CHAPTER 17

Learned Image Codes from Denoising

Filters can be used to produce a description of an image patch in terms of local patterns (Section 6.1), so you can build a code – the response of a set of filters – that represents a pixel neighborhood. The deblurring procedures of Section 7.2.5 (or a little linear algebra **exercises**) suggest you can recover an image from a set of filter responses (or code).

It is natural to want to generalize simple filter based pattern detectors into codes that represent images. Simple filter based pattern detectors shouldn't be linear (recall the examples of Figure 6.1, where the ReLU distinguished between a bright blob on a dark background and a dark blob on a light background). The construction shouldn't require explicit choice of filters – you want to choose the code that performs best at one or more tasks, rather than reason about filters in detail. This suggests that codes are most easily built with a task in mind.

A natural task is denoising. You would denoise an image by making a code from it, then decoding that code. The simplest denoising process – apply a gaussian filter to a noisy image – can be seen in this light (the noisy image is the code, and the gaussian filter decodes this code). Although the simplest image denoising uses a filter to estimate the value of a pixel at each location (Section 6.2), there are limits to a linear denoising process (eg Section 6.2.2). A non-linear function of the image (Section 7.2.2) yielded a fairly successful denoiser, suggesting that the decoding is likely to be non-linear, too. All this suggests the following recipe: encode a noisy image; decode the noisy image into a denoised image; and choose encoder and decoder so that this process works reliably for many images.

This chapter sketches out a simple version of this extraordinarily powerful recipe: how to build and choose simple encoders and decoders. The next chapter adds enough detail so that you could build one that works acceptably. The recipe is hugely powerful because it is adaptable. Chapter **??** describes a procedure to build encoders that draws on the ideas of Chapter 10 and results in extremely effective decoders that can be suborned to encode text as well as images. Chapter **??**) shows how to use it to predict a variety of useful things – depth, surface normal, high resolution image, defogged image – from a single image. A small variation in the recipe will label image pixels as to object (Chapter 21).

17.1 CONVOLUTIONAL ENCODERS

Recall from Section 5.1 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.

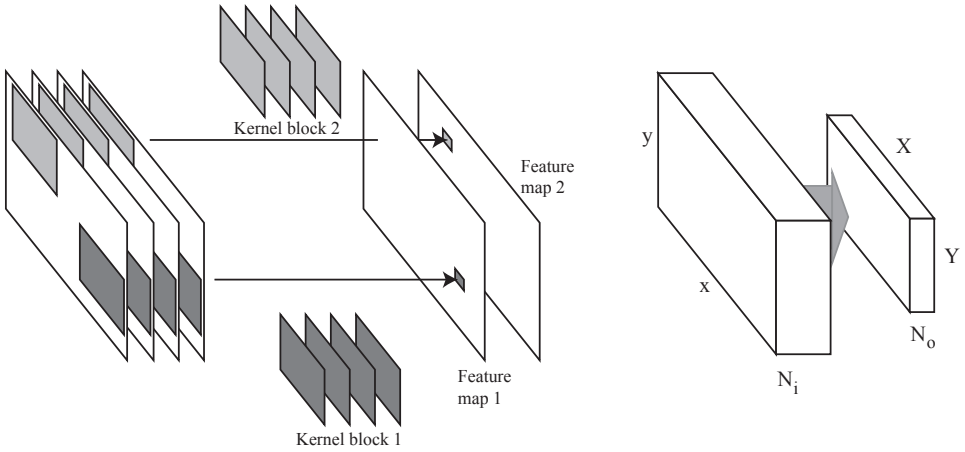


FIGURE 17.1: *Multi-channel convolution can be abstracted as a convolutional layer, a linear operation that takes a tensor and produces a tensor. The operation (details in Section 6.1.4) takes a N_i channel tensor (where $N_i = 3$ for a color image) with dimension $N_i \times x \times y$ and a bank of N_o kernels, each of which is $N_i \times d \times d$. Apply each filter in the bank to the input tensor 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 tensor 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**.

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.

