CHAPTER 10

Forming and Using Patch Dictionaries

Chapter 9 established that there are many patches in a particular image that are like one another, and that this property can be exploited to inpaint or denoise. If you wish to reconstruct some patch in a particular image, you might look in other images for examples. The main difficulty you will encounter is finding the patches you want in the pool of available patches.

This chapter describes methods to control that complexity. Imagine you have a very large pool of patches, taken from a large number of images. There will be numerous near duplicates (which is why inpainting and non-local means worked). It is natural to try to cluster the near duplicates together. You could now work with the clusters rather than the patches. Alternatively, you could use the clusters to simplify finding a matching patch.

The key engine here is *clustering.* Clustering is a procedure that takes individual data items – in this case, patches, but others will appear – and produces blobs or *clusters* consisting of many similar data items. Each cluster has some representation in terms of parameters. You must determine (a) how many clusters there are; (b) what the cluster parameters are; and (c) which data items belong to which cluster.

10.1 CLUSTERING AND K-MEANS

10.1.1 Core Ideas

In **agglomerative clustering**, you start with each data item being a cluster, and then merge clusters recursively to yield a good clustering. Agglomerative clustering needs a good inter-cluster distance to fuse nearby clusters. There are three recipes for inter cluster distances. The distance between the closest elements tends to yield extended clusters and is known to statisticians as *single-link clustering*). The maximum distance between an element of the first cluster and one of the second tends to yield rounded clusters and is known to statisticians as *complete-link clustering.* The average of distances between elements in the cluster also tends to yield rounded clusters and is known to statisticians as *group average clustering.*

In **divisive clustering**, you start with the entire data set being a cluster, and then split clusters recursively to yield a good clustering Divisive clustering needs some criterion for splitting clusters. Mostly, this recipe is not much used in vision applications (though you will see echoes of the idea below) and I will not pursue splitting criteria.

Either algorithm needs to know when to stop. This can be difficult if there is no model for the process that generated the clusters. One strategy, popular in interactive data visualization and sometimes useful in other circumstances, is to

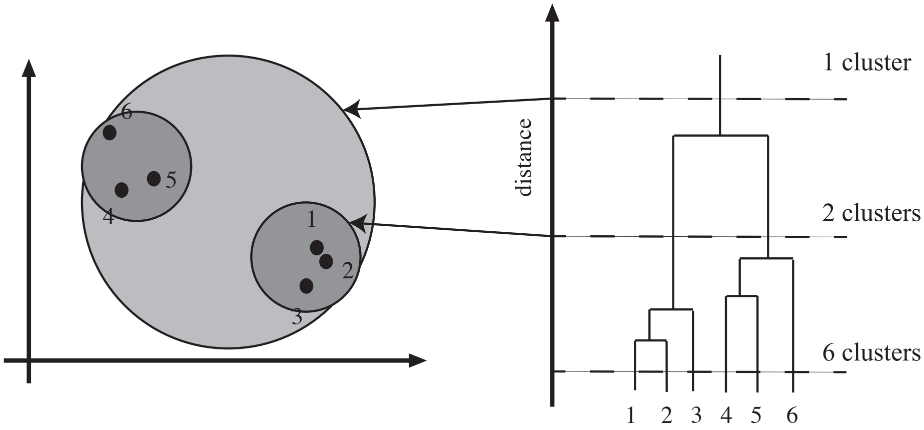

FIGURE 10.1: **Left**, *a data set;* **right**, *a dendrogram obtained by agglomerative clustering using single-link clustering. If one selects a particular value of distance, then a horizontal line at that distance splits the dendrogram into clusters. This representation makes it possible to guess how many clusters there are and to get some insight into how good the clusters are.*

recognize the recipes generate a hierarchy of clusters. You could simply generate the whole hierachy, then navigate it in some application appropriate way. For visualization, this hierarchy can be displayed in the form of a *dendrogram*—a representation of the structure of the hierarchy of clusters that displays inter-cluster distances—and an appropriate choice of clusters is made from the dendrogram. Dendrograms are difficult to intepret when there are many data items, but can be a helpful guide in simple cases (Figure 10.2).

