CHAPTER 10

Using Patches to Inpaint and Denoise

This chapter introduces a new and very important property of images: similar patches tend to appear rather often in an image. This property is extremely powerful, because it means that if a patch at some location is degraded, there is very likely another version of that patch at another location that is not. In turn, you can look far away from the pixel you are denoising to find useful information about what it looks like. You could even look in other images, as long as you can control the computational problems involved in finding the right patch.

Figure 10.1 illustrates this essential property. I selected five locations in an image at random. I took the 5×5 patches centered at each location, then found the top 20 matching patches in the image. The best matches are very good. Some very good matches are to patches that are far away from the original location. It should be clear that the size of the patch (very often, referred to as the *scale*) you try to match has strong effects on this property. Experience of images will tell you that, if the patch is 1×1, there will be a large number of matches (which is slightly surprising **exercises**). If the patch is very large, there must be few matches. But patches of moderate scale find many matches. Figure **??** shows matches varying by scale.

10.1 INPAINTING MISSING PIXELS BY MATCHING

10.1.1 Replacing Knocked-out Pixels

Imagine you have an image where some pixels have been set to zero (knocked out) but all others are reliable. The problem of dealing with knocked out pixels is known as *inpainting*. Because real pixels are never zero, you know which pixels are noise. For the moment, assume that the noise pixels are scattered and are selected randomly (as long as they are well separated, the details do not really matter), and are moderately rare. Denoising this image requires estimating the true value of the knocked out pixels. You should immediately think of applying a median filter (Section **??**), which certainly applies to this example.

Here is an alternative strategy. Look at the patch around a noise pixel (the target patch). Images are quite repetitive in structure, meaning that there is likely another patch in the image that matches this one. Now find a pool of patches that match the target well enough. The center pixel in each of these patches is a good estimate of the value of the knocked out pixel. The knocked out pixel can then be replaced either by summarizing these center pixels using a mean, or choosing randomly among them. Alternatively, you could take the center pixel from the best matching patch.

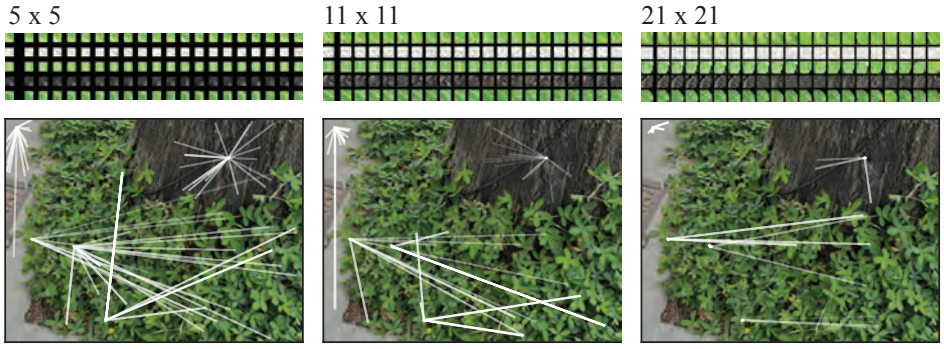


FIGURE 10.1: *Images are made up of patches chosen from quite a small vocabulary, and so any one patch in an image tends to match a number of other patches quite well.* **Top row:** *shows five image patches in detail, selected from the image below at random, and their matching patches for three different patch sizes. The* **left** *column in each shows the patch, and the* **other columns** *show 20 matching patches, found in the image, in order of SSD distance (smallest and so best matching to the left). Notice that there are many patches that are very like a given patch.* **Bottom row** *shows the matches on top of the image. The center of each of the fans of line segments is the query patch (letters key the patch to the details), and each line segment joins a patch to a matching patch. The lines are brighter for small distances, and fainter for large distances. A thicker line occurs when two or more matching patches are close to one another. Notice how matching patches can be quite far away from the query patch (long lines) and how some patches repeat often (many bright lines) whereas others have few matches (many faint lines). Notice also that smaller patches have more widely distributed matches (broad fan, few thick lines) and larger patches tend to have matches that are close together.* Image credit: *Figure shows my photograph of vegetation in Sao Paulo.*

The details are straightforward. When you compute the goodness of the match, some pixels should not contribute. At least the center pixel of the target patch can't (because you don't know its value) and there might be other known noise pixels either in the target patch or in the image patch to which you are matching. This means you need to use a masked normalized SSD, using a mask that is zero at every pixel you don't know.