17.1.1 Convolutional Layers and Tensors

Section 6.1.4 described multi-channel convolution, which takes a filter bank and a tensor 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 tensor (Figure 17.1). The feature dimension of the new tensor is given by the number of filters. The spatial dimensions are largely the same as that of the original tensor, with small changes depending on the padding (Section 5.1.3) 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 tensor 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 6.1.3. 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 tensor 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 tensor. Then

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

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

> **Remember this:** *A convolutional layer computes a multichannel convolution of an input tensor with a collection of filters to produce an output tensor. The spatial size, number of filters, stride and padding are prescribed, and the weights of the filters will be learned.*

17.1.2 Convolutional Encoders

Now imagine applying a convolutional layer with stride one followed by a ReLU to an image. Under some circumstances, it can prove advantageous to replace the ReLU with some other nonlinear function (sometimes known as a *activation*). The result is a tensor 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 tensor 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 tensor can usefully be passed into yet another convolutional layer, followed by yet another ReLU, and so on. If one applies multiple layers, the output tensor will be

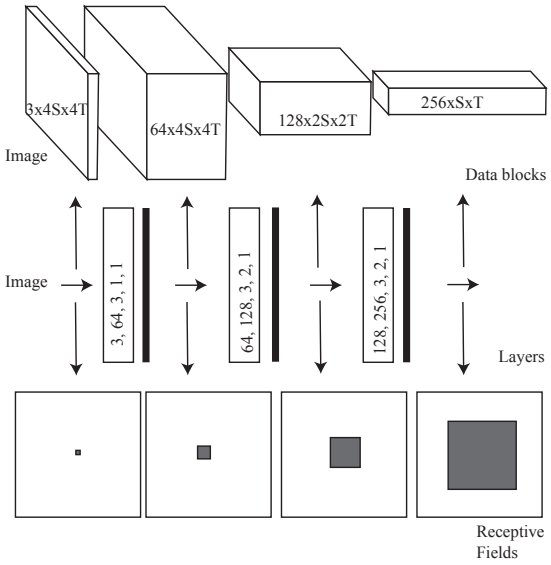


FIGURE 17.2: *The architecture of a very simple convolutional encoder, visualized in terms of tensors (***top** *- the notation $f \times x \times y$ means the tensor has f channels and spatial dimension $x \times y$), layers (***center** *- the notation N_i, N_o, d, s, p means the convolutional layer 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 tensor of features leaving the encoder is $1/4$ of that arriving. As is typical, the tensors get spatially smaller but have larger feature dimension. What goes in is the image; you should think of the next tensor 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.*

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 tensor 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 tensor 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**). It is known that the effective receptive field (the window of pixels that has a large effect on the value) is much smaller than the receptive field, and some important consequences flow from this point (Section **??**).

A convolutional encoder, as in Figure 17.3, 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 tensors should shrink spatially relatively slowly and grow in the feature dimension quite fast. If you interpret a convolutional layer followed by a ReLU as a pattern detector, this is natural. There are more kinds of patterns than there are pixels; and there are more kinds of patterns of patterns than there are kinds of patterns. This means the feature dimension should grow as a tensor moves through the encoder. But patterns are bigger than pixels; and patterns of patterns are bigger than patterns. This means that there will be fewer patterns of patterns in an image than there will be pixels, so the data tensor should shrink in spatial dimensions.

Remember this: *A convolutional encoder consists of a sequence of convolutional layers, each followed by an activation. This activation is almost always a ReLU. The stride of the layers is arranged so that the tensors get smaller spatially as one moves through the encoder. The number of filters in each layer is arranged so that the feature dimension of the tensors increases as one moves through the encoder.*

17.2 CONVOLUTIONAL DECODERS

The tensor 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 find the image that produced the representation. You could search for the thing that is most like an image that also produces the representation when you feed it into an encoder. The trick is to do recover the image without search. This is the job of a *decoder*, which is a device that maps the code back into an image.

