CHAPTER 15

Learned Image Codes

Image denoising with a filter estimates the value of a pixel at a location with an estimator (smoothing filter; median filter) that gives a good estimate of the pixel value even if the image is noisy. This chapter extends this idea dramatically. Assume you can build a code that (a) represents a pixel neighborhood; (b) can be estimated correctly from a noisy image; and (c) can be used to reconstruct the denoised image. You could use this construction to denoise an image. Further, the construction yields others that make it possible to, for example, predict depth from a single image (Chapter **??**).

15.1 ENCODING AND DECODING

Recall from Section 4.1 that linear filters are a form of pattern detector. A natural way to build an image representation is in terms of a range of patterns that (a) commonly appear in images; (b) differ from image to image; (c) can be detected accurately in noisy images; and (d) can represent all that is happening in any image. These patterns would need to be detected at different scales, because the image will tend to zoom in or zoom out.

An *encoder* is a device that maps an image into a representation in terms of patterns, patterns of patterns, etc. that are present, sometimes called a *latent representation*. Convolution with some kernel is a very simple example of an encoder. A simple pattern detector (convolution followed by ReLU) is a more interesting example. Even more interesting is something that detects patterns of patterns. Building a more sophisticated encoder is difficult if you choose the convolution kernels by hand, however. The alternative is to learn these kernels.

One way to learn encodings is to build an encoder together with a *decoder*, which is a device that maps the latent representation back into an image (rather like interpolating samples with a convolution). Learning is by forcing the encoder-decoder pair to be an *autoencoder*, which means the decoded object is the same as the image that goes into the encoder. This chapter sets up the main framework of encoders, decoders and learning weights in the context of autoencoding. If you actually build an autoencoder using the information in this chapter, it won't work. Building one that works requires a number of important practical tricks, the subject of the next chapter. Properly built, autoencoders are extremely useful.

15.1.1 Blocks and Convolutional Layers

Section 4.1.6 described multi-channel convolution, which takes a filter bank and a 3D block of features. There are two spatial dimensions, corresponding to x and y in the image, and one dimension – usually referred to as the *feature dimension* – comparable to the color channel in a color image. The result is another such 3D block of data (Figure 15.1). The feature dimension of the new block is given by

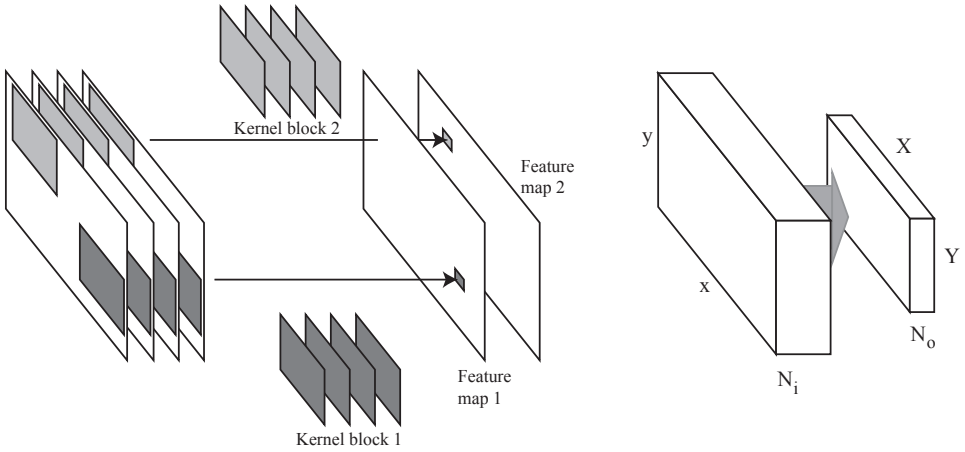

FIGURE 15.1: *Multi-channel convolution can be abstracted as a convolutional layer, a linear operation that takes a block of data and produces a block of data. The operation (details in Section 4.1.6) takes a N_i channel block of data (where $N_i = 3$ for a color image) with dimension $N_i \times x \times y$ and a bank of N_o kernels, each of which is $N_i \times d \times d$. Apply each filter in the bank to the input block to produce one feature channel of dimension $1 \times X \times Y$, and add the bias for that channel to each element. Now stack each of these channels, to produce a block of data that is $N_o \times X \times Y$. Fixed parameters are the input number of features N_i; the output number of features N_o; the kernel size d; the stride s; and the padding p. I will describe these layers as N_i, N_o, d, s, p in figures that follow. The detailed relationships between x, X and d (etc.) depend on choices about stride, padding and so on* **exercises** .

the number of filters. The spatial dimensions are largely the same as that of the original block, with small changes depending on the padding (Section 4.1.2) used.