FIGURE 10.2: *Inpainting occasional missing pixels by matching patches is very successful.* **Top row** *shows images;* **bottom row** *shows detail. For reference, the original image is on the* **left***; just under* 1% *of the pixels in this image have been set to zero (locations chosen uniformly at random) to produce the* **center** *image; the image on the right has been reconstructed by finding the closest* 5×5 *patch that matches the patch surrounding the knocked out pixel (but doesn't have a knocked out pixel in it), then replacing the knocked out pixel with the center of the patch. Look for problems by finding a black pixel in the center detail image that is replaced by an implausible pixel in the right image (a fruitless search!).* Image credit: *Figure shows my photograph of vegetation in Sao Paulo.*

Definition: 10.1 *The masked normalized SSD*

The *masked normalized SSD* scores the similarity between two images $\mathcal{U}$ and $\mathcal{V}$ of the same size ($N \times M$ pixels). You are given a mask $\mathcal{M}$ which is one for pixels that are known and zero for pixels that are not known. Compute

$$\text{mnSSD}(\mathcal{U}, \mathcal{V}) = \left(\frac{1}{\sum \mathcal{M}}\right) \sum \mathcal{M}_{ij} \left(\mathcal{R}_{ij} - \mathcal{B}_{ij}\right)^2.$$

There should always be at least one matching patch, otherwise you can't obtain the value of the missing pixel. This means the pool of matches that are good enough should always contain the best match. Build this pool from the best match, together with the top k matches that are better than some threshold. Figure 10.2



FIGURE 10.3: *Inpainting still works when a lot of pixels have been knocked out, as long as you are careful about how you match.* **Top row** *shows images;* **bottom row** *shows detail. For reference, the original image is on the* **left***; just under* 6% *of the pixels in this image have been set to zero. The locations were chosen uniformly at random, and pixels were knocked out in* 3×3 *blocks to produce the* **center** *image; the image on the right has been reconstructed by finding the closest* 7×7 *patch that matches the patch surrounding the knocked out block (but doesn't have a knocked out pixel in it), then replacing the knocked out block with the center of the patch. Look for problems by finding a black block in the center detail image that is replaced by an implausible pixel in the right image (a fruitless search!).* Image credit: *Figure shows my photograph of vegetation in Sao Paulo.*

shows an example.

This procedure works for blocks of pixels that have been knocked out as well, with minimal changes. Figure 10.4 shows an example where 3×3 blocks of pixels have been knocked out.

10.1.2 Incremental Inpainting

Now imagine that the process that knocks out pixels doesn't just choose pixels at random, but has some some kind of spatial structure. For example, you might have an image with writing on it, and want to replace the writing. Alternatively, the image might have one more more large holes in it.

The pixel inpainting procedure above will work, but some details need to change. When isolated pixels are knocked out, you expect that the patch around the pixel is known. If the image has a large hole in it, this no longer applies. Fixing a pixel requires you have at least some known pixel values close to it. Choose such



FIGURE 10.4: *Incremental inpainting can fill in large holes. On the* **top left**, *an image with a large hole in it;* **top right** *shows the inpainted image, using* 11×11 *patches and a radial order. Alternatives appear in Figure* **??**. Image credit: *Figure shows my photograph of vegetation in Sao Paulo.*

a pixel, and match the patch using the known pixels only. You can do this with a mask that zeros the contribution of knocked out pixels to the SSD. This produces a pool of matches. Now estimate the value of the pixel using this pool. For the moment, choose the center of the best match. Place this value in the image, and you now have an image with a slightly smaller hole in it, so you should be able to find more candidate pixels for replacing. In this *incremental reconstruction* approach, the order in which you visit pixels and the size of the patch becomes important and can quite strongly affect the result.

As Figure 10.4 shows, really quite large holes in images can be fixed quite satisfactorily in this way. The scale of the patch and the order of inpainting matters a lot (Figure 10.5). Top down, left to right order means the pixels are ordered by vertical coordinate, then by horizontal coordinate. This tends to produce somewhat disordered inpaintings, because some pixels with all-inpainted neighbors are inpainted before others with known neighbors. Radial order means that pixels on the edge of the blob are filled in first, so pixels with more known neighbors are inpainted first. This tends to preserve structure. Notice also the effect of the scale of the patch. Matching larger patches tends to reproduce more long-scale structure, as you should expect. For example, the regions inpainted with larger patches appear to have leaves and stems in them.

10.1.3 Texture Synthesis by Incremental Reconstruction

Imagine you have a small texture image you would like to make larger, where the larger image should have the same texture as the original part. There are a variety of application reasons to do this. For example, you might want to apply a texture to a computer graphics model. Just tiling the texture won't work. The patches may not join up properly, and even if they do the periodic structure that results looks bad (**exercises**). Think of the problem as a rather odd inpainting problem – rather than knocking out a block of pixels, the noise process has obscured pixels outside the image.