17.2.1 Unfiltering to Decode

You can recover the original image from a filtered version with some caveats. You might apply the strategy of Section 7.2.5, 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, so the convolution theorem says there is some filter that will unfilter the image.

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

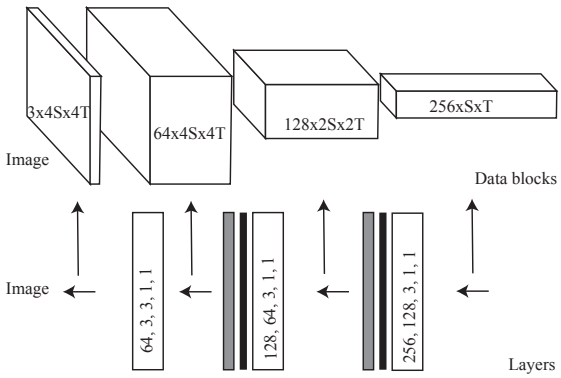


FIGURE 17.3: *The architecture of a very simple convolutional decoder, visualized in terms of tensors (***top***), layers (***center*** - the notation f, $d \times x \times y$, sN, pM means f filters in the tensor, of spatial extent $x \times y$, accepting a d dimensional layer, with stride N and padding 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 18.2). I have arranged the filters so that the spatial dimension of the tensor of features leaving the decoder is* 4 *times that arriving. As is typical, the tensors get spatially bigger but have smaller feature dimension. What goes in is the tensor of codes; you should think of the next tensor 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.*

sum of copies of a particular pattern – the reconstruction kernel – placed at each location.

17.2.2 Convolutional Decoders

The output of a convolutional decoder is a non-linear function of the image, so you should expect a direct application of unfiltering won't work (try it – it doesn't). If you think of the encoding as a map of where particular patterns of patterns of patterns (etc.) occur, then you should be able to reconstruct the image by placing the relevant patterns at each location. Do so 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 you could construct a sequence of convolutional layers followed by ReLU's. This would take tensors and produce tensors, starting with the encoding and ending 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 tensors 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 tensor as being like an image, and upsample it as in Section 3.1 (there are other strategies). The resulting object is known as a *convolutional decoder*.

Remember this: *A convolutional decoder consists of a sequence of convolutional layers, each followed by an activation. This activation is almost always a ReLU. Either an upsampler, or another arrangement ensures that tensors get larger spatially as one moves through the encoder. The number of filters in each layer is arranged so that the feature dimension of the tensors decreases as one moves through the encoder.*

17.3 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. Generally, you adjust the filter banks until you get good behavior at some task, where good behavior is measured using some cost function averaged over many instances of the task. You can't train an encoder to produce the right encoding of an image, because you don't know what that is. However, you could train an encoder-decoder pair together, to form an *autoencoder* – something that accepts an image, makes a code, and then reproduces the image from that code.

It isn't enough to train the autoencoder to reproduce the image it is given; there are many ways to do that, and most do not produce an interesting image code. Instead, you train it to accept a noisy version of an image and produce a clean version of that image. Doing so requires that the encoder computes a code that (a) describes the image (if it didn't, the decoding would fail) and (b) is robust to noise (if it didn't, it would produce something that wasn't correctly denoised). More important, the autoencoder should be able to denoise images *that it hasn't seen in training* – otherwise, it is capable of producing codes only for its training images, which isn't much help. Obtaining this *generalization* property takes care.

17.3.1 Learning by Descent on a Loss Function

Training the autoencoder requires choosing a set of filters for encoder and decoder so that if you pass in a noisy version of an image, the output is close to the original, noise-free image. You adjust the filters so that the measure of similarity is good for a large number of images. Going further requires some notation.

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 $\mathcal{I}$. Write $\mathcal{I}^*$ for a noisy version of $\mathcal{I}$. Construct a cost function $\mathcal{C}(\mathcal{O}(\mathcal{I}^*, \theta), \mathcal{I}_i)$ that compares the output of the auto-encoder to $\mathcal{I}$. This cost function is typically a weighted combination of the L2 norm and the L1 norm (Section 7.2.4).

