CHAPTER 18

Making an Autoencoder that Works

It is just possible to build autoencoder like that of Chapter 17 in a simple programming environment. I do not encourage the exercise – though it is informative, the knowledge will be hard-won. Even with a great deal of fiddling, the result will work poorly. Some difficulties are now at best minor nuisances. For example, there is a quite extraordinary amount of housekeeping, but software environments will now deal with this for you (Section 18.1). As another example, you have to ensure the output image isn't negative (Section 18.2).

The simple encoder of Figure 17.3 has three convolutional layers. As a result, all the elements in the code it produces – the tensor leaving the last layer – are produced by a relatively simple function of relatively few image pixels. Experience shows that such functions just aren't enough to produce a useful representation of an image. Obtaining good image codes from convolutional encoders seems to require quite *deep* encoders. Here, deep means that there are many layers stacked on one another. Current encoders run from 10's to 1000's of layers, depending on details. A deep encoder requires a deep decoder. In turn, there are many layers between the output of the autoencoder and its input. This makes it very difficult to preserve fine details in the image. It also makes learning difficult. A variety of important changes to the structure of the simple model make a real difference in performance and in learning (Section **??**). Changes to the optimization procedure can be helpful, too (Section **??**).

The autoencoder of Chapter 17 is an example of a *neural network* (or network), a category containing a wide range of of enormously useful approximation procedures. It seems to be hard to define this category crisply. Mostly, the networks one encounters in vision consist of many layers of linear operators (which don't have to be convolutions) followed by nonlinearities (which don't have to be ReLU's). The layers are arranged to make backpropagation possible (you don't have to have a sequence, a directed acyclic graph will work). Neural networks were originally built by gross analogy with anatomy, and many networks contain or consist of *units* (a term I advocate over *neurons*, which pushes the analogy at least one step too far). Units form a weighted sum of their inputs, apply a non-linearity, and report the result. You can see a convolutional layer followed by a ReLU as a collection of units (the kernel weights followed by the ReLU).

18.1 ENVIRONMENTS AND OPEN SOURCE CODE

All but the simplest neural networks require an extraordinary amount of housekeeping. Building an autoencoder would need: efficient convolution code; correct evaluation of gradients (which are surprisingly easy to get wrong); housekeeping for backpropagation; various code to implement an optimizer and scheduling; and code to monitor the learning process. All this can (and should!) be done for you

by an environment.

Good software environments manage an important practical difficulty by mediating interactions with accelerator hardware. A very high component of the computation workload for autoencoders (and neural networks generally) is, essentially, linear algebra. It is a remarkable fact that a *GPU* (or, occasionally, graphics processing unit) supports the operations required. GPUs were originally designed to support very fast rendering for high speed computer gaming. There are now several software environments that support the necessary housekeeping to map a network onto a GPU, evaluate the network and its gradients on the GPU, train the network by updating parameters, and so on. The easy availability of these environments has been an important factor in the widespread adoption of neural networks.

At time of writing, environments include:

- **Deep Learning Toolbox**, a Matlab toolbox for deep learning, at `https://www.mathworks.com/help/deeplearning/index.html?s_tid=CRUX_lftnav`.
- **MLX** `https://github.com/ml-explore/mlx`
- **PaddlePaddle:** This is an environment developed at Baidu research. You can find it at `http://www.paddlepaddle.org`. If you arrive at that page and find it in Chinese, note that there is a button you can click that gives an English version. It is also available at `https://github.com/paddlepaddle/paddle`. There is tutorial material on each page.
- **PyTorch:** This is an environment developed at Facebook's AI research. You can find it at `https://pytorch.org`. There are video tutorials at `https://pytorch.org/tutorials/`. The environment is very widely adopted.
- **Tensorflow:** This is an environment developed at Google. You can find it at `https://www.tensorflow.org`. There is extensive tutorial material at `https://www.tensorflow.org/tutorials/`.
- **Keras:** This is an environment developed by François Chollet, intended to offer high-level abstractions independent of what underlying computational framework is used. It is supported by the TensorFlow core library. You can find it at `https://keras.io`. There is tutorial material at that URL.

Each of these environments has their own community of developers. But these aren't the only environments. You can find a useful comparison at `https://en.wikipedia.org/wiki/Comparison_of_deep-learning_software` that describes many other environments.

Earlier environments you may encounter, but which appear to be no longer used include:

- **Matconvnet:** This is an environment for MATLAB users, originally written by Andrea Vedaldi and supported by a community of developers. You can find it at `http://www.vlfeat.org/matconvnet`. There is a tutorial at that URL.

- **Darknet:** This is an open source environment developed by Joe Redmon. You can find it at `https://pjreddie.com/darknet/`. There is some tutorial material there.
- **MXNet:** This is a software framework from Apache that is supported on a number of public cloud providers, including Amazon Web Services and Microsoft Azure. It can be invoked from a number of environments, including R and MATLAB). You can find it at `https://mxnet.apache.org`.