In the original description of multichannel convolution, each kernel is placed at every sample point to compute the result. Skipping sample points appropriately will have the effect of downsampling. The *stride* of a multichannel convolution controls this skipping. If the stride is s, the kernel is placed at every s'th sample point, meaning the block gets smaller for $s > 1$. It is often convenient to add some offset to the result of each kernel in the filter bank. Doing so could, for example, shift the operating point of the ReLU for a given pattern – faint versions of the pattern may get no response, as in Section 4.1.4. This constant is known as the *bias*.

Write $\mathcal{I}_{k,ij}$ for the k'th feature dimension at the i, j'th location in the input block which has feature dimension N_i, $\mathcal{K}^{(p)}$ for the p'th kernel in the filter bank which contains N_o kernels, each of which is $N_i \times d \times d$, $\mathcal{B}_p$ for the bias of the p'th kernel in the filter bank, and $\mathcal{N}_{p,qr}$ for the p'th feature dimension at the q, r'th location in the output block. Then

$$\mathcal{N}_{p,qr} = \sum_{kuv} \mathcal{I}_{k,sq-u,sr-v} \mathcal{K}^{(p)}_{kuv} + \mathcal{B}_p.$$

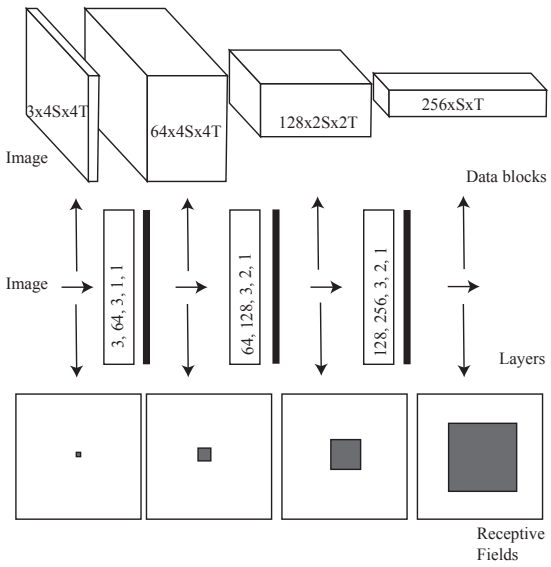

FIGURE 15.2: *The architecture of a very simple convolutional encoder, visualized in terms of data blocks (***top** *- the notation $f \times x \times y$ means the block has f channels and spatial dimension $x \times y$), layers (***center** *- the notation N_i, N_o, d, s, p means the convolutional layer accepts N_i channels, produces N_o channels, uses $d \times d$ filters, has stride s and padding p) and receptive fields* **bottom**. *The thick lines represent ReLU layers. I have arranged the filters so that the spatial dimension of the block of features leaving the encoder is $1/4$ of that arriving. As is typical, the data blocks get spatially smaller but have larger feature dimension. What goes in is the image; you should think of the next block as pattern detector scores; the next as pattern-of-pattern detector scores; and so on. Effective convolutional encoders are often significantly deeper and involve further architectural practices, below.*

This operation is referred to as a *convolutional layer*. The values of the filter kernels and the bias will be learned. Fixed parameters are the input number of features N_i; the output number of features N_o; the kernel size d; the stride s; and the padding (which doesn't appear in this expression).

Convolutional layers turn blocks into blocks. The ReLU of Section **??**, applied elementwise, will also map a 3D block to another 3D block of the same dimensions, and is another layer. In general, layers turn blocks into blocks, and further examples of layers will appear later.

15.1.2 Convolutional Encoders

Now imagine applying a convolutional layer with stride one followed by a ReLU to an image. The result is a block of data where, at each location, there is a measure of the goodness of match between the image around that location and each of the filters in the bank. This is a local description of the image, but it can be made much richer, by passing the description into another convolutional layer

followed by another ReLU. The block that comes out can be thought of as detecting patterns of patterns – structures that are more complicated than those encoded by a simple filter. This block can usefully be passed into yet another convolutional layer, followed by yet another ReLU, and so on. If one applies multiple layers, the output block will be significantly redundant, because the receptive fields for neighboring elements may be moderately large, and will very largely overlap. A natural cure is to apply a layer with stride 2 (or possibly even larger).

The window of pixels in the original image that is used to compute the value at some location in a data block is referred to as its *receptive field*. Usually, all that matters is the size of the receptive field, which will be the same for every location in a given block if we ignore the boundary of the input image. The receptive field of a location in the first convolutional layer will be given by the kernel of that layer. Determining the receptive field for later layers requires some bookkeeping (among other things, you must account for any stride or pooling effects, **exercises**).

The simplest convolutional encoder, as in Figure 15.3, consists of a sequence of convolutional layers, each followed by a ReLU; most of the convolutional layers have stride 1, but there are occasional layers with stride 2. Practice has shown that data blocks should shrink spatially relatively slowly and grow in the feature dimension quite fast. The block of data that comes out of a convolutional encoder is a set of features. The particular features are heavily dependent on the filter kernels in each layer, but in principal could form an extremely rich and detailed image description. If it is an image representation, you should be able to get an image out of it.

