can use the learned dictionary (centroids) to define features for a supervised learning task [35]. A typical pipeline used for image recognition applications (that is easy to use with learned features) is based on the classic spatial pyramid model developed in the computer vision literature [13,28,39,11]. It is similar in many ways to a single-layered convolutional neural network [29,30]. The main components of this pipeline are: (i) the unsupervised training algorithm (in this case, K-means), which generates a bank of filters $\mathcal{D}$, (ii) a function $f : \mathcal{R}^n \to \mathcal{R}^k$ that maps an input image patch $x \in \mathcal{R}^n$ to a feature vector $y \in \mathcal{R}^k$ given the dictionary of k filters. For instance, we could choose $f(x; \mathcal{D}) = g(\mathcal{D}^\top x)$ for an element-wise nonlinear function $g(\cdot)$.

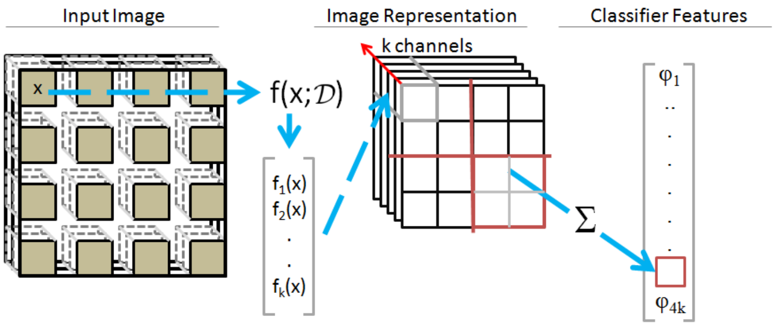


Fig. 5: A standard image recognition pipeline used in conjunction with K-means dictionary learning.

Using the learned feature extractor $f(x; \mathcal{D})$, given any p-by-p pixel image patch, we can now compute a representation $y \in \mathcal{R}^k$ for that patch. We can thus define a (single layer) representation of the entire image by applying the function f to many sub-patches. Specifically, given an image of w-by-w pixels, we define a $(w-p+1)$-by-$(w-p+1)$-by-k array of features by computing the representation y for every p-by-p "sub-patch" of the input image. For computational efficiency, we may also "step" our p-by-p feature extractor across the image with some step-size (or "stride") greater than 1. This is illustrated in Figure 5.

Before classification, we typically reduce the dimensionality of the image representation by pooling. For a step size of 1 pixel, our feature extractor produces a $(w-p+1)$-by-$(w-p+1)$-by-k representation. We can reduce this by summing (or applying some other reduction, e.g., max) over local regions of the feature responses. Once we have the pooled responses, we could attempt to learn higher-level features by applying K-means again (this is the approach pursued by [1]), or just use all of the features directly with a standard linear classification algorithm (e.g., SVM).

4.1 Parameters

The processing pipeline above has a large number of tunable parameters, such as the patch size p, the choice of $f(x;\mathcal{D})$, and the step size. It turns out that getting these parameters set correctly can make a major difference in performance for practical applications. In fact, these parameters often have a bigger influence on classification performance than the training algorithm itself. When we are unhappy with performance, searching for better choices of these parameters can often be more beneficial than trying to modify our learning algorithm [11,18]. Here we will briefly summarize current advice on how to choose these parameters.

First, when we use K-means to train the filter bank $\mathcal{D}$, we noted previously that the input dimensionality can significantly influence the data requirements and success of training. Thus, in addition to other effects it may have on classification performance, it is important to choose the patch size p wisely. If p is too large (e.g., 64 pixels, as in Figure 4b), then K-means may yield poor results. Though this situation can be debugged visually for applications to image data, it is much more difficult to know when K-means is doing well when it is applied to other kinds of data such as when training a multi-layered network where the higher-level features are hard to visualize. For this reason, it is recommended that p be chosen by cross validation or it should be set so that the dimensionality of the data passed to K-means is kept small (typically not more than a few hundred, depending on the amount of data used). For image patches, 6-by-6 or 8-by-8 pixel (color or gray) patches work consistently well in the pipeline outlined above.

Depending on the choice of pooling method, the step size and pooling regions may need to be chosen differently. There is a significant body of work covering these areas [6,5,4,12,24,18]. In our own experience, for single layers of features, average pooling over a few equally-sized regions (e.g., a 2-by-2 or 3-by-3 grid) can work very well in practice and is a good “first try” for image recognition.