Another important thing to notice about clustering from the example of figure **??** is that there is no right answer. There are a variety of different clusterings of the same data. For example, depending on what scales in that figure mean, it might be right to zoom out and regard all of the data as a single cluster, or to zoom in and regard each data point as a cluster. Each of these representations may be useful. This is a general feature of clustering as a problem. It is hardly ever helpful to talk about whether a clustering is right, but it is often important to focus on how useful it is.

10.1.2 The K-Means Algorithm

Write $\mathbf{x}_i$ for a set of N image patches, straightened into vectors for convenience. Assume you know that there are k clusters. Write $\mathbf{c}_j$ for the center of the jth cluster. Write $\delta_{i,j}$ for a discrete variable that records which cluster a data item belongs to, so

$$\delta_{i,j} = \begin{cases} 1 & \text{if } \mathbf{x}_i \text{ belongs to cluster } j \\ 0 & \text{otherwise} \end{cases}$$

Every data item belongs to exactly one cluster, so $\sum_j \delta_{i,j} = 1$. Every cluster must contain at least one point, so $\sum_i \delta_{i,j} > 0$ for every j. The sum of squared distances from data points to cluster centers is then

$$\Phi(\delta, \mathbf{c}) = \sum_{i,j} \delta_{i,j} \left[(\mathbf{x}_i - \mathbf{c}_j)^T (\mathbf{x}_i - \mathbf{c}_j) \right].$$

Notice how the $\delta_{i,j}$ are acting as "switches". For the i'th data point, there is only one non-zero $\delta_{i,j}$ which selects the distance from that data point to the appropriate cluster center.

You could cluster the data by choosing the δ and $\mathbf{c}$ that minimizes $\Phi(\delta, \mathbf{c})$. This would yield the set of k clusters and their cluster centers such that the sum of distances from points to their cluster centers is minimized. There is no known algorithm that can minimize Φ exactly in reasonable time. The $\delta_{i,j}$ are the problem: it turns out to be hard to choose the best allocation of points to clusters.

There is a remarkably effective approximate solution. Notice that if the $\mathbf{c}$'s are known, getting the δ's is easy – for the i'th data point, set the $\delta_{i,j}$ corresponding to the closest $\mathbf{c}_j$ to one and the others to zero. Similarly, if the $\delta_{i,j}$ are known, it is easy to compute the best center for each cluster – just average the points in the cluster. So iterate:

- Assume the cluster centers are known and allocate each point to the closest cluster center.
- Replace each center with the mean of the points allocated to that cluster.

Choose a start point by randomly choosing cluster centers (there are better options, below), and then iterate these stages alternately. This process eventually converges (**exercises**). It is not guaranteed to converge to the global minimum of the objective function, however. In this form, it is also not guaranteed to produce k clusters, but this is easily fixed, below. This algorithm is usually referred to as *k-means*

10.1.3 Initializing K-means

One natural strategy for initializing k-means is to choose k data items at random, then use each as an initial cluster center. This approach is widely used, but has some difficulties. The quality of the clustering can depend quite a lot on initialization, and an unlucky choice of initial points might result in a poor clustering. One (again quite widely adopted) strategy for managing this is to initialize several times, and choose the clustering that performs best in your application. Another strategy, which has quite good theoretical properties and a good reputation, is known as *k-means++*. You choose a point $\mathbf{x}$ uniformly and at random from the dataset to be the first cluster center. Then you compute the squared distance between that point and each other point; write $d_i^2(\mathbf{x})$ for the distance from the i'th point to the first center. You now choose the other $k-1$ cluster centers as IID draws from the probability distribution

$$\frac{d_i^2(\mathbf{x})}{\sum_u d_u^2(\mathbf{x})}.$$

10.1.4 How to choose K

Usually, you don't know how many clusters there should be, and need to choose this by experiment. The obvious strategy – cluster for a variety of different values of k, then look at the value of the cost function for each – does not work. If there are more centers, each data point can find a center that is closer to it, so the value should go down as k goes up. The best k is then the number of data points, which is not helpful.

In some special cases, you might know a label associated with each data point. In such cases, one can evaluate the clustering by looking at the number of different labels in a cluster (sometimes called the purity), and the number of clusters. A good solution will have few clusters, all of which have high purity. Mostly, you won't have a label to check purity.