15.1.3 Unfiltering

Imagine you have a filtered image, and wish to recover the original. Two strategies from the previous chapters seem plausible. You might apply strategy of Section 7.3.1, but now where the linear operator $\mathcal{B}$ is replaced by whatever filter had been applied. A regularized reconstruction would be linear and shift-invariant in the input, too. The results of that section suggest that the outcome might not be perfect, but could be acceptable.

Alternatively, you would find the Fourier transform of the filtered image, divide by the Fourier transform of the filter, then inverse Fourier transform. Notice that the convolution theorem means that doing so involves convolving the filtered image with some other filter (apply an inverse Fourier transform to the reciprocal of the Fourier transform of the original image). There are some obstacles that need to be dealt with – there might be zeros in the Fourier transform of the filter, for example – but the procedure should seem do-able. Either argument yields that the image can be reconstructed by convolving with some filter, sometimes called a *reconstruction kernel*. Equivalently, the reconstructed image consists of a weighted sum of copies of a particular pattern – the reconstruction kernel – placed at each location.

15.1.4 Convolutional Decoders

It is possible to recover an image from the more complex representation produced by a convolutional encoder, with some conditions. If you think of the encoding

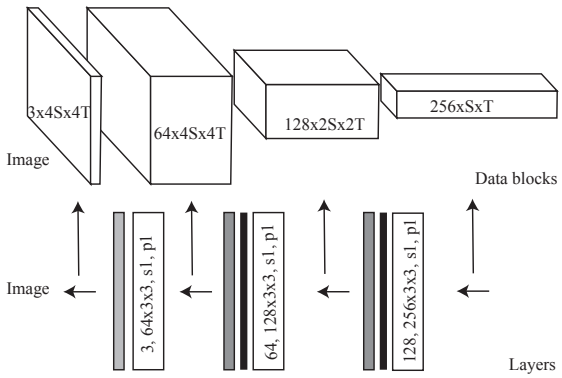

FIGURE 15.3: *The architecture of a very simple convolutional decoder, visualized in terms of data blocks (***top***), layers (***center*** - the notation f, $d \times x \times y$, sN, pM means f filters in the block, of spatial extent $x \times y$, accepting a d dimensional layer, with stride N and padding M). The receptive field of a decoder is are not usually discussed, and I have omitted this here. The thin layers represent ReLU layers. The gray layers represent upsampling by 2 in each dimension. The final pale gray layer could be one of a number of things that are intended to deal with the fact that images have a limited range, including the identity (more details in Section 16.2.1). I have arranged the filters so that the spatial dimension of the block of features leaving the decoder is* 4 *times that arriving. As is typical, the data blocks get spatially bigger but have smaller feature dimension. What goes in is the data block of codes; you should think of the next block as pattern-of-pattern-of-pattern maker; the next as a pattern-of-pattern maker; and so on. Effective convolutional decoders are often significantly deeper and involve further architectural practices, below, but this picture covers the major features for now.*

as a map of where particular patterns of patterns of patterns (etc.) occur, then the overall strategy should be clear: reconstruct the image by placing the relevant patterns at each location. Do this by creating a layout of high level patterns, then replacing the components of the high level patterns with a layout of lower level patterns, and so on. At each stage, clip the layouts to avoid negative values accumulating. This process of placing patterns is a convolutional layer, and the clipping is a ReLU layer. So construct a sequence of layers that takes data blocks and produces data blocks, starts with the encoding and ends with an image.

One trick is necessary. The encoding is smaller in space dimensions than the image and has more feature channels. The sequence of layers must, on occasion, make data blocks get bigger in space and smaller in feature channels. Smaller in feature channels is easy to achieve: one just uses fewer filters. Bigger in space is also easy to achieve: regard the data block as being like an image, and upsample it as in Section 2.2 (there are other strategies: **exercises**). The resulting object is known as a *convolutional decoder*.

15.2 LEARNING BY DESCENT

What makes an encoder or a decoder work well is a good choice of filter banks. It turns out that these can (and as far as anyone knows, should) be learned from data. A good choice of filter banks will give a good encoding or a good decoding of images that weren't used in training. Obtaining this *generalization* property takes care. Further, you can't train an encoder to produce the right encoding of an image, because you don't know what that is. There might be many very good encodings of images that differ only by (say) some complicated transform in feature space, but are otherwise equivalent. There isn't any particular reason to choose one of these. The right strategy is to train an encoder-decoder pair together, to form an *autoencoder* – something that can reconstruct an image from a noisy version.