It is straightforward to extend the incremental inpainting procedure to make

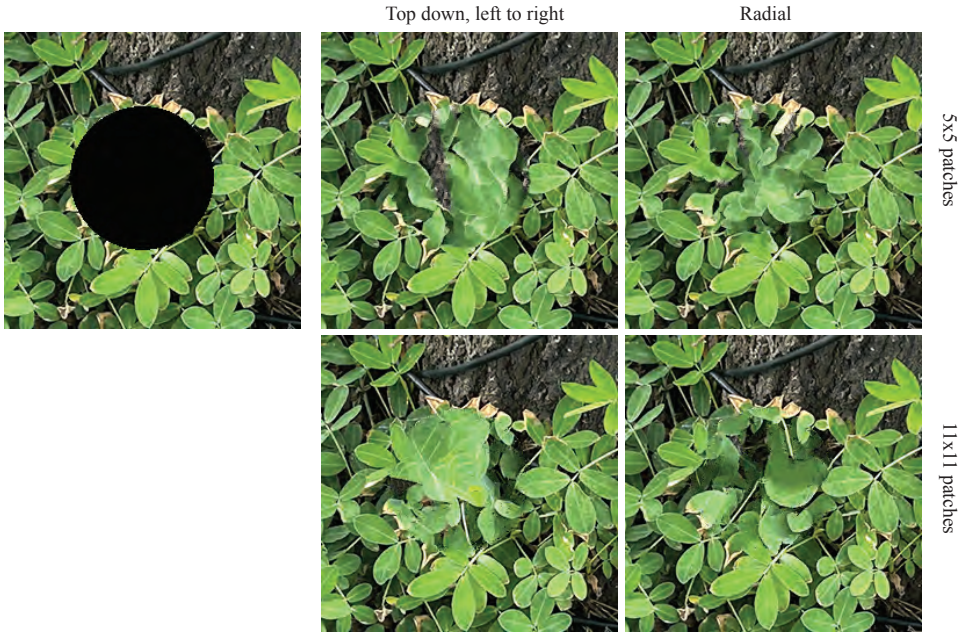


FIGURE 10.5: *The size of the patch you use in matching and the order in which you inpaint pixels have significant effects on the results of incremental inpainting. This figure shows a detail of Figure 10.4, with the blob inpainted in four different ways: patch size differs from row to row, order from column to column. Details in main text.* Image credit: *Figure shows my photograph of vegetation in Sao Paulo.*

a larger texture image from a small one. Find a pixel location whose value isn't known, but where many neighbors have known values. Find matching patches in the known parts of the image, where you compute the SSD using only the known pixel values. Now choose the center pixel value from the pool of matching patches at random (rather than using the best match). The value of this pixel is now known, and you can iterate. The result is a synthesized texture image. Figure 10.7 shows an example.

It is quite important to have several patches to choose the pixel value from, and to choose at random. This prevents the texture you synthesize from being overly repetitious or even constant. The size of the patch you use also has important effects (Figure **??**)

10.2 DENOISING WITH PATCHES

Inpainting relies on a somewhat specialized model of noise. A small fraction of pixels need fixing, you which ones they are, and other pixel values are reliable. If the noise is, say, additive gaussian noise, these constraints don't apply. In turn, this means that all the pixels surrounding the pixel you want to fix may be somewhat wrong as well, so matching using their values may not be wise.