The alternative strategy, which might seem crude to you, is extremely important in practice. Usually, one clusters data to use the clusters in an application (one of the most important, vector quantization, is described in Section **??**). There are usually natural ways to evaluate this application. For example, vector quantization is often used as an early step in texture recognition or in image matching; here one can evaluate the error rate of the recognizer, or the accuracy of the image matcher. One then chooses the k that gets the best evaluation score on validation data. In this view, the issue is not how good the clustering is; it's how well the system that uses the clustering works.

10.1.5 Scattered Points and Junk Clusters

If you experiment with k-means, you will notice one irritating habit of the algorithm. It almost always produces either some rather spread out clusters, or some single element clusters. Most clusters are usually rather tight and blobby clusters, but there is usually one or more bad cluster. This is fairly easily explained. Because every data point must belong to some cluster, data points that are far from all others (a) belong to some cluster and (b) very likely "drag" the cluster center into a poor location.

There are ways to deal with this. If k is very big, the problem is often not significant, because then you simply have many single element clusters that you can ignore. It isn't always a good idea to have too large a k, because then some larger clusters might break up. An alternative is to have a junk cluster. Any point that is too far from the closest true cluster center is assigned to the junk cluster, and the center of the junk cluster is not estimated. Notice that points should not be assigned to the junk cluster permanently; they should be able to move in and out of the junk cluster as the cluster centers move.

10.1.6 Efficient Clustering and Hierarchical K-Means

One important difficulty occurs in applications. We might need to have an enormous dataset (billions of items is a real possibility), and so a very large k. In this case, k-means clustering becomes difficult because identifying which cluster center is closest to a particular data point scales linearly with k (and we have to do this for every data point at every iteration). There are two useful strategies for dealing with this

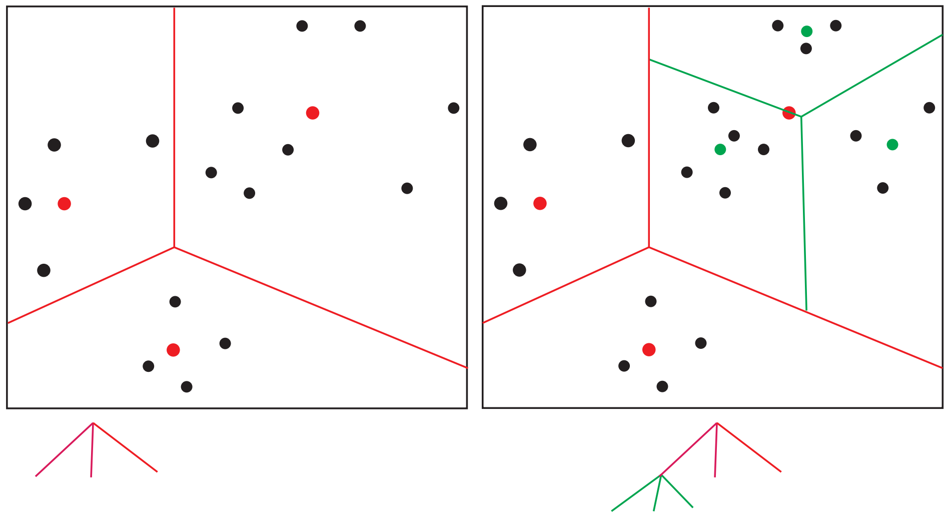

FIGURE 10.2: *Hierachical k-means builds a tree of cells in space. On the* **left**, *a set of data points (***black***) in 2D, with three cluster centers in* **red**. *This clustering divides the plane into three cells corresponding to the cluster centers. The cells contain the points that are closest to their cluster center. On the* **right**, *the points in the top right cell are clustered again into three clusters (centers in* **green***), so that cell is subdivided into three cells. Below each 2D set of points is the relevant abstract tree.*

problem.

The first is to notice that, if we can be reasonably confident that each cluster contains many data points, some of the data is redundant. We could randomly subsample the data, cluster that, then keep the cluster centers. This helps rather a lot, but not enough if you expect the data will contain many clusters.

A more effective strategy is to build a hierarchy of k-means clusters. We randomly subsample the data (typically quite aggressively), then cluster this with a small value of k. Each data item is then allocated to the closest cluster center, and the data in each cluster is clustered again with k-means. We now have something that looks like a two-level tree of clusters. The leaves of this tree contain subsets of the data. All intermediate nodes contain a set of cluster centers, and there is a child associated with each cluster center. Of course, applying this process recursively will produce a more interesting tree of clusters (**exercises**).