15.2.1 Learning by Descent on a Loss Function

Write $\mathcal{E}(\cdot;\psi)$ for an encoder which accepts an image (in the $\cdot$ slot), produces an encoding, and has parameters ψ (the filter banks). Write $\mathcal{D}(\cdot;\phi)$ for a decoder that accepts an encoding ($\cdot$ slot again), produces an image, and has parameters ϕ (the filter banks). Stack the ψ and ϕ into one vector θ. Write $\mathcal{S}$ for a set of N training images. The i'th image is $\mathcal{I}_i$.

The autoencoder produces some image $\mathcal{O}(\mathcal{I},\theta) = \mathcal{D}(\mathcal{E}(\mathcal{I};\psi);\phi)$ when given $\mathcal{I}$. Construct a cost function $\mathcal{C}(\mathcal{O}(\mathcal{I},\theta),\mathcal{I}_i)$ that compares the output of the autoencoder to $\mathcal{I}$. This cost function is typically a weighted combination of the L2 norm and the L1 norm (Section 7.2.2).

Now write

$$\mathcal{L}_{\mathcal{S}}(\theta) = \frac{1}{N}\sum_{i\in\mathcal{S}} \mathcal{C}(\mathcal{O}(\mathcal{I}_i,\theta),\mathcal{I}_i)$$

for the *loss* – an average over a set $\mathcal{S}$ of images of the cost per image. The problem is to find a θ that produce an acceptably small value of the loss. In an ideal world, $\mathcal{S}$ would be all possible images, but this isn't practical. Instead, train on some large set of images (the *training set*). If this set is large enough and representative enough, expect that the autoencoder will also have low loss on other images, a property called *generalization*.

Obtaining the best loss for a set of training images might look like an optimization problem, but be careful. Optimization methods look for true optima (or points that are very close to them). In this problem, a value of θ that gives a low loss may be better than the value that exactly minimizes the loss on the training set. What is important is the autoencoder behaves well on new, future images – equivalently, that the loss on some other, unknown, set of images is small. The best value of θ on the training set may well incorporate special properties of the training set, and so behave badly on other sets, whereas a value of θ that has low loss on the training set might generalize.

Furthermore, viewed as a pure optimization problem, this problem is quite hard. There will be a lot of filters, and so a lot of parameters, otherwise there might not be a strong reason to learn the parameters. The objective is very expensive to evaluate exactly, because autoencoders are regularly trained on hundreds of thousands to millions of images. The objective $\mathcal{L}$ is not quadratic (and, as the encoder or decoder have ReLU's in them, not even everywhere differentiable). A

true second order method is likely hopeless, because there are a lot of parameters and so the Hessian will be enormous. A conventional first order method is going to have problems because evaluating the gradient would require summing over a very large number of images. Further, line search isn't practical, because evaluating the objective exactly is impractical.

15.2.2 Stochastic Gradient Descent

A family of first order methods is very successful at finding good values of θ. All members of the family depend on the fact that a very good estimate of a population mean can be obtained by drawing a small sample uniformly and at random, then computing the mean of that sample. So, for example, the average weight of a mouse (which isn't a random variable, but could only be evaluated by weighing all mice and averaging) could be estimated very accurately by drawing a random sample of B mice and averaging their weights. The resulting average is a random variable, with an approximately normal distribution, whose mean is the true mean and whose standard deviation is $\frac{1}{\sqrt{B}}$ times the standard deviation of the population weight. In the case of the loss function, choose a sample size B – usually called a *batch size* – draw $\mathcal{B}$, a set of B images $\mathcal{I}_j$ drawn uniformly and at random, and form

$$\nabla_\theta \mathcal{L}_\mathcal{B}(\theta) = \frac{1}{B} \sum_{j \in \mathcal{B}} \nabla_\theta \mathcal{C}(\mathcal{I}_j; \theta)$$

and use this as an estimate of

$$\nabla_\theta \mathcal{L}_\mathcal{S}$$

to take a descent step. Write

$$\hat{\nabla}_\theta \mathcal{L}$$

for this estimate. Choose a stepsize η_n for the n'th step, and the descent method becomes

$$\theta_{n+1} = \theta_n - \eta_n \hat{\nabla}_\theta \mathcal{L}.$$

This is *stochastic gradient descent* or *SGD*. Calling η_n a stepsize is dubious (the gradient isn't a unit vector); an alternative is to call it the *learning rate* (which isn't much better because it isn't a rate).

A variety of considerations affect the choice of η_n. If η_n is too big, the procedure can diverge (try it!). If η_n is too small, θ doesn't change very much. In the early stages of training, its likely a good idea to travel quite long distances, so η_n should be large for small n. Similarly, after a large number of steps, it is likely a bad idea to travel a long distance (among other reasons, the estimated gradient might be wrong). It is known that, if (a) $\eta_n \to 0$ as $n \to \infty$ and (b) $\sum_n \eta_n \to \infty$ as $n \to \infty$, then the sequence $\mathcal{L}(\theta_n)$ will decrease toward the value of some local minimum. Stripped of the notation, this should seem fairly obvious: if the distance you can travel is arbitrarily long *and* the step you take decreases with time *and* mostly you go downhill, eventually you'll be close to some form of minimum.