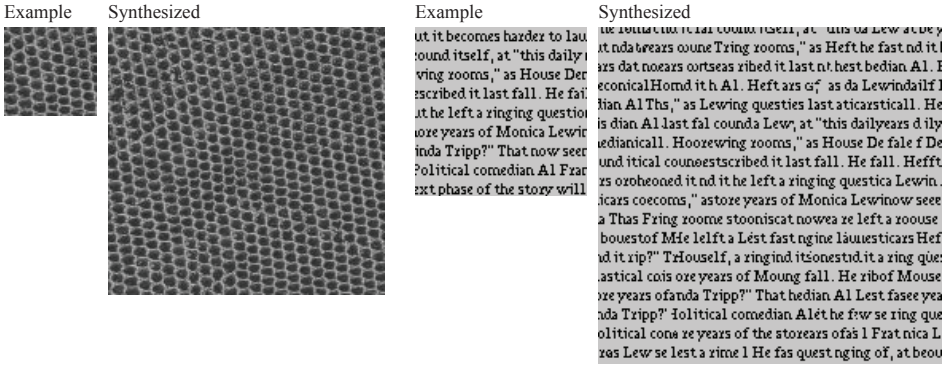


FIGURE 10.6: *Synthesizing a texture image from a small example using incremental reconstruction, and works for textures where the repeated structure isn't obvious. The case on the* **left** *is a texture that is quite periodic, but with random deformations. Notice how realistic the synthesized texture seems to be, and look for repeated cells (they are hard to find). The case on the* **right** *shows a small piece of text from a document that was well known in 1999, together with a synthesized image expanding that example. Text is hard, because although it does have structure (lines), it isn't periodic. Notice how the synthesized text is almost readable.* Image credit: *Elements of Figure 3 from "Texture Synthesis by Non-parametric Sampling", IEEE International Conference on Computer Vision, Corfu, Greece, September 1999.* **No permission yet**

10.2.1 Non-Local Means

Here is one strategy to exploit the other patches. Estimate the true value of the pixel in the center of a target patch as a weighted sum of all other pixels, where the weight is big when the patch around the pixel is similar to the target patch and small when it isn't. These weights should be normalized to add up to one. The big difference between this strategy and filtering is that distant pixels can contribute if they are in comparable patches. This approach should yield a good estimate of the true value, at a ferocious cost. You need to look at every pixel in the image to estimate the true value of a single pixel, so estimating the whole image is quadratic in the number of pixels.

The approach is easily formalized. Write $K(\mathbf{p}_{ij}, \mathbf{p}_{uv})$ for a function that compares an image patch $\mathbf{p}_{ij}$ around the i, j'th pixel with the image patch around the u, v'th pixel. This function should be large when the patches are similar, and small when they are different. A useful estimate of the pixel value $\mathbf{x}_{ij}$ at i, j is then

$$\sum_{uv \in \text{image}} \frac{K(\mathbf{p}_{ij}, \mathbf{p}_{uv})\mathbf{x}_{uv}}{\sum_{kl \in \text{image}} K(\mathbf{p}_{ij}, \mathbf{p}_{lm})}.$$

Notice that the weights sum to one. The estimate clearly depends quite strongly on the choice of K.

The gaussian Kernel: One natural choice uses SSD between patches. Write

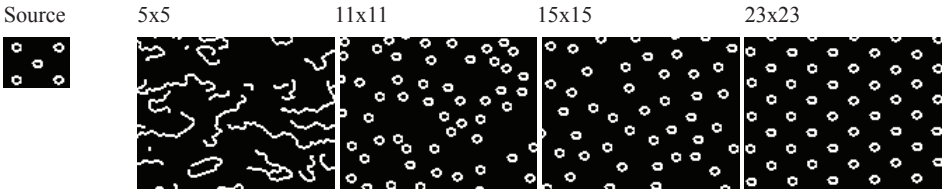


FIGURE 10.7: *The size of the patch used for texture synthesis matters a great deal. Various synthesized textures using differently sized windows and the same original example image. The smallest window cannot see the whole of a ring; the next can, but does not see that spacing is regular; a larger window can see relatively regular spacing; and a very large window simply copies the example.* Image credit: *Elements of Figure 3 from "Texture Synthesis by Non-parametric Sampling", IEEE International Conference on Computer Vision, Corfu, Greece, September 1999.* **No permission yet**