You should think of this tree as dividing the space patches into cells. It is straightforward to draw the tree. The root consists of a set of cluster centers, each of which is associated with a child node. Datapoints belong to the child node associated with the closest cluster center, meaning that the root divides the space into cells (formally, Voronoi cells). This applies recursively to the children, which each divide their cell into subcells, and so on (Figure **??**)

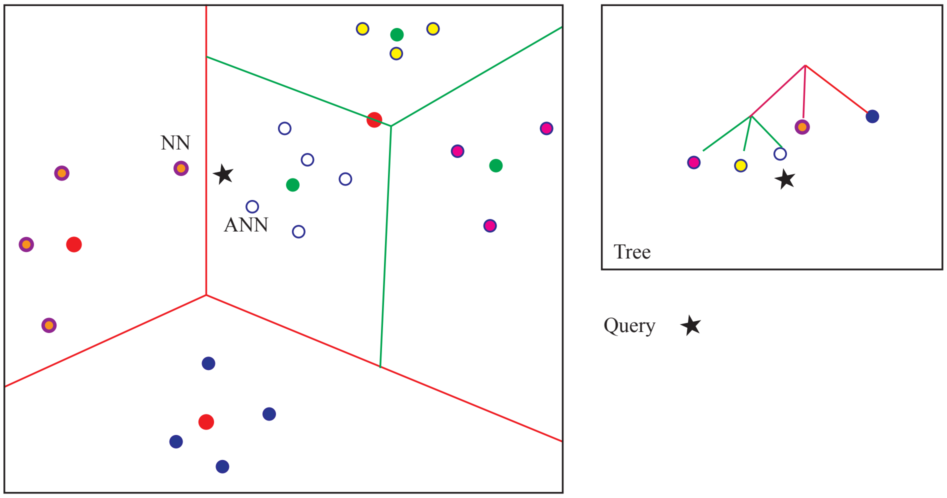

FIGURE 10.3: *Finding an approximate nearest neighbor using hierarchical k-means is easy, but finding the true nearest neighbor is much harder than it looks. On the* **left**, *a set of data points in 2D. The* **red lines** *delineate the regions of the plane that are closest to the three cluster centers at the root of the tree (***red** *points). Two of these regions are leaves of the tree. The third is subdivided (***green lines** *delineate the different cells of points that are closest to the cluster centers which are shown as* **green** *points). On the* **right**, *the tree drawn as an abstract tree. Each data point has been colored with which leaf it belongs to for the abstract tree and the plane drawing. This tree is queried with the point drawn as a star. Notice that the nearest neighbor is not in the cell that the query point is in (it's on the other side of a red line). In fact, to find the nearest neighbor, you would need to go up to the root of the tree and then down to the correct leaf.*

10.2 DENOISING AND REPRESENTING IMAGES WITH PATCH DICTIONARIES

10.2.1 Hierarchical K Means and Approximate Nearest Neighbors

There is really no reason to believe the best match to a patch you want to reconstruct is in the image you are reconstructing. There is often a very good one, as Chapter 9 showed, and this can be convenient because there aren't that many patches in the image. Instead of looking in the image for patches, you could build a very large dictionary of patches taken from some collection of reference images, then reconstruct using that. The key questions here are how to control the complexity of the dictionary, and how to find the closest patch to your patch.

In principle, finding the closest patch is an instance of a standard problem, that is, high dimensional nearest neighbors – find the point in a very large collection of points that matches a query point, where all points are high dimensional vectors. Unfortunately, this problem is extremely nasty, and computational geometers have found it very hard to get a meaningful complexity improvement over the obvious procedure of measuring the distance between each point and the query, then taking