For concreteness, here is one procedure for choosing η_n. Choose some value that is small, but not too small (1e-3 has a following here). Take many steps using that value. Now reduce the value, and continue. Repeat as necessary. One way to

reduce the value is to multiply by a constant. Typically, both the starting constant, the number of steps, and the constant to multiply by are chosen by experiment. The general procedure for choosing η_n is known as *step length scheduling* or *learning rate scheduling*. There is a rich variety of alternative methods in any reasonable API, suggesting (correctly) that different choices work for different applications. A crude – but surprisingly powerful – procedure for keeping track of the training process is to plot the value of training loss as a function of the number of steps. Typically, one averages for some number of steps, then plots. This plot – often called a *learning curve* – can be monitored during training for signs of trouble like divergence. Another useful plot is a plot of loss computed for a held out validation set. If this stays a lot larger than the training loss, that is a sign of trouble.

Notice that some first-order optimization tricks are not available. Descent isn't guaranteed because you might have an unlucky estimate of the gradient. You can't do line-search in the descent direction, because it is too expensive to evaluate the objective function. You can't actually tell whether you're at an optimum, because it is too expensive to evaluate the objective function. As Section **??** shows, a variety of interesting gradient scaling tricks become available and are genuinely helpful.

15.2.3 Evaluating the Gradient by Backpropagation

Descent requires forming

$$\hat{\nabla}_\theta \mathcal{L} = \frac{1}{B} \sum_{j \in \mathcal{B}} \nabla_\theta \mathcal{C}(\mathcal{O}(\theta), \mathcal{I}_j).$$

There is an efficient recursion to compute this, because the predicted output is a function of a number of layers. Each layer has its own set of parameters. Drop the distinction between decoder layers and encoder layers and write the w'th layer as $L_w(\cdot; \theta_w)$ (here θ_w consists of the elements of θ apply to the w'th layer. Write all this out layer by layer, keeping track of the blocks that move through the layers, to get

$$\begin{aligned}
\mathcal{D}(\mathcal{E}(\mathcal{I}; \psi); \phi) &= B_{k+1} \\
&\quad \text{where} \\
B_{k+1} &= L_k(B_k; \theta_k) \\
B_k &= L_{k-1}(B_{k-1}; \theta_{k-1}) \\
&\dots \\
B_1 &= \mathcal{I}
\end{aligned}$$

In this notation, computing the gradient is a straightforward application of the chain rule, which leads to a recursion known as *backpropagation*. The derivation is simple, but tedious, and is relegated to **exercises** . Write $\nabla_{\mathcal{O}} \mathcal{C}$ for the gradient of the loss with respect to the prediction – this is a vector if the prediction has been straightened into a vector. Write $\mathcal{J}_{L_w; \theta_w}$ for the derivative of the function L_w with respect to parameters θ_w, and $\mathcal{J}_{L_w; B_w}$ for the derivative of the function L_w with

respect to inputs B_w. This leads to a recursion:

$$\begin{aligned}
\mathbf{u}_0^T &= \nabla_{\mathcal{O}}\mathcal{C}^T \\
\nabla_{\theta_k}\mathcal{C} &= \mathbf{u}_0^T\mathcal{J}_{L_k;\theta_k} \\
\mathbf{u}_1^T &= \mathbf{u}_0^T\mathcal{J}_{L_k;B_k} \\
\nabla_{\theta_{k-1}}\mathcal{C} &= \mathbf{u}_1^T\mathcal{J}_{L_{k-1};\theta_{k-1}} \\
&\cdots \\
\mathbf{u}_r &= \mathbf{u}_{r-1}^T\mathcal{J}_{L_{k-r+1};B_{k-r+1}} \\
\nabla_{\theta_{k-r}}\mathcal{C} &= \mathbf{u}_r\mathcal{J}_{L_{k-r};\theta_{k-r}} \\
&\cdots \\
\nabla_{\theta_1}\mathcal{C} &= \mathbf{u}_{k-1}\mathcal{J}_{L_1;\theta_1}
\end{aligned}$$

Notice how first you evaluate each layer on its inputs (which are outputs of the previous layer); this is a *forward pass* from the input to the output. You need to do this to ensure you're evaluating derivatives at the right point. Next, you evaluate gradients of each layer from the output to the input, using the results of the forward pass; this is a *backward pass*, and is responsible for the name.