Finally, the number of features k learned by the algorithm has a significant influence on the results. It has been observed several times [18,11] that learning large numbers of features can substantially improve supervised classification results. Indeed, it is frequently best to set k as large as compute resources will allow, considering data requirements. Though performance typically asymptotes as k becomes extremely large, increasing the size of k is a very effective way to squeeze out a bit of extra performance from an already-built system. This trend, combined with the fact that K-means is especially effective for building very large dictionaries, is the main advantage of the system presented above.

4.2 Encoders

The choice of the feature “encoding” function $f(x;\mathcal{D})$ also has a major impact on recognition performance. For instance, consider the “hard assignment” encoding where we take $f(x;\mathcal{D}) = s$, with s the standard “one-hot” code used during K-means training (Eq. (1)). It is well-known that this scheme performs very poorly compared to other choices [18,1]. Thus, once we have run K-means to train our filters, one should certainly make an effort to choose a better encoder f. There are many encoding schemes present in the literature [34,38,40,42,19,20,7] and, while they often include their own training algorithms, one can choose to use K-means-trained dictionaries in conjunction with many of them.

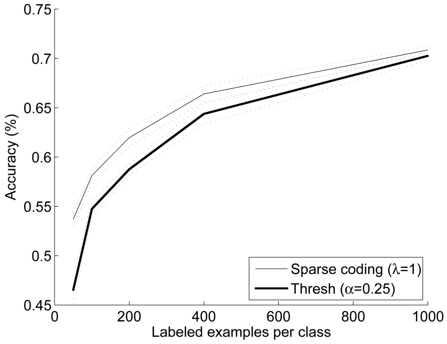


Fig. 6: A comparison of the performance between two encoders on the CIFAR-10 [25] benchmark as a function of the number of labeled examples. When labeled data is scarce, an expensive encoder can be worthwhile. If labeled data is plentiful, a fast, simple encoder such as a soft-threshold nonlinearity is sufficient.

Unfortunately, many encoding schemes proposed for computer vision tasks are potentially very expensive to compute. Many require us to solve a difficult optimization problem in order to compute $f(x; \mathcal{D})$ [34,40,20,7]. On the other hand, some encoders are simple nonlinear functions of the filter responses $D^\top x$ that can be computed extremely quickly. In previous work it has appeared that algorithms like sparse coding are generally the best performers in benchmarks [39,4]. However, in some cases we can manage to get away with much simpler encodings. Specifically, when we use the single-layer architecture outlined above, it turns out that algorithms like sparse coding and more sophisticated variants (e.g., spike-slab sparse coding [20]) are difficult to top when we have very little labeled data. But as can be seen in Figure 6, with much more labeled data the disparity in performance between sparse coding and a very simple nonlinear encoder decreases significantly. We have found, not surprisingly, that as we use increasing quantities of labeled data the supervised learning stage takes over and works equally well with most reasonable encoders.

As a result of this observation, application designers should consider the quantity of available labeled data. If labeled data is abundant, a fast feed-forward encoder works well (and is the easiest to use on large datasets). If labeled data is scarce, however, it can be worthwhile to use a more expensive encoding scheme. In the large-scale case we have found that soft-threshold nonlinearities ($f(x; \mathcal{D}, \alpha) = \max\{0, \mathcal{D}^\top x - \alpha\}$, where α is a tunable constant) work very well.

By contrast, sigmoid nonlinearities (e.g., $f(x; \mathcal{D}, b) = (1 + \exp(-\mathcal{D}^\top x + b))^{-1}$) appear to work significantly less well [33,12] in similar instances.