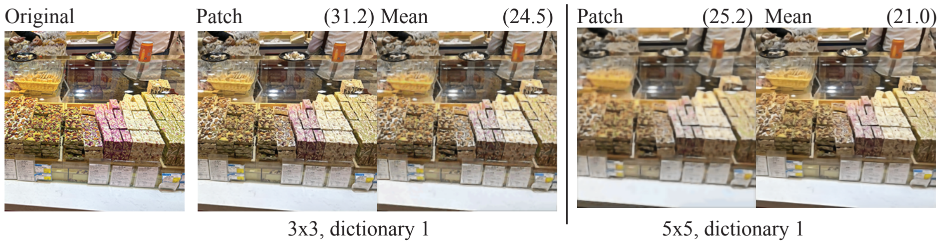

FIGURE 10.4: *Patch based reconstructions using patches from large image dictionaries can be very successful. On the* **left**, *the original image. Others show reconstructions using the approximate nearest neighbor (***Patch***) or the mean of the query cell (***Mean***) using patches of the size indicated. For these reconstructions, the patches overlap and are averaged (the stride is always 1). The PSNR of the reconstruction is shown in parentheses for reference. In each case, the tree contained 1e7 patches, taken from approximately 200, 000 images in the ImageNet test dataset (Section* **??***), and contains no patches from this image.* Image credit: *Figure shows my photograph of a sweetshop in Beijing.*

the smallest. But the obvious procedure is unmanageably slow for our purposes.

It turns out that relaxing the problem slightly offers important complexity improvements. Instead of finding the nearest neighbor, find a point that is very often about as close to the query point as the nearest neighbor is (the exact meaning of "very often" and "about as close" need not concern us, but the details have been worked out). The relaxed problem is often known as *approximate nearest neighbors.*

Here is a procedure to find approximate nearest neighbors. Build a tree out of the dictionary of patches using hierarchical k-means. Each leaf of the tree contains a set of data points that are close to one another, by construction. Find the closest point to a query point by passing the query point down the tree. Do this by recursively choosing the child whose cluster center is closest to the query point until you hit a leaf. Now choose the data item in the leaf that is closest to the query point.

This procedure does not yield the nearest neighbor, even in 2D (Figure **??**), but it has a strong chance of yielding a patch that is as close as the nearest neighbor. You may think it easy to obtain the nearest neighbor by some form of tree walk: it is not. It is straightforward to come up with examples where the tree is a great deal more complicated than the one shown in this figure, and the walk required to find the nearest neighbor is larger. The 2D example suggests that each cell in the tree has few neighboring cells, so you could just check every neighbor of the cell in which the query lands. This isn't practical: the number of neighboring cells will grow exponentially with dimension (**exercises**). With that said, the procedure I have described is extremely useful and widely reliable in practice (versions in Sections **??**, **??** and **??**).

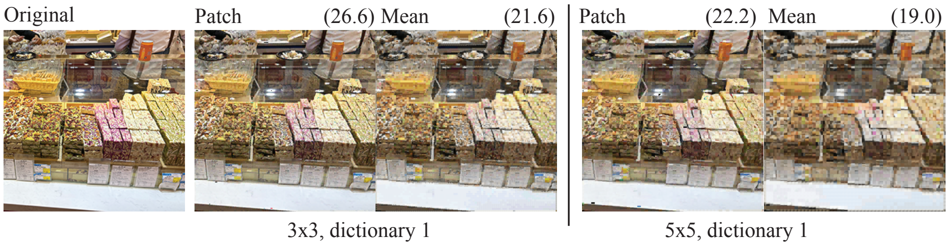

FIGURE 10.5: *Choosing whether patches overlap or not has significant effects. On the* **left**, *the original image. Others show reconstructions using the approximate nearest neighbor (***Patch***) or the mean of the query cell (***Mean***) using patches of the size indicated. For these reconstructions, there is no overlap of patches, which butt against one another (so the stride for* 3×3 *patches is* 3, *and for* 5×5 *patches is* 5*). Notice the blocky structure in the reconstructed images, and the decline in PSNR compared to Figure 10.4. In each case, the tree contained 1e7 patches, taken from approximately 200, 000 images in the ImageNet test dataset (Section* **??***), and contains no patches from this image.* Image credit: *Figure shows my photograph of a sweetshop in Beijing.*

10.2.2 Simple Reconstruction and Denoising with Patches

Once you have a big dictionary which has a tree structure so you can find an approximate nearest neighbor, reconstructing an image is relatively straightforward: turn the image into patches, replace each patch in the image with an approximate nearest neighbor, then turn the patches into an image. There are two questions of detail: precisely which patch you recover from the tree and whether the patches overlap or not.