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 second order method is 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 and so is impractical. This means line search won't work either.

17.3.2 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*

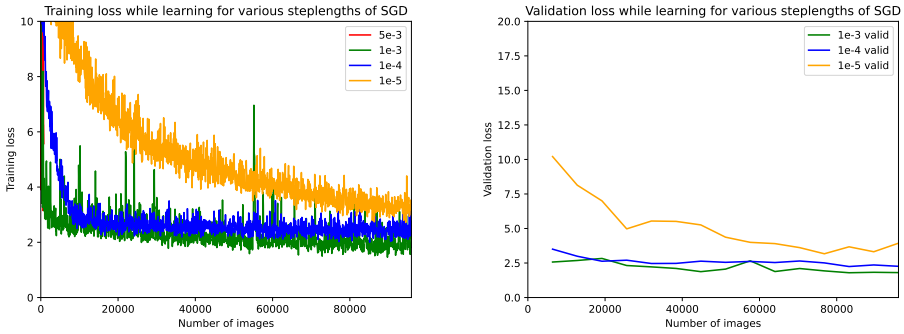


FIGURE 17.4: *Choice of steplength or learning rate can have significant effects on training. The curves show loss for a simple autoencoder in training. On the* **left**, *training loss measured every 640 training images for a variety of steplengths. Notice: the loss is quite noisy (because the gradient is stochastic); for a large steplength (5e-3), training can diverge (the curve disappears fairly fast, and a few batches later the loss is NAN); and smaller steplengths produce slower descent (1e-4 never catches up with 1e-3, and 1e-5 doesn't come close). On the* **right**, *validation loss (evaluated on images the autoencoder has never seen in training), computed every 6400 images. Notice the behavior of the validation loss is tracking that of the training loss, which is a good sign – the autoencoder likely generalizes.*

– draw $\mathcal{B}$, a set of B images $\mathcal{I}_j$ drawn uniformly and at random, and form

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

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 steplength 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*.

17.3.3 Steplength and Learning Rate

Calling η_n a steplength is dubious (the gradient isn't a unit vector); an alternative is to call it the *learning rate* (but it isn't a rate).

A variety of considerations affect the choice of η_n. First, you can't set η_n by linesearch, because you can't evaluate the objective function efficiently (it may be a sum over millions of images). If η_n is too big, the procedure can diverge (try it! Figure 17.4). If η_n is too small, θ doesn't change very much (Figure 17.4). In the

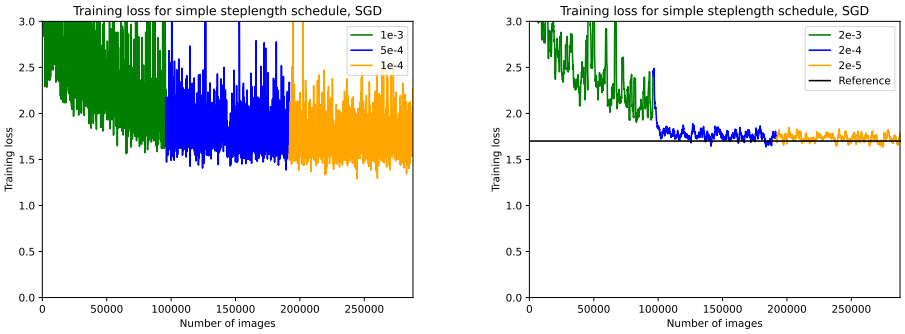


FIGURE 17.5: *In early training, a larger steplength will tend to have the advantage of fast progress, but in later trainining a smaller steplength is preferred because it will result in a finer scale search. Typically, one reduces the steplength with a steplength schedule. On the* **left**, *the steplength has been halved every 100, 000 images. The improvement obtained by the first drop in steplength is clear, but that from the second is harder to see. On the* **right**, *I started with a steplength of 2e-3, and reduced by a factor of 10 every 100, 000 images. I then plotted a 20 item moving average of the loss curve (which smoothes it significantly). Notice that even with this smoothing, the loss with the larger steplength occasionally jumps up; and there is a small but useful improvement from the second drop in steplength. You can spot this improvement by comparing the curve with the horizontal reference line that shows a constant loss value.*

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. There are a variety of interesting procedures to adjust the gradient to improve descent (Section **??**).