5 Local Receptive Fields and Multiple Layers

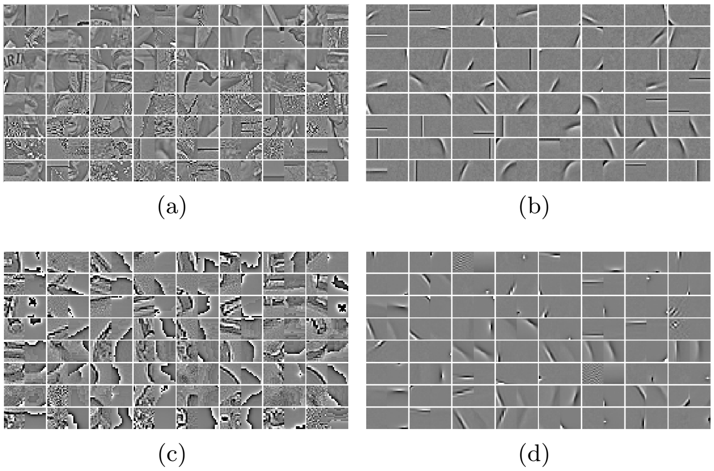
Several times we have referred to the difficulties involved in applying K-means to high-dimensional data. In Section 4.1 it was explained that we should choose the image patch size p ("receptive field" size) carefully to avoid exceeding the modeling capacity of K-means (as in Figure 4b). If we have a very large image it is generally not effective to apply K-means to the entire input at once.[2] Instead, applying K-means to p-by-p pixel sub-patches is a reasonable substitute, since we expect that most learned features will be localized to a small region. This "trick" allows us to keep the input size to K-means relatively small (e.g., just p^2 input dimensions for grayscale patches), but still use the resulting filters on a much larger image by either reusing the filters for every p-by-p pixel sub-patch of the image, or even by re-running K-means independently on each p-by-p region (if, for some reason, the features present in other parts of the image differ significantly). Though this approach is well-known from computer vision applications the same trick works more generally and in some cases is indispensable for building working systems.

Consider a situation where our input is, in fact, a concatenation of two independent signals. Concretely, let us take two image patches drawn at random from a larger dataset and concatenate them side-by-side as in Figure 7a. When we run K-means on this type of data we end up with the centroids in Figure 7b where individual centroids tend to model just one of the two independent components, much like we would expect from ICA. Unfortunately, as observed previously, to achieve this result we actually need more data than if we had tried to model the two patches separately. Hence, whenever we can determine *a priori* that our input variables can be split into independent chunks, we should try to split them up immediately and run K-means on each chunk separately. Note that the contrived example here occurs in real applications, such as learning features from RGB-Depth data [27]: Figure 7c shows examples of image intensity concatenated with depth patches and Figure 7d shows centroids learned from them. Since at this scale depth tends to be only weakly related to raw pixel intensity, it might be better to run K-means separately on each modality of the data.

5.1 Deep Networks

In Section 4 we presented a simple pipeline that enabled us to extract a single layer of features from an image by taking a dictionary learned from small patches and using it to extract features over a larger image (see Figure 5). We then used

[2] Note that for very large inputs it becomes impractical to perform whitening, which requires solving a very large optimization problem (e.g., eigenvalue decomposition). In the authors' experience K-means starts giving poor results before this computational bottleneck is reached.



(a)

(b)

(c)

(d)

Fig. 7: (a) A dataset composed of concatenated pairs of independently sampled image patches. (b) Centroids trained from pairs of image patches. Note that, as expected, K-means learns filters that only involve half of the input at a time. Due to the increased dimensionality, significantly more data is needed compared to training on each image separately. (c) A dataset composed of image-depth pairs. The left half of each example is a whitened grayscale patch. The right half is a "depth image" [27] where pixel intensity represents the distance of a surface from the camera. (d) Centroids learned jointly from image-depth pairs learn only a few weak features that use both modalities for similar reasons as in (b).

a pooling stage to reduce the number of features before applying a supervised learning algorithm. It would be nice if we could learn higher layers of features by taking the resulting single-layer representation and passing it back through our feature learning pipeline. For instance, one simple way we might try to accomplish this is to compute the pooled feature values for all of the examples in our unlabeled dataset X to give us a new dataset Z, then apply the exact same learning pipeline to Z to learn new features. This simple approach has been applied in [1], but it turns out that this straight-forward strategy can hit a major snag: the inputs Z used for the second layer of features will often be very high dimensional if, as is common, we use very large dictionaries of features (e.g., $k = 10000$ or more).