Recovering a patch from the tree: You could simply do what Section 10.2.1 describes. As Figure 10.4 shows, a large patch dictionary (in this case 1e7 patches from about 2e5 images) gives a very good reconstruction of an image that wasn't used to build the dictionary – image patches are shared across images. This might not be the best estimate of the matching patch. The dictionary is built out of images that may themselves be noisy. The leaves of the tree used for matching each contain a set of patches which should be quite similar to one another (**exercises**). These might be noisy, and so you could to suppress this noise by averaging this set. In this case, each leaf of the tree contains one patch – the average of the data – and any query that arrives at the leaf gets this patch as a response. As Figure 10.4 shows, this strategy results in a reconstruction that isn't as good as the one an approximate nearest neighbor provides, but is quite good.

Overlapping patches: decomposing images into patches is straightforward (most APIs will do this for you without drama), but you do need to think about whether the patches overlap and by how much. If the patches do not overlap, then recovering the image from the patches is easy (you just tile them) but there might be nasty visible boundaries where the tiles butt against each other. A grid like this is usually very easy to spot, and users spot it easily (Figure 10.5 shows examples of the effect). If the patches do overlap, reconstructing the image from the patches

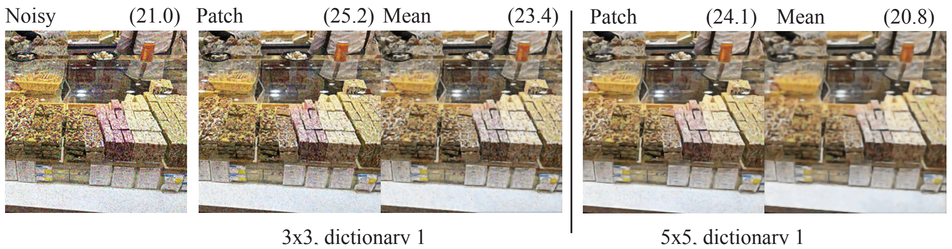

FIGURE 10.6: *Approximate nearest neighbor matching yields quite good denoising. On the* **left**, *a noisy version of the image of Figure 10.4. Others show reconstructions using the approximate nearest neighbor (***Patch***) or the mean of the query cell (***Mean***) using patches of the size indicated. For these reconstructions, the patches overlap and are averaged (the stride is always* 1*). In each case, the tree contained 1e7 patches, taken from approximately 200, 000 images in the ImageNet test dataset (Section* **??***), and contains no patches from this image.* Image credit: *Figure shows my photograph of a sweetshop in Beijing.*

requires some thought because you will have multiple matched patch values at every pixel location and they will likely not agree. The usual strategy is simply to average them, which can cause some loss of detail (this is the strategy of Figures 10.4).

As Figure 10.4 shows, a large enough dictionary will represent a new image well. In turn, you can expect replacing an image patch with its approximate nearest neighbor should denoise the image well, at least for some kinds of noise. Assume the noise maps a patch to one that is "near" the original (as, for example, Gaussian noiatese will). Then the noisy patch should usually be in the same cell as the original patch (check Figure **??** if you're uncertain; **exercises**). Now think about the approximate nearest neighbor to the noisy patch. Quite often, this should be the same as the approximate nearest neighbor of the original patch. Even if it is not, it should seldom be far away, because all the data points in a cell are quite close. Replacing each patch with its approximate nearest neighbor denoises an image rather well, as Figure **??** shows.

All of these statements depend on having a good, comprehensive dictionary of patches. Ideally, the dictionary should hold something close to any actual image patch you encounter. The usual strategy is to collect a large number of diverse images (or, as I have done, use someone else's published collection). You should not extract too many patches from each image in the collection, because these patches are likely somewhat correlated (otherwise the methods of Chapter 9 would not work). Whether an image collection is big enough depends on the size of the patch you are working with (Figure 10.7).

10.3 VECTOR QUANTIZATION

Replacing all training data that arrives at a leaf with the mean of that data doesn't affect reconstructions much, as the figures show. This is the key observation underlying a procedure known as *vector quantization*.