15.3 LOSSES AND GENERALIZATION

Section 15.2.1 was deliberately vague about loss functions. The purpose of a loss function is to "push" the parameters of a learned system in a helpful direction. Keep in mind that the learning procedure follows approximate gradients, meaning that the value of the loss is not usually particularly significant but the gradients are crucial. They should push the system – right now, an autoencoder – to behave in a desirable way. What is important is good behavior *on a future test set*, rather than on the training set. The function used to measure performance on the test set may not be a good – or even usable – loss function, so the loss function used for training may need to be some kind of approximation of the performance measure.

15.3.1 L2 and L1 Losses

Loss functions typically evaluate *residuals* – the difference between what the system provides and ground truth. The *SSD loss* compares a reconstructed training image $\mathcal{R}$ to the ground truth $\mathcal{G}$ by

$$\mathcal{C}_{L2}(\mathcal{R}, \mathcal{G}) = \sum_{ij} \Delta_{ij}^2,$$

where $\Delta_{ij} = \mathcal{R}_{ij} - \mathcal{G}_{ij}$ is the *residual*. This is the square of the L2 norm of Δ, and is sometimes (rather disreputably) referred to as an *L2 loss*. This might seem a natural training loss, but it has an important disadvantage. Reconstructions from an autoencoder trained with an SSD loss tend to be blurry; Figure **??** shows why. The key issue is that the square of a small number is very small.

One way to discourage this blurring is to use an L1 loss as well. Recall from Section 7.2.2 that using an L1 norm as a penalty for the gradient tends to cause

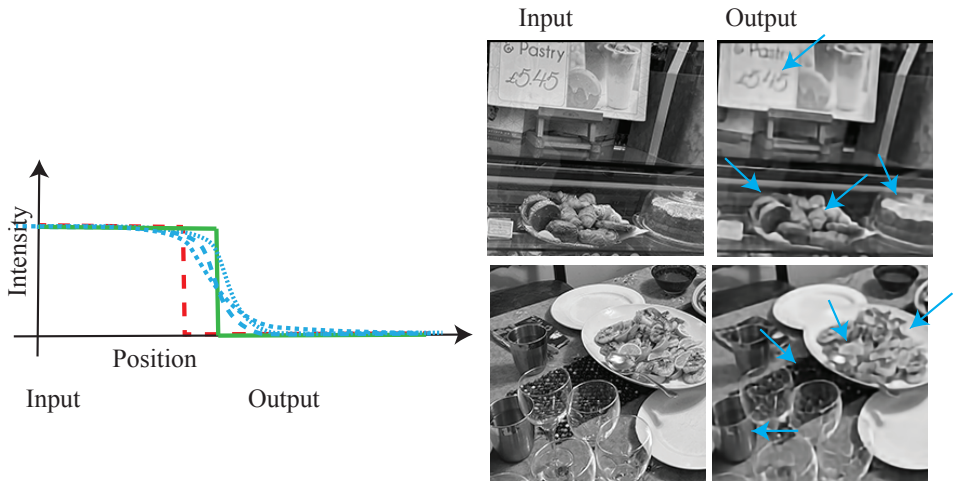

FIGURE 15.4: *The L2 loss tends to produce blurry images. Assume some system is trying to reproduce an image with a strong, sharp edge (***top left***). The* **green** *(full) curve is a cross-section of the intensity through that edge. The* **red** *(big dashes) reconstruction has high L2 loss, because it places a sharp edge in the wrong place, and so is penalized by the square of a large error. The* **blue** *(small and smaller dashes; dots) reconstructions are blurry, and place the edge in only about the right place. Nonetheless, the L2 loss is for these is small, because it is the sum of squares of small errors, and the square of a small number is even smaller. In turn, a reconstruction that has sharp edges pays a high penalty for putting them in slightly wrong locations, whereas a reconstruction that produces blurry edges will have a low loss even if they are somewhat misplaced.* **Top right** *shows detail blocks of input and output for two images. Notice the loss of detail (arrows).* Image credit: *Images are my photographs of a cheerful dinner table and an enticing shop window.*

the gradient to have zeros, assuming the optimization process can cope. Using an L1 term, written

$$\mathcal{C}_{L1}(\mathcal{R}, \mathcal{G}) = \sum_{ij} |\, \Delta_{ij} \,|$$

will tend to encourage the residual to have zeros in it, and will tend to discourage blurring (Figure **??**).

Autoencoders are now usually trained with a weighted sum of L1 and SSD losses. As Section 16.2.1 shows, a variety of other terms might appear as well. This means that you must choose weights. The choice of these weights should have effects on the behavior of the resulting autoencoder.