Concretely, let's consider a simple example.[3] Suppose that our goal is to learn features for a 20-by-20 pixel image patch. With the approach of Section 4 we train $k = 10000$ 16-by-16 pixel filters with K-means from 16-by-16 patches

[3] The numbers used in this example are chosen to illustrate the problem and its solution. For a more detailed and realistic setup, see [10].

cropped out of our dataset. We then take the learned dictionary $\mathcal{D}$ and extract feature responses with a step size of 4 pixels $f(x;\mathcal{D})$ from the 20-by-20 pixel images, yielding a 2-by-2-by-10000 image representation. Finally, we sum up the responses over each 2-by-2 region (i.e., all responses produced by each filter) to yield a 1-by-1-by-10000 "pooled representation" which we will take as Z. We can think of each feature value z_j as being a slightly translation-invariant version of the feature detector associated with the filter $\mathcal{D}^{(j)}$. Note that each vector in Z now has 10000 dimensions to represent the original 400-dimensional patch. At this scale, even learning a "complete" representation of 10000 features from the 10000-dimensional inputs Z becomes challenging. Regrettably, there is no obvious choice of local receptive field that can be made here: the 10000 features are unorganized and we have no way to split them up by hand.

One proposed solution to this problem is to use a simple form of pair-wise "dependency test" to help identify groups of dependent input variables in an automated way. If we can do this, then we can break up the input vector coordinates into small groups suitable for input to K-means instead of picking groups by hand. This tool is most valuable for building multiple layers of features with K-means.

As an example, we can use a type of dependency called "energy correlation". Given two whitened inputs (i.e., two inputs z_j and z_k that have no linear correlation) their energy correlation is just the correlation between their squared responses. In particular, if we have $\mathbb{E}\left[z\right] = 0$ and $\mathbb{E}\left[zz^\top\right] = I$, then we will define the dependency between inputs z_j and z_k as:

$$d[z_j, z_k] = \mathrm{corr}(z_j^2, z_k^2) = \mathbb{E}\left[z_j^2 z_k^2 - 1\right] / \sqrt{\mathbb{E}\left[z_j^4 - 1\right] \mathbb{E}\left[z_k^4 - 1\right]}.$$

This metric is easy to compute by first whitening the input data Z with ZCA whitening [3], then computing the pairwise similarities between all of the features:

$$d(j,k;Z) \equiv d[z_j, z_k] \equiv \frac{\sum_i {z_j^{(i)}}^2 {z_k^{(i)}}^2 - 1}{\sqrt{\sum_i ({z_j^{(i)}}^4 - 1) \sum_i ({z_k^{(i)}}^4 - 1)}}.$$

This computation is practical for fewer than 10000 input features. It can still be computed approximately for hundreds of thousands of features if necessary [10]. Thus, we now have a function $d(j,k;Z)$ that can provide a measure of the dependency between features z_j and z_k observed in a given dataset Z.

Now we would like to try to learn some "higher level" features on top of the Z representation. Using our dependency test we can find reasonable choices of receptive fields in a relatively simple way: we pick one feature, say z_0, and then use the dependency test to find the R features with the strongest dependence on z_0 according to the test (i.e., find the indices j so that $d(0,j;Z)$ is large). We then run K-means using only these R features as input. If we pick R small enough (e.g., 100 or 200) the usual normalization, whitening and K-means training steps can be applied easily and require virtually no tuning to work well. Because of the smaller input dimension we only need to train a few hundred centroids and

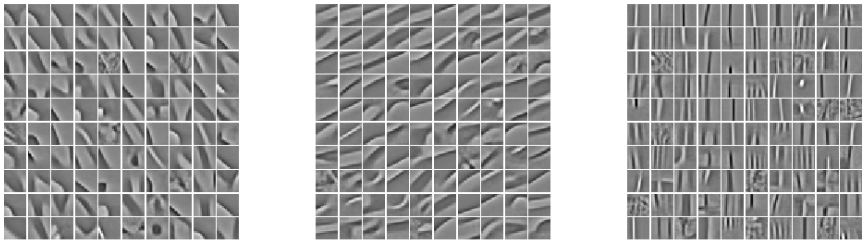


Fig. 8: Groups of features selected by an automated dependency test. The features corresponding to each group of filters would be processed separately by K-means to build a second layer of features.

thus we can use much less data than would be required to run K-means on the original 10000-dimensional dataset. This procedure can be repeated for many choices of the "seed" feature (z_0 above) until we have trained dictionaries from receptive fields covering all of the input variables in z. Figure 8 shows the first-layer filters from $\mathcal{D}$ corresponding to some of these groups of features (i.e., these are the $\mathcal{D}^{(j)}$ whose pooled responses z_j have high dependence according to the energy correlation test).