Vector quantization encodes continuous vectors of fixed dimension (like patches)

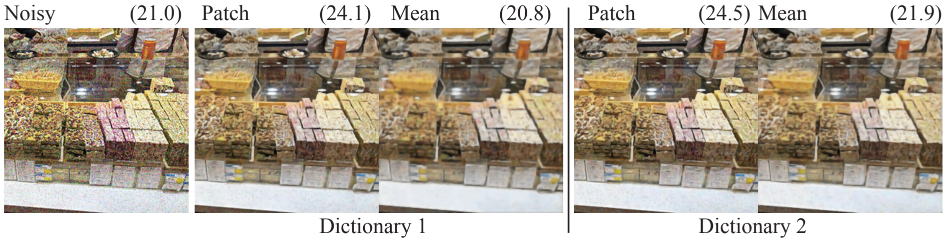

FIGURE 10.7: *The number of patches in the tree has important effects on the reconstruction. On the* **left**, *a noisy version of the image of Figure 10.4. Others show reconstructions using the approximate nearest neighbor (***Patch***) or the mean of the query cell (***Mean***) using* 5×5 *patches. For these reconstructions, the patches overlap and are averaged (the stride is always* 1*). For dictionary 1, the tree contained 1e7 patches taken from approximately 200, 000 images in the ImageNet test dataset (Section* **??***) and had about 2000 leaves. For dictionary 2, the tree contained 5e7 patches taken from approximately 1.2e6 images. Neither dictionary contains patches from this image.* Image credit: *Figure shows my photograph of a sweetshop in Beijing.*

as short codes. Find a large number of examples and cluster them using whatever procedure appeals. At each cluster center, place a unique code. Now encode a new vector by (a) finding the closest cluster center and then (b) reporting that cluster center's code. Optionally, keep a table linking codes to cluster centers, so that you can reconstruct something close to the signal from the code.

Mostly, I've already done an example. When you replace training data that arrives at a leaf with the mean of that data, you are reconstructing cluster centers – the cluster is the set of data items in the leaf – from a code for that leaf. The example shows that this reconstruction is quite accurate, but it doesn't show how useful the procedure is.

10.3.1 Vector Quantization for Noise Suppression

Vector quantization can suppress noise rather well (Figure 10.6). The cluster center closest to the query patch from the noisy image is usually closer to the true underlying patch than the query patch is. When this happens and you replace the query patch with the cluster center you get a reconstruction that is closer to the true image than the original noisy image was. On occasion, the cluster center closest to the query patch is further from the true patch than the query, and so the reconstruction produces occasional large errors. Averaging overlapping patches will tend to reduce the effects of these errors. Chapter **??** gives an extremely important application of this simple idea.

10.3.2 Vector Quantization to Build Features

Typical of signals like sound, images, video, and so on is that different versions of the same thing have different sizes. Instances of the same sounds can last for different times; pictures appear at different resolutions; videos can appear at different resolutions in space and time. This is inconvenient, because the linear classifier of Section 17.1.1 requires a vector of fixed dimension. Chapter **??** showed a very important modern construction for images.

Here is a somewhat older construction that remains useful because ti can be so widely applied. Choose a patch size and some appropriate number of cluster centers. Represent the signal by cutting it into patches of fixed size, and turn these into vectors. Cluster a large set of training vectors. Now use the resulting set of cluster centers to represent a test signal by first vector quantizing each patch in the test signal, then building a histogram of the cluster centers. This histogram has fixed size (the number of cluster centers), and so can be used in a linear classifier.

Although the encoding of individual patches could be quite good, the histogram has little or no information about how the pieces of signal are arranged. This depends to some extent on the degree to which patches overlap. So, for example, the representation can tell whether an image has stripy or spotty patches in it, but is unlikely to know where those patches lie with respect to one another unless they are very large and overlap substantially. This is a much smaller problem than your intuition might tell you – with some care as to details, this recipe yields fast and moderately accurate image classifiers. There is a surprisingly simple construction that improves such a classifier. Build three (or more) dictionaries, rather than one, using different sets of training patches. For example, you could cut the same signals into pieces on a different grid. Now use each dictionary to produce a histogram of cluster centers, and classify with those. Finally, use a voting scheme to decide the class of each test signal. In many problems, this approach yields small but useful improvements.

This construction is now largely obsolete for image classification, but it illustrates an important principle. For at least some kinds of classification task, the details of spatial layout don't matter all that much. Just looking at the overall composition of the image (i.e. "stripey" vs "spotty") might get you where you want to be.