Here is a way to think about the relative weight of L1 and L2 loss (Section **??** discusses other cases). Assume the system is predicting one number, x, and the intended prediction is t. The residual Δ is $x - t$. The weighted sum of losses is

$$a\Delta^2 + b|\,\Delta\,|.$$

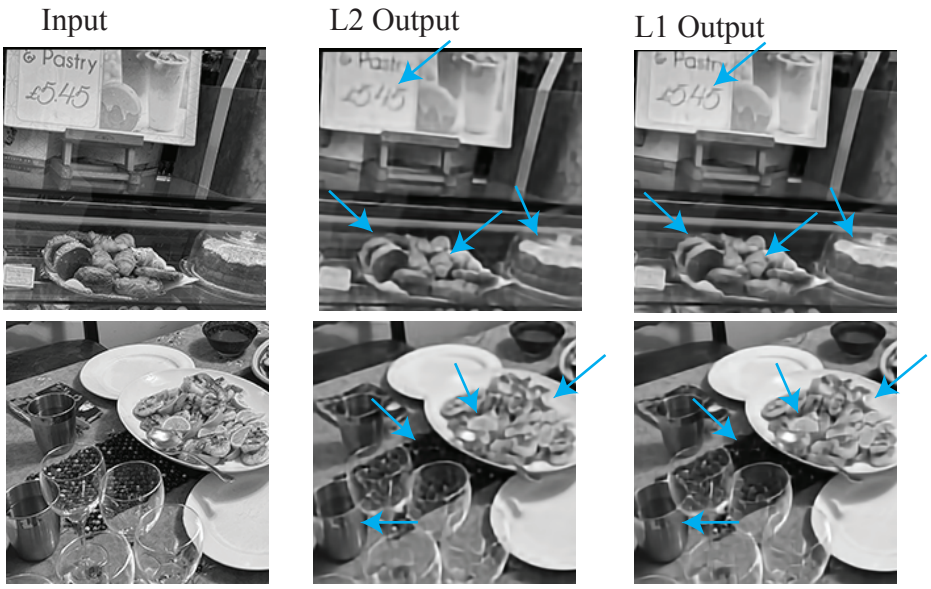

FIGURE 15.5: *The L1 loss tends to reduce blur in reconstructed images.* **Left** *shows detail blocks from images;* **center** *shows the corresponding blocks from an autoencoder trained with SSD loss only; and* **right** *shows reconstructions from a simple autencoder trained with an L1 loss only. The effect is small, but it is there (look at the text in the shop window if you're not convinced).* Image credit: *Images are my photographs of a cheerful dinner table and an enticing shop window.*

This is often referred to as a *L1/L2 loss*. When $|\Delta| = b/a$, the two losses have the same value. As $|\Delta|$ grows, the SSD term dominates; similarly, as $|\Delta|$ shrinks, the L1 term dominates. In turn, this suggests that if x is in the range $0-1$, b/a should be in this range too. If the residual is large, the SSD term should be important, so b/a around 0.1 looks good.

15.3.2 Losses and Gradients

The main point of the exercise is not the loss function, but the gradient that it provides the learned system. In fact, Section **??** gives an example of a procedure that yields gradients without an actual loss (this isn't the usual case). Further, stochastic gradient descent doesn't use an exact gradient, so a loss function that is not differentiable at some points is often not a serious problem.

For example, think about the L1 loss, which isn't differentiable when the residual is zero, and recall the notation of Section 15.2.3. The differentiability problem means that for some example images, at some pixels, we do not know the value of the gradient. These will be the pixels where the residual is zero. At other pixels, the gradient is either 1 or -1. Apply the strategy of using either -1 and 1 for the value of the gradient at the pixels where the residual is zero; you could

choose randomly, or always use one value. No problems result, from the following arguments:

- There are few such pixels, in few images. Any error that results will be swamped by the noise in the gradient caused by random choice of examples.
- If the residual was very slightly different at that pixel, the gradient value you used would be correct.
- This strategy properly represents the subgradient (only convincing if you know what a subgradient is; it isn't worth the trouble to expand this argument for others).

Here is an example of a bad loss. The *indicator function* is a function that tests its argument against a condition, then reports 1 if the condition is true and zero otherwise. For example,

$$\mathbb{I}_{[x<0]}(x) = \begin{cases} 1 & \text{if } x < 0 \\ 0 & \text{otherwise} \end{cases}$$

is 1 when $x < 0$ and 0 otherwise. Note some redundancy here; the condition usually means it is obvious what the argument is, so it is quite usual to write $\mathbb{I}_{[x<0]}$ rather than $\mathbb{I}_{[x<0]}(x)$. The following (BAD) choice of loss could be intended to force an output to be non-negative:

$$\mathcal{C}_{\text{bad}}(\mathcal{I}) = \sum_{ij} \mathbb{I}_{[\mathcal{I}_{ij}<0]}$$

(i.e. count the negative pixels). This (again, very bad) choice of loss is bad not because it isn't differentiable, but because it provides no gradient – for every value of $\mathcal{I}_{ij}$ other than zero, the gradient is zero, and for the remaining case it is undefined.