17.3.4 Evaluating the Gradient by Backpropagation

Descent requires forming

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

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 tensors that move through the layers, to get

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

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**. To evaluate the gradient, you first evaluate each layer on its inputs (which are outputs of the previous layer); this *forward pass* from the input to the output determines the value of each variable in each layer. You need to do this to ensure you're evaluating derivatives at the right point.

Now 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 (recall these are

matrices **exercises**). You then have a recursion:

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

This recursion means 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.

Remember this: *Train an autoencoder using stochastic gradient descent on a loss. The loss is an average over examples of a function comparing the output to the original, noise-free image. Stochastic gradient descent approximates the true gradient – which is very expensive to compute – by drawing a batch of examples at random, and computing the gradient for that batch, then taking a step in the opposite direction. The size of the step is determined by a scheduling algorithm, but typically starts large and gets smaller. Individual steps do not necessarily improve the loss, but for appropriate step sizes, the loss should drift down. The gradient is evaluated using a recursion called backpropagation*

17.4 LOSSES AND GENERALIZATION

Section 17.3.1 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.

17.4.1 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 17.6 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 7.2.4 that using an L1 norm as a penalty for the gradient tends to cause the gradient to have zeros, assuming the optimization process can cope **exercises**. 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 **??**). This occurs because the small differences that are cheap in the L2 loss are now much more expensive.

Autoencoders are now usually trained with a weighted sum of L1 and SSD losses. As Section 18.2 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

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

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.

Remember this: *Training an autoencoder with an L2 loss alone tends to produce blurred images, because the autoencoder can get a fairly small loss value by putting a blurred edge in a slightly wrong location. Using a weighted combination of L1 and L2 losses can help. If the error is small, the L1 loss should dominate, encouraging small errors to move closer to zero; if the error is larger, the L2 loss should dominate.*

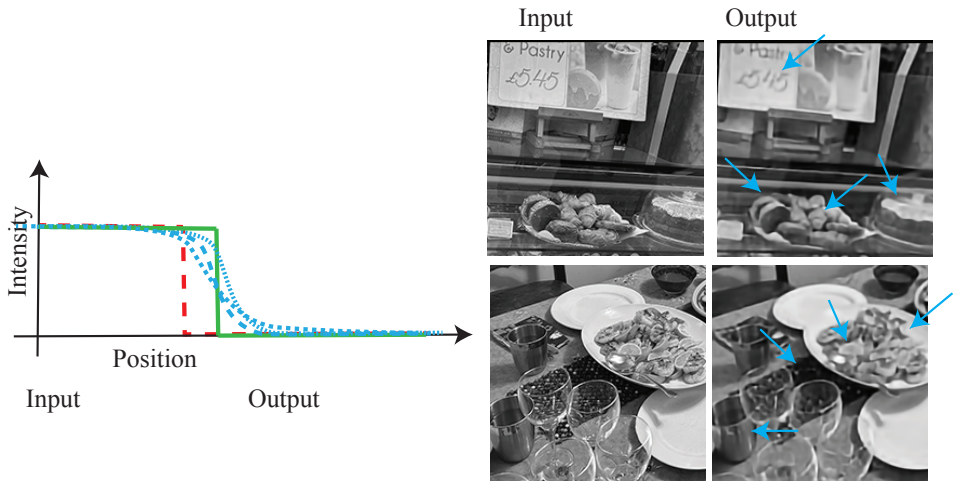


FIGURE 17.6: *The L2 loss tends to produce blurry images. Assume some system is trying to reproduce an image with a strong, sharp edge (***top left***). The* **green** *(full) curve is a cross-section of the intensity through that edge. The* **red** *(big dashes) reconstruction has high L2 loss, because 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 tensors 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.*

17.4.2 Unhelpful Gradients means a Bad Loss

The main point of the exercise is not the loss function, but the gradient that it provides the learned system. What you want is gradients that push the system toward good behavior from any state. Such gradients don't actually have to come from a loss (Section **??**; but this isn't the usual case). Something that might at first glance look like a usable loss may not be, if it provides unhelpful gradients.