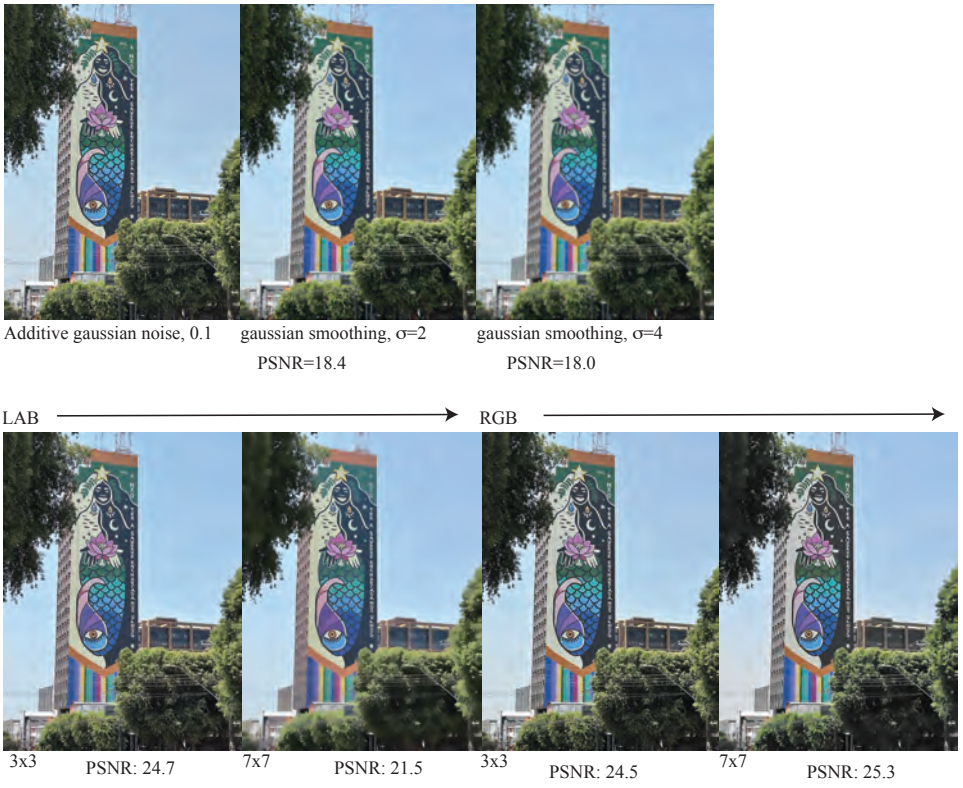


FIGURE 10.8: **Top row:** *Gaussian smoothing suppresses noise, but blurs edges, whereas non-local means preserves edges while smoothing gaussian noise (***bottom row***).* Image credit: *Figure shows my photograph of a building in downtown Manaus.*

$\text{NSSD}(\mathbf{p}_{ij}, \mathbf{p}_{uv})$) for the sum of squared differences between the two patches normalized to deal with the number of pixels in the patch (exercises), write σ for some scale chosen to work well, note that I have suppressed the size of the patch, and use

$$K_{\text{NSSD}}(\mathbf{p}_{ij}, \mathbf{p}_{uv}) = e^{\frac{\left(-\text{NSSD}_{(\mathbf{p}_{ij}, \mathbf{p}_{uv})}\right)}{2\sigma^2}}.$$

The method described here is sometimes known as *non-local means*. As described, it is very slow (quadratic in the number of pixels). Methods to speed it up remain difficult, and are out of scope (**exercises**). As Figure 10.8 shows, non-local means can suppress a great deal of noise without blurring edges.

Some details are important. Using larger patches will tend to increase the computation time, and can improve denoising up to a point. If the patch is too small, there will be many matches but some noise will be preserved. Similarly, if the patch is too large, there will be too few matches to be helpful. Further, the choice of color representation has an effect (see Figure 10.8). You should expect this. RGB values are somewhat correlated (Section 7.1.1), while LAB values are not. This means that some SSD values computed from RGB will overstate the difference between patches (**exercises**).

The gaussian kernel weights down patches that are different from the target patch, but pays no attention to the distance between patches. A natural extension, known as the *bilateral filter*, downweights patches based on their distance. This gives

$$K_{\text{bilat}}(\mathbf{p}_{ij}, \mathbf{p}_{uv}) = e^{\frac{\left(-\text{NSSD}_{(\mathbf{p}_{ij}, \mathbf{p}_{uv})}\right)}{2\sigma^2}} e^{\frac{\left(-\left[(i-u)^2+(j-v)^2\right]\right)}{2\sigma_d^2}}$$

where σ_d controls the rate at which a patches contribution falls off with distance. The bilateral filter admits significant speedups (**exercises**).

10.3 DENOISING AND REPRESENTING IMAGES WITH PATCH DICTIONARIES

10.3.1 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 10 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 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

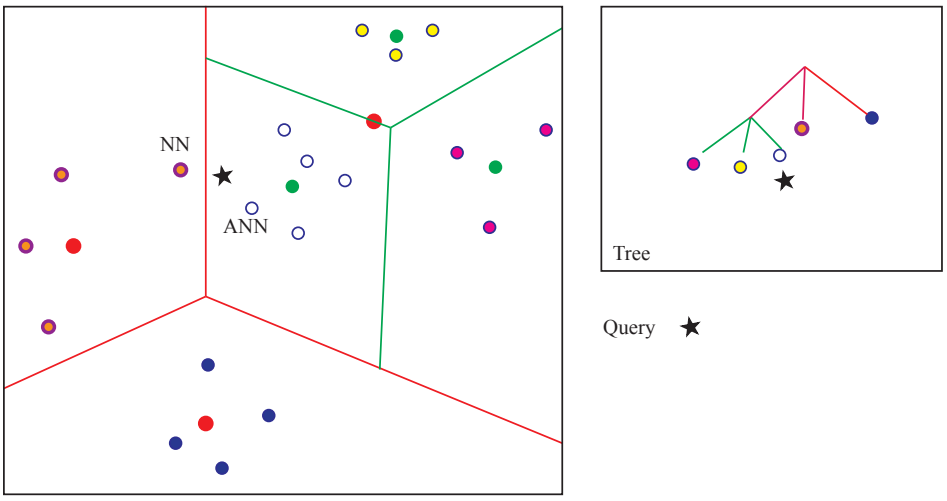


FIGURE 10.9: *Finding an approximate nearest neighbor using hierarchical k-means is easy, but finding the true nearest neighbor is much harder than it looks. On the* **left**, *a set of data points in 2D. The* **red lines** *delineate the regions of the plane that are closest to the three cluster centers 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.*