Smoothing this loss very slightly to produce (say)

$$\mathcal{C}_{\text{bad}}(\mathcal{I}) = \sum_{ij} \frac{1}{1 + e^{a\mathcal{I}_{ij}}}$$

(for a some large number, $a > 0$) does not help. Again, the gradient is tiny for most image values. There is nothing to *push* the system to the right behavior unless it is already very close.

15.3.3 Cheating

Training procedures are very effective at finding parameter values that produce small training losses. These parameter values may not actually do what you expect, an effect sometimes called *cheating*. Cheating is a quite common phenomenon in learned systems. A good rule of thumb is to assume the system is cheating unless you have very strong evidence that it is not. Because cheating occurs when training has found a way to get a small loss without doing what you want it to do, this is equivalent to assuming you don't understand the system and losses as well as you think you do (often a wise assumption).

Here is an example in the simple case of an autoencoder with two encoder layers and two decoder layers. For concreteness, the first layer in this example has 3x3 filters with stride 1. Make one filter that simply reports the image value at the center location (the other filters don't matter). The pixel value is non-negative, and so passes through the ReLU without alteration. The next layer has 3x3 filters and stride 2; this means that mild ingenuity with multiple filters is required to pick out the pixel values to pass on (exercises). Very little further ingenuity is required to ensure the decoder layer produces the original image. Experience shows that searching for parameters using stochastic gradient descent is extraordinarily powerful, and is perfectly capable of finding a set of parameters that cheats like this. This is cheating because the search has minimized the loss function, but the representation isn't actually of any use. Worse, adding layers, filters, and so on might simply increase the scope for cheating while making it more difficult to understand the detailed structure of any particular cheating strategy.

15.3.4 Denoising to Avoid Cheating

The autoencoder is able to cheat because it can pass on the input image. Autoencoding *in and of itself* isn't particularly interesting – why bother coding and then decoding the image if you just want the original – but is a means to an end. The encoding of the image should represent all that is important about the image, and should be robust – if the encoder is presented with a noisy version of the image, it should produce the code for the original image. This suggests training the autoencoder to denoise images. As Chapter **??** shows, pixels near a particular image location contain a great deal of information about the value at that location. Making an autoencoder denoise should force it to exploit everything it can in the neighborhood of a pixel when it encodes the image.

15.3.5 Generalization

A good autoencoder should denoise *all* images, not just the images it was trained on. This property is an instance of a broader idea to do with learned systems, often called *generalization*. The goal of training a learned system is to have it perform well on inputs that are like its training data, but are not exactly the same. Being precise about the meaning of "like ... but not exactly the same" is surprisingly hard.

A system that fails to generalize has found a way to perform well on training images, but not on any other. Typically, this occurs because the system relies on a correlation that is present in the training data, but may not be present in other data. For example, if the noise only changes some bright red pixels, the trained autoencoder might cheat on any pixel that isn't bright red. It is obvious that this choice of noise model is bad, but there may be strong correlations in the training data that are not obvious, and aren't in all relevant data.

There are several strategies to encourage generalization that apply here. The most basic involves using a great deal of training data. This is quite do-able for image denoising, because it is relatively straightforward to obtain very large collections of images (Section 41.2). However many images in your basic training set, you can make this set look significantly bigger by *augmentation*, which creates new

images from old. For a denoising application, notice there are many operations you can apply to an image that result in an image: cropping an image and resizing it; left-right flipping it; up-down flipping it; or making it slightly brighter or slightly darker. Further, you can make it hard for the system to cheat by memorizing examples. It is not a good idea to construct a dataset of noisy/clean pairs in advance, because there might be some unexpected correlation between noise and image. Instead, apply noise to the image when a batch is formed (so the system could see many different noisy versions of the same image).

Another strategy is to discourage large values in the filter coefficients, a practice known as *regularization*. Imagine two filters that get about the same response from a range of real inputs. The one with smaller coefficients is likely a better choice. The large coefficients appear to have no effect on real data (because the filters get about the same response on a range of real inputs), but might produce a large response on some new piece of data that is somewhat unlike the training data. This large response is likely spurious; worse, it may cause a cascade of errors where some other filter responds strongly to the spurious response.

Regularization can be implemented by adding a term to the loss that penalizes the sum of squared parameter values, with a small weight (chosen by experiment, Section **??**). This is equivalent to adding a term to the gradient that "shrinks" the weights (exercises), and so is often referred to as *weight decay*.

Another regularization strategy is *dropout*, where one randomly replaces elements of a data block with zeros during training. This is intended to advantage filters that are robust to error. Dropout will tend to disadvantage a filter that relies too strongly on one input, because that input might be dropped out. Some housekeeping is required to implement dropout properly, because the filter sees a "smaller" input in training (where some inputs might be zeroed) than in test. A good API will have a dropout implementation that takes care of this, and I leave the topic to the manual of your API. Further strategies involve discouraging large values in data blocks (*normalization*) and are dealt with in Section **??**.