Here is an example. 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

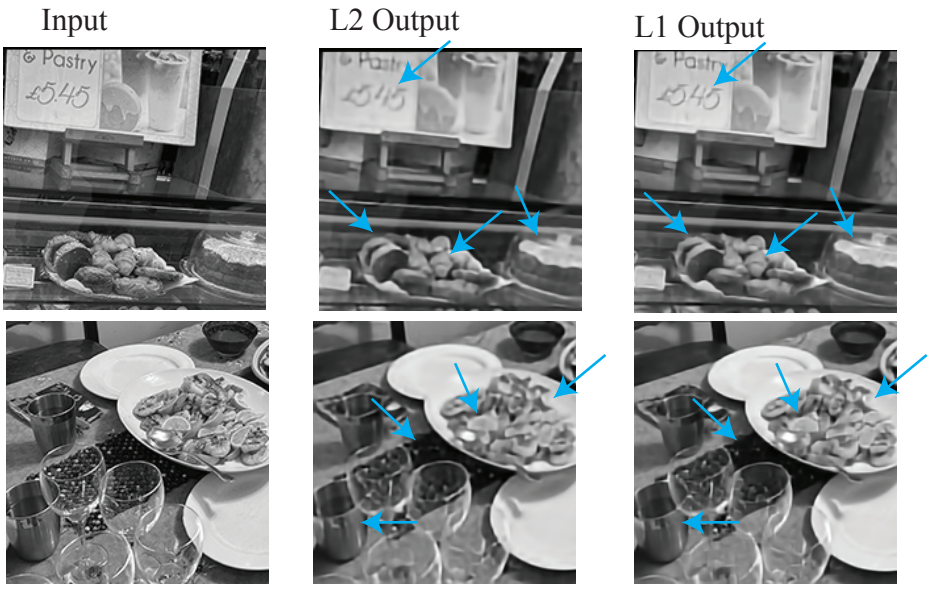


FIGURE 17.7: *The L1 loss tends to reduce blur in reconstructed images.* **Left** *shows detail tensors from images;* **center** *shows the corresponding tensors from an autoencoder trained with SSD loss only; and* **right** *shows reconstructions from a simple autencoder trained with an L1 loss only. 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.*

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.

A loss function that is not differentiable at some points is often not a serious problem. because stochastic gradient descent doesn't use an exact gradient. For example, think about the L1 loss, which isn't differentiable when the residual is zero, and recall the notation of Section 17.3.4. 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).

Remember this: *The loss doesn't matter very much, but the gradients are crucial. You want a loss that pushes the system toward good behavior from wherever it is. It is not usually a problem if losses fail to be differentiable at some points.*

17.4.3 Cheating and Denoising

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

Now assume you train an autoencoder to reproduce its input image. It is likely to be able to do so without actually producing a useful image representation – all it needs to do is pass the input values to the output. Experience shows that searching for parameters using stochastic gradient descent is extraordinarily powerful, and is perfectly capable of cheating 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.

The autoencoder is able to cheat because it can pass on the input image. Forcing it to denoise an image should result in something that produces reasonable image codes. 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. As Chapters 6 and 7 show, pixels near a particular image location contain a great deal of information about the value at that location. Experience shows that making an autoencoder denoise forces it to exploit everything it can in the neighborhood of a pixel when it encodes the image, and so forces it to produce a usable code.

> **Remember this:** *Stochastic gradient descent involves an extremely powerful search, so it is common that learned systems achieve low losses without doing what you expect. You should always suspect that this cheating is occurring unless you have evidence to the contrary. Forcing an autoencoder to denoise is an effective apotropaic against cheating.*

17.4.4 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 36.3). 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 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 tensor 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 tensors (*normalization*) and are dealt with in Section 18.3.3.