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

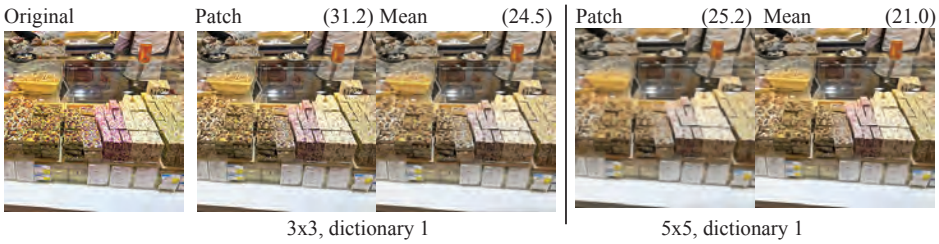


FIGURE 10.10: *Patch based reconstructions using patches from large image dictionaries can be very successful. On the* **left**, *the original image. Others show reconstructions using the approximate nearest neighbor (***Patch***) or the mean of the query cell (***Mean***) using patches of the size indicated. 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.*

which the query lands. This isn't practical: the number of neighboring cells could 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 **??**).

10.3.2 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 10.3.1 describes. As Figure 10.10 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 10.10 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,

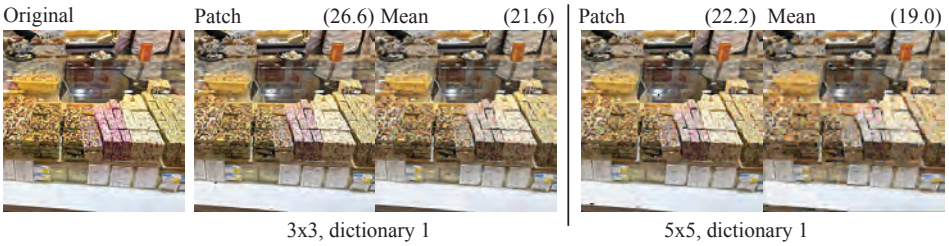


FIGURE 10.11: *Choosing whether patches overlap or not has significant effects. On the* **left**, *the original image. Others show reconstructions using the approximate nearest neighbor (***Patch***) or the mean of the query cell (***Mean***) using patches of the size indicated. For these reconstructions, 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 10.10. 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.*

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 10.11 shows examples of the effect). If the patches do overlap, reconstructing the image from the patches 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 10.10).

As Figure 10.10 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 10 would not work). Whether an image collection is big enough depends on the size of the patch

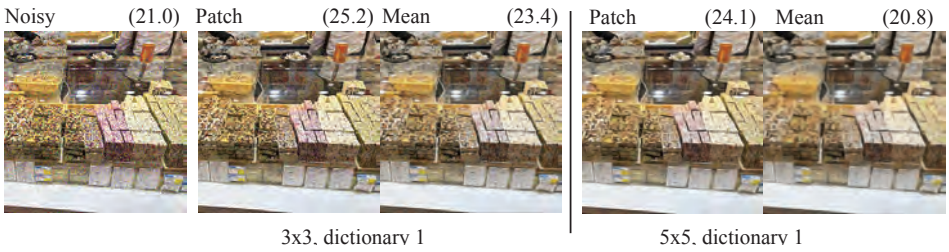


FIGURE 10.12: *Approximate nearest neighbor matching yields quite good denoising. On the* **left**, *a noisy version of the image of Figure 10.10. Others show reconstructions using the approximate nearest neighbor (***Patch***) or the mean of the query cell (***Mean***) using patches of the size indicated. 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.*

you are working with (Figure 10.13).

10.4 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) 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.