The effectiveness of this sort of approach combined with K-means has been shown in previous work [10]. Table 1 details results obtained on the full CIFAR dataset with various settings and comparisons to other contemporary methods. First, we can see in these results that learning 2 layers of features is essentially fruitless when using naive choices of receptive fields: a single receptive field that includes all of the inputs, or receptive fields that connect to random inputs. Indeed, the results for 2 layer networks are *worse* (77.4% and 77.6%) than obtained using a single layer alone (78.3%). This should be expected: using a single receptive field, K-means is unable to build good features due to the high-dimensional input (like Figure 4b), yet using a random receptive field wastes representational power modeling unrelated inputs (like Figure 7). By contrast, results obtained with the receptive field learning scheme above (with 2nd-order dependency measure) are significantly better: achieving a significant improvement over the baseline single-layer results, and even out-performing a much larger single-layer network. With 3 layers, this system improves further to 82.0% accuracy. Achieving the best possible results (as reported in [9]) may require supervised training of the entire network, but this result demonstrates very clearly the importance of controlling the *connectivity* of features in order for K-means to work well in deep networks (where we typically use only unlabeled data to construct features).

Training only from unlabeled data is much more useful in a scenario where we have limited labeled training data. Tables 2 and 3 show results obtained from similar experiments on the CIFAR-10 dataset when using only 400 labeled examples per class, and the STL-10 [11] dataset (where only 100 labels are available

Table 1: Results on CIFAR-10 (full)

Architecture	**Accuracy (%)**
1 Layer	78.3%
1 Layer (4800 maps)	80.6%
2 Layers (Single RF)	77.4%
2 Layers (Random RF)	77.6%
2 Layers (Learned RF)	81.2%
3 Layers (Learned RF)	82.0%
VQ (6000 maps) [12]	81.5%
Conv. DBN [26]	78.9%
Deep NN [8]	80.49%
Multi-column Deep NN [9]	**88.79%**

Table 2: Results on CIFAR-10 (400 ex. per class)

Architecture	**Accuracy (%)**
1 Layer	64.6% (±0.8%)
1 Layer (4800 maps)	63.7% (±0.7%)
2 Layers (Single RF)	65.8% (±0.3%)
2 Layers (Random RF)	65.8% (±0.9%)
2 Layers (Learned RF)	69.2% (±0.7%)
3 Layers (Learned RF)	**70.7%** (±0.7%)
Sparse coding (1 layer) [12]	66.4% (±0.8%)
VQ (1 layer) [12]	64.4% (±1.0%)

per class). The results are very similar, even though we have less supervision: poor choices of receptive fields almost entirely negate the benefits of training multiple layers of features, but using the simple receptive field selection technique above allows us to successfully build up to 3 layers of useful features with K-means.

Table 3: Classification Results on STL-10

Architecture	**Accuracy (%)**
1 Layer	54.5% (±0.8%)
1 Layer (4800 maps)	53.8% (±1.6%)
2 Layers (Single RF)	55.0% (±0.8%)
2 Layers (Random RF)	54.4% (±1.2%)
2 Layers (Learned RF)	58.9% (±1.1%)
3 Layers (Learned RF)	**60.1**% (±1.0%)
Sparse coding (1 layer) [12]	59.0% (±0.8%)
VQ (1 layer) [12]	54.9% (±0.4%)

6 Conclusion

In this chapter we have reviewed many results, observations and tricks that are useful for building feature-learning systems with K-means as a scalable unsupervised learning module. The major considerations that we have covered, which practitioners should keep in mind before embarking on a new application, are summarized as:

1. Mean and contrast normalize inputs.

2. Use whitening to "sphere" the data, taking care to set the ϵ parameter appropriately. If whitening cannot be performed due to input dimensionality, one should split up the input variables.
3. Initialize K-means centroids randomly from Gaussian noise and normalize.
4. Use damped updates to help avoid empty clusters and improve stability.
5. Be mindful of the impact of dimensionality and sparsity on K-means. K-means tends to find sparse projections of the input data by seeking out "heavy-tailed" directions. Yet when the data is not properly whitened, the input dimensionality is very high, or there is insufficient data, it may perform poorly.
6. With higher dimensionalities, K-means will require significantly increased amounts of data, possibly negating its speed advantage.
7. Exogenous parameters in the system (pooling, encoding methods, etc.) can have a bigger impact on final performance than the learning algorithm itself. Consider spending compute resources on more cross-validation for parameters before concluding that a more expensive learning scheme is required.
8. Using more centroids almost always helps when using the image recognition pipeline described in this chapter, provided we have enough training data. Indeed, whenever more compute resources become available, this is the first thing to try.
9. When labeled data is abundant, find a cheap encoder and let a supervised learning system do most of the work. If labeled data is limited (e.g., hundreds of examples per class), an expensive encoder may work better.
10. Use local receptive fields wherever possible. Input data dimensionality is the main bottleneck to the success of K-means and should be kept as low as possible. If local receptive fields cannot be chosen by hand, try an automated dependency test to help cut up your data into (overlapping) groups of inputs with lower dimensionality. This is likely a necessity for deep networks!

The above recommendations cover essentially all of the tools, tricks and insights that underlie recent feature-learning results based on K-means. Though it is unclear how far K-means can be pushed in comparison to more expressive algorithms, the tips above are enough to know when K-means is appropriate and to get it working in many challenging scenarios.

References

1. Agarwal, A., Triggs, B.: Hyperfeatures: Multilevel local coding for visual recognition. In: 9th European Conference on Computer Vision. vol. 1, pp. 30–43 (2006)
2. Aharon, M., Elad, M., Bruckstein, A.: K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation. IEEE Transactions on Signal Processing 54(11), 4311–4322 (2006)
3. Bell, A., Sejnowski, T.J.: The 'independent components' of natural scenes are edge filters. Vision Research 37(23), 3327–3338 (1997)
4. Boureau, Y., Bach, F., LeCun, Y., Ponce, J.: Learning mid-level features for recognition. In: 23rd Conference on Computer Vision and Pattern Recognition. pp. 2559–2566 (2010)

5. Boureau, Y., Ponce, J., LeCun, Y.: A theoretical analysis of feature pooling in visual recognition. In: 27th International Conference on Machine Learning. pp. 111–118 (2010)
6. Boureau, Y., Roux, N.L., Bach, F., Ponce, J., LeCun, Y.: Ask the locals: multi-way local pooling for image recognition. In: 13th International Conference on Computer Vision. pp. 2651–2658 (2011)
7. Bradley, D.M., Bagnell, J.A.: Differentiable sparse coding. In: Advances in Neural Information Processing Systems 22. pp. 113–120 (2008)
8. Ciresan, D.C., Meier, U., Masci, J., Gambardella, L.M., Schmidhuber, J.: Flexible, high performance convolutional neural networks for image classification. In: International Joint Conference on Artificial Intelligence. pp. 1237–1242 (2011)
9. Ciresan, D.C., Meier, U., Schmidhuber, J.: Multi-column deep neural networks for image classification. In: Computer Vision and Pattern Recognition. pp. 3642–3649 (2012)
10. Coates, A., Ng, A.Y.: Selecting receptive fields in deep networks. In: Advances in Neural Information Processing Systems 24. pp. 2528–2536 (2011)
11. Coates, A., Lee, H., Ng, A.Y.: An analysis of single-layer networks in unsupervised feature learning. In: 14th International Conference on AI and Statistics. pp. 215–223 (2011)
12. Coates, A., Ng, A.Y.: The importance of encoding versus training with sparse coding and vector quantization. In: 28th International Conference on Machine Learning. pp. 921–928 (2011)
13. Csurka, G., Dance, C., Fan, L., Willamowski, J., Bray, C.: Visual categorization with bags of keypoints. In: ECCV Workshop on Statistical Learning in Computer Vision. pp. 59–74 (2004)
14. Dhillon, I.S., Modha, D.M.: Concept decompositions for large sparse text data using clustering. Machine Learning 42(1), 143–175 (2001)
15. Efron, B., Hastie, T., Johnstone, I., Tibshirani, R.: Least angle regression. The Annals of statistics 32(2), 407–499 (2004)
16. Fei-Fei, L., Perona, P.: A Bayesian hierarchical model for learning natural scene categories. In: Computer Vision and Pattern Recognition. vol. 2, pp. 524–531 (2005)
17. Garrigues, P., Olshausen, B.: Group sparse coding with a laplacian scale mixture prior. In: Advances in Neural Information Processing Systems 23. pp. 676–684 (2010)
18. van Gemert, J.C., Geusebroek, J.M., Veenman, C.J., Smeulders, A.W.M.: Kernel codebooks for scene categorization. In: 10th European Conference on Computer Vision. vol. 5304, pp. 696–709 (2008)
19. Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: 14th International Conference on Artificial Intelligence and Statistics. pp. 315–323 (2011)
20. Goodfellow, I., Courville, A., Bengio, Y.: Spike-and-slab sparse coding for unsupervised feature discovery. In: NIPS Workshop on Deep Learning and Unsupervised Feature Learning (2011)
21. Hinton, G., Osindero, S., Teh, Y.: A fast learning algorithm for deep belief nets. Neural Computation 18(7), 1527–1554 (2006)
22. Hyvärinen, A., Hurri, J., Hoyer, P.: Natural Image Statistics. Springer-Verlag (2009)
23. Hyvärinen, A., Oja, E.: Independent component analysis: algorithms and applications. Neural networks 13(4-5), 411–430 (2000)