Remember this: *You want the autoencoder to denoise images that weren't seen in training. This generalization can be hard to achieve. The two main strategies to encourage generalization are (a) augmentation (making the training dataset bigger by operations on the images) and (b) regularization (which reduces the space of learned systems being searched by ensuring filter coefficients aren't too big or by randomly setting coefficients to zero).*

17.5 YOU SHOULD

17.5.1 remember these facts:

A convolutional layer accepts an input tensor and produces an output tensor using a learned set of filters 318
A convolutional encoder consists of a sequence of convolutional layers, each followed by an activation. 320
A convolutional decoder consists of a sequence of convolutional layers, each followed by an activation. 322
Train using SGD on a loss. 327
L2 losses make blurred images; L1 losses help 328
Loss doesn't matter much, gradients are crucial. 331
Always suspect cheating by learned systems 332
Encourage generalization by augmenting data and by regularization 333

17.5.2 remember these procedures:

17.5.3 be able to:

- Recognize a bank of filters as a way to represent small patterns in images.
- Denoise an image by smoothing with either Gaussian or median filters.
- Form a gradient estimate using derivative of Gaussian filters.
- Use the model of sampled functions in simple calculations.
- Recognize interpolation as a convolution that passes from a sampled function to a continuous function.

EXERCISES

QUICK CHECKS

17.1. You are given a filtered image. How would you recover the original? what might go wrong?

17.2. A multichannel convolution with stride 1, kernel size $2d + 1$, padding d and N_o filters accepts an $N_i \times X \times Y$ block. How big is the block that comes out?

17.3. A multichannel convolution with stride 2, kernel size $2d+1$, padding d and N_o filters accepts an $N_i \times 2X \times 2Y$ block. How big is the block that comes out?

17.4. A multichannel convolution with stride 1, kernel size $2d+1$, padding 0 and N_o filters accepts an $N_i \times X \times Y$ block. How big is the block that comes out?

17.5. A convolutional layer with stride 1 and kernel size $2d + 1$ is followed by a second convolutional layer with stride 1 and kernel size $2d + 1$. How big is the receptive field for a feature in the second layer?

17.6. Section 17.3.4 says: " 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 (recall these are matrices)". Why are these matrices? what are the elements of the matrices?

17.7. You want to ensure that an autoencoder produces a non-negative number at every location. Section 17.4.2 says that

$$\sum_{ij} \frac{1}{1 + e^{a\mathcal{I}_{ij}}}$$

would be a bad choice of loss for $a > 0$ and large. What happens if $a > 0$ and small? what happens if $a < 0$?

17.8. Section 17.4.1 says: "Recall from Section 7.2.4 that using an L1 norm as a penalty for the gradient tends to cause the gradient to have zeros, assuming the optimization process can cope **exercises**." Explain; do you expect gradient descent on an L1 loss produce zeros?

LONGER PROBLEMS

17.9. This exercise derives backpropagation. Write the encoder-decoder pair

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

and the loss $\mathcal{C}(\mathcal{O}(\mathcal{I}_j^*,\theta),\mathcal{I}_j)$.

(a) 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. Under some conventions, it is a row vector, but I will adhere to the convention that every vector is a column vector. Show that

$$\nabla_{\theta_k}\mathcal{C}(\mathcal{O}(\mathcal{I}_j^*,\theta),\mathcal{I}_j) = \nabla_{\mathcal{O}}\mathcal{C}^T\mathcal{J}_{L_k;\theta_k}.$$

(b) Show that

$$\nabla_{\theta_{k-1}} \mathcal{C}(\mathcal{O}(\mathcal{I}_j^*, \theta), \mathcal{I}_j) = \nabla_{\mathcal{O}} \mathcal{C}^T \mathcal{J}_{L_k;B_k} \mathcal{J}_{L_{k-1};\theta_{k-1}}.$$

(c) Use the last two subexercises to deduce the form of the recursion

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