10.4.1 Vector Quantization for Noise Suppression

Vector quantization can suppress noise rather well (Figure 10.12). 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

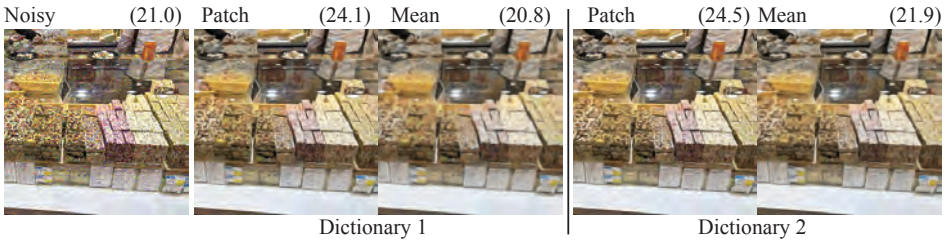


FIGURE 10.13: *The number of patches in the tree has important effects on the reconstruction. On the* **left**, *a noisy version of the image of Figure 10.10. Others show reconstructions using the approximate nearest neighbor (***Patch***) or the mean of the query cell (***Mean***) using* 5×5 *patches. 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.*

application of this simple idea.

10.4.2 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 20.2.1 requires a vector of fixed dimension. Chapter 20 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.

10.5 YOU SHOULD

10.5.1 remember these definitions:

The masked normalized SSD 176

10.5.2 remember these procedures:

10.5.3 be able to:

- FOO

EXERCISES

QUICK CHECKS

10.1. Do you expect to observe every possible set of RGB pixel values in images? why?

10.2. Why do you need a masked normalized SSD to inpaint missing regions?

10.3. Section 10.1.3 has: "Just tiling the texture won't work. The patches may not join up properly" – Explain.

10.4. You wish to inpaint an hole in an image. Why does the order you choose to fill in pixels matter?

10.5. You wish to inpaint an hole in an image. Suggest a good order in which to visit pixels to inpaint. Would this work for every case?

10.6. You want to increase the size of a patch of texture as in Section 10.1.3. Why is it important to choose from several patches at random?

10.7. Section 10.3.2 has: "The leaves of the tree used for matching each contain a set of patches which should be quite similar to one another" Why is this the case?

10.8. Section 36.3 has: "Then the noisy patch should usually be in the same cell as the original patch (check Figure **??**)" Why is this the case?

10.9. You have two patch dictionaries that you use for denoising. Each performs about the same at denoising tests. One is for 5×5 patches, and the other is for 7×7 patches. Which one has more patches in it?

LONGER PROBLEMS

10.10. This exercise explores the expected effect of matching distance on inpainting.

- **(a)** You inpaint a hole in a color image using patches from that image, but matching using only intensity. Describe the errors you expect.
- **(b)** Section 10.2.1 has: " RGB values are somewhat correlated (Section 7.1.1), while LAB values are not. This means that some SSD values computed from RGB will overstate the difference between patches." What effects do expect from choosing RGB over LAB when you inpaint a hole in a color image using patches from that image?
- **(c)** Section 10.2.1 has: " RGB values are somewhat correlated (Section 7.1.1), while LAB values are not. This means that some SSD values computed from RGB will overstate the difference between patches." What effects do expect from choosing RGB over LAB when you inpaint a hole in a color image using a dictionary of patches built from many images?

10.11. This exercise explores the difficulty of improving approximate nearest neighbor estimates by looking at neighboring cells.

- **(a)** Divide d dimensional space into cells by splitting each dimension into k intervals. Show there are k^d cells.
- **(b)** Divide d dimensional space into cells by splitting each dimension into k intervals. Show most cells have 2^d neighbors.
- **(c)** You have a query point that lies in a cell in a tree built by hierarchical k means, and wish to improve your estimate of the approximate nearest neighbor by looking at neighboring cells. How many neighboring cells could there be? (careful – this number can't exceed the number of leaves on the tree).

10.12. This exercise explores deblurring using patch based denoisers
- **(a)** You deblur a lightly blurred image using a patch-based denoiser. What errors do expect at patch boundaries?
- **(b)** Could you improve the deblurring by working with overlapping patches? In particular, imagine you deblur the first patch by matching; if the patches overlap, you now have some information about the deblurred version of the neighboring patches. Could you use it?
- **(c)** You choose to deblur a heavily blurred image using a patch-based denoiser and non overlapping patches. What might go wrong?
- **(d)** Could you improve the deblurring by working with overlapping patches? In particular, imagine you deblur the first patch by matching; if the patches overlap, you now have some information about the deblurred version of the neighboring patches. Could you use it?