24. Jarrett, K., Kavukcuoglu, K., Ranzato, M., LeCun, Y.: What is the best multi-stage architecture for object recognition? In: 12th International Conference on Computer Vision. pp. 2146–2153 (2009)
25. Krizhevsky, A.: Learning multiple layers of features from Tiny Images. Master's thesis, Dept. of Comp. Sci., University of Toronto (2009)
26. Krizhevsky, A.: Convolutional Deep Belief Networks on CIFAR-10. Unpublished manuscript (2010)
27. Lai, K., Bo, L., Ren, X., Fox, D.: A large-scale hierarchical multi-view RGB-D object dataset. In: International Conference on Robotics and Automation. pp. 1817–1824 (2011)
28. Lazebnik, S., Schmid, C., Ponce, J.: Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In: Computer Vision and Pattern Recognition (2006)
29. LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D.: Backpropagation applied to handwritten zip code recognition. Neural Computation 1, 541–551 (1989)
30. LeCun, Y., Huang, F.J., Bottou, L.: Learning methods for generic object recognition with invariance to pose and lighting. In: Computer Vision and Pattern Recognition. vol. 2, pp. 97–104 (2004)
31. Lee, H., Grosse, R., Ranganath, R., Ng, A.Y.: Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In: 26th International Conference on Machine Learning. pp. 609–616 (2009)
32. Mairal, J., Bach, F., Ponce, J., Sapiro, G.: Online learning for matrix factorization and sparse coding. Journal of Machine Learning Research 11, 19–60 (2010)
33. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: 27th International Conference on Machine Learning. pp. 807–814 (2010)
34. Olshausen, B.A., Field, D.J.: Emergence of simple-cell receptive field properties by learning a sparse code for natural images. Nature 381(6583), 607–609 (1996)
35. Raina, R., Battle, A., Lee, H., Packer, B., Ng, A.: Self-taught learning: transfer learning from unlabeled data. In: 24th International Conference on Machine learning. pp. 759–766 (2007)
36. Tibshirani, R.: Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society. Series B (Methodological) pp. 267–288 (1996)
37. Varma, M., Zisserman, A.: A statistical approach to material classification using image patch exemplars. Transactions on Pattern Analysis and Machine Intelligence 31(11), 2032–2047 (November 2009)
38. Wang, J., Yang, J., Yu, K., Lv, F., Huang, T., Gong, Y.: Locality-constrained linear coding for image classification. In: Computer Vision and Pattern Recognition. pp. 3360–3367 (2010)
39. Yang, J., Yu, K., Gong, Y., Huang, T.S.: Linear spatial pyramid matching using sparse coding for image classification. In: Computer Vision and Pattern Recognition. pp. 1794–1801 (2009)
40. Yu, K., Zhang, T., Gong, Y.: Nonlinear learning using local coordinate coding. In: Advances in Neural Information Processing Systems 22. pp. 2223–2231 (2009)
41. Zetzsche, C., Krieger, G., Wegmann, B.: The atoms of vision: Cartesian or polar? Journal of the Optical Society of America 16(7), 1554–1565 (July 1999)
42. Zhou, X., Yu, K., Zhang, T., Huang, T.: Image classification using super-vector coding of local image descriptors. 11th European Conference on Computer Vision pp. 141–154 (2010)