PROGRAMMING EXERCISES

10.13. This exercise will implement simple patch based inpainting for a single image. It is straightforward to build a slow solution quickly. Take a textured color image, at least 350 pixels on each edge, and replace a circular set of pixels with radius at least 10 pixels in the center of the image with zeros. Now inpaint this patch by matching 5×5 patches in that image, using the procedure in the text. Assume that you know which pixels should be inpainted, and proceed in angular order.
- **(a)** What happens if you proceed in left to right order, starting at the top?
- **(b)** What happens if use 21×21 patches instead?
- **(c)** What happens if you match on intensity rather than RGB?
- **(d)** Adjust patch size and matching order so you get the best result on your original picture. Now take a second, substantially different, picture, make a hole in it as in the first picture, and inpaint using your best settings. Is the result still good? Can you improve on it?

10.14. This exercise will investigate non-local means based denoising. Implement non-local means as described in the text. This is easy to do, but will be agonizingly slow. Apply it to denoising one of the images in your collection and compare the result if you denoise with a simple gaussian filter. For noise, choose small numbers of pixels uniformly and at random and flip them to randomly chosen colors. This is a case where you expect the gaussian filter to perform badly. Is non-local means better? Is it better to use RGB or LUV when denoising by smoothing with a gaussian?

10.15. This exercise will investigate non-local means based denoising in more detail. Obtain at least 100 color images, at least 350 pixels on edge. You need a somewhat faster non-local means than you are likely to be able to write in reasonable time. Use an API implementation of non-local means (I used the version in scikit image which you can find at `https://scikit-image.org/docs/0.25.x/api/skimage.restoration.html#skimage.restoration.denoise_nl_means`). Most APIs have a slow version (essentially, an optimized version of what you did in the previous subexercise) and a faster version (which uses a clever trick to evaluate the distance between patches quickly and ignores patches that are too far away).
- **(a)** Split your set of color images into 20 train and 80 validation images. Use the train images to choose the scale of the gaussian filter that gives the best mean PSNR for denoising. For noise, choose small numbers of pixels

uniformly and at random and flip them to randomly chosen colors. Use the train images to choose the color representation (RGB or LUV) that gives the best mean PSNR when denoising by smoothing with a gaussian. Now compute the mean PSNR for denoising the test images using the chosen color representation and gaussian scale.

(b) Use the train images to choose settings for your API that produce the best mean PSNR using the same noise model. Use these settings to compute the mean PSNR for denoising the test images using the chosen settings.

10.16. This exercise will implement patch based denoising using a patch dictionary. It is moderately elaborate.

(a) Obtain a collection of at least 10, 000 varied images of a reasonable size. I used the ImageNet validation dataset, which I found at `https://www.kaggle.com/datasets/titericz/imagenet1k-val`, but there are now many such. Another reasonable choice would be one of the Laion datasets (start at `https://laion.ai`), but be aware these are huge. From this dataset, build a collection of 100, 000 patches of size 5×5. I did so by selecting two patches each in random locations from 50, 000 images. Ensure your code will work for larger or smaller patches.

(b) Using these patches and hierachical k-means, build a tree. Aim to have a few hundred patches in each leaf, so your tree can be quite shallow if you use a reasonable k (large numbers of 10s to small numbers of hundreds). It's a good idea to be confident that your tree is what you think it is and that you can walk a patch down the tree before proceeding. You can do so by passing in one of the original patches and checking you end up in the leaf that contains it. Another useful check is adding a little noise to original patches and checking that mostly you end up in the right leaf.

(c) Obtain at least 100 images *not in the original collection* but of a comparable size. Add stationary additive independent zero mean gaussian noise to these images, then reconstruct them using the patch dictionary. To do this, find the leaf corresponding to the query patch, then find the closest patch in that leaf. What mean PSNR do you get? is it better to use overlapping patches or not? What happens if you replace the closest patch in the leaf with the mean of the patches in the leaf?

(d) Obtain at least 100 images *not in the original collection* but of a comparable size. For these images, choose small numbers of pixels uniformly and at random and flip them to randomly chosen colors. Now reconstruct them using the patch dictionary. To do this, find the leaf corresponding to the query patch, then find the closest patch in that leaf. What mean PSNR do you get? is it better to use overlapping patches or not? What happens if you replace the closest patch in the leaf with the mean of the patches in the leaf?

(e) Can you improve your results in the last two subexercises by using larger patches (but the same size dictionary)?

(f) Can you improve your results in the last two subexercises by using larger patches and a larger dictionary?

(g) Can you improve your results in the last two subexercise by backtracking in the tree? At a leaf, you backtrack by finding its parent, then finding the child in the parent whose mean is next closest to the query patch, then querying that leaf. Clearly, you can investigate more leaves, and go higher up the tree. It is important to backtrack by only a fixed number of probes (the leaves you look at), or else the procedure boils down to a

very slow search of the whole tree.