> **Remember this:** *Almost everyone who builds or uses neural networks uses an environment, API or software framework to handle the prodigious quantities of housekeeping required. There are a number of such environments. Choose whatever makes you happy. Keep in mind that many researchers and companies publish source code, weights and data. Using this is a good way to get started.*

18.2 TRICKS FOR NUISANCES

Input scale: The mechanics of learning can get difficult if large numbers appear in any tensors. Typically, the optimization either descends slowly or even diverges. Large numbers can cause a variety of problems. One is precision: subtracting two large but similar numbers and getting the answer right requires a lot of bits, because the answer will be small. Another is gradient scale: very large values in the gradient can make it hard to choose a learning rate that is small enough to get the learning process to converge. You will find that scaling the input image so that it occupies the range $[-1, 1]$ can significantly improve practical learning (**exercises**).

Color images: All the remarks of Section 7.1.1 apply here. You can choose the color representation you use for both input and output when you train an autoencoder to denoise color images. You should choose a color representation that is as decorrelated as possible, so: decompose into LAB; use sophisticated denoising on L; and use a heavily smoothed version of A and B.

Output scale: Images have values in a fixed range, typically either $[0, 255/256]$ or $[0, 255]$. What comes out of a convolutional layer does not. There are a variety of ways to deal with this problem. You could apply a function that maps the output to the range you want. For example,

$$\text{sigmoid}(x) = \frac{e^x}{1 + e^x}$$

will map any x to the range $(0, 1)$. There are alternative notations for the sigmoid (**exercises**). Another example is $\tanh(x)$, which will map any x to the range $(-1, 1)$ (and getting from there to $(0, 1)$ is easy). This approach has difficulties, however. Assume the output needs to be close to 1. Then for either of these mappings, the value of x will need to be large. Worse, changing the output to make

it slightly larger will require a very large change in x, meaning the gradient will be very small which can create problems in training.

You might think that using

$$f(x) = \text{ReLU}(x) - \text{ReLU}(x-1)$$

would be a good idea (because it maps any x to the range $[0, 1]$). In fact, it is a terrible idea, because once the output is outside that range, there is no gradient to push it back into the range (Section 17.4.1; **exercises**).

A second strategy is to accept the output of the convolutional layer, but penalize values that appear outside the range you want with a loss term. For example, apply the loss

$$L_{ud}(x) = x^2 \mathbb{I}_{[x<0]} + (x-1)^2 \mathbb{I}_{[x>1]}.$$

It is often useful to mix these strategies. For example, a final layer that applies

$$f(x) = a(\tanh(x) + b)$$

will map x to the range $(a(b-1), a(b+1))$. If you choose $b < 1$ and $a > 1/(b+1)$, then the output can be below zero (but not much) and above one (but not much). Further, the gradients won't be too small at zero or one. You can then push the outputs to be in that range with a penalty term.

Remember this: *The scaling of inputs to a neural network can be important. You can control the scaling of the outputs by a mapping that ensures the output is in the appropriate range. Doing so can create gradient problems, and an alternative is to use a variety of losses to discourage unacceptable values.*

18.3 MANAGING THE CONSEQUENCES OF DEPTH

An autoencoder built according to the recipes above and trained with enough data will perform tolerably, but you will notice some annoying effects. If there are relatively few layers in the encoder and the decoder, the reconstructions will tend to be quite sharp, but the autoencoder will not be very good at dealing with more than light noise. This is because the receptive fields of elements in the encoder must be quite small, and so the codes do not see many input pixels.

There will usually be many layers between the input and the output of a denoising autoencoder. Just stacking convolutional layers will mean that any information about a pixel must work its way through all the layers. This makes it very difficult for the autoencoder to recover very fine scale information. Skip connections – which feed information from early encoder layers to early decoder layers – can help, at the risk of possible cheating by the autoencoder (Section 18.3.1). If there are many layers in the encoder and decoder, some components of the gradient will

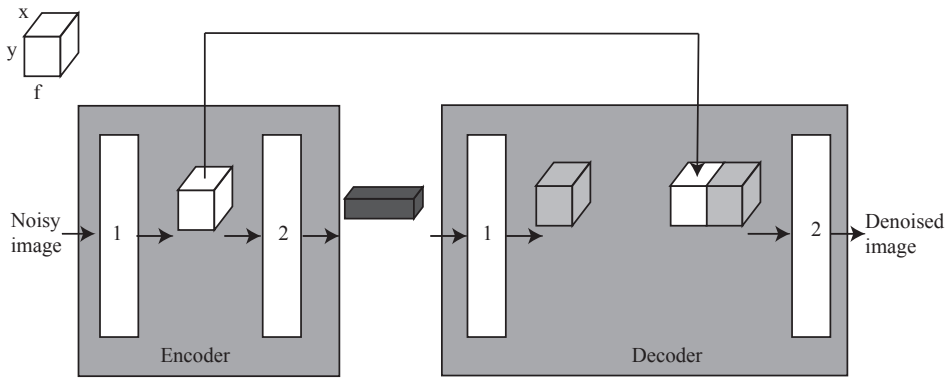

FIGURE 18.1: *In a very simple two layer autoencoder with skip connections, the first encoder layer produces a tensor of features (white block; the inset shows the dimensions, two spatial and one feature) which is passed to the second layer. The second layer downsamples to a smaller, deeper tensor (dark gray block). The decoder receives this, passes it through the first decoder layer and upsamples. Arrange the number of filters, etc. to ensure that this tensor has the same spatial extent as the white block (pale gray block). The white block and gray block are stacked along the feature dimension and passed through the second decoder layer. The second decoder layer needs to have more filters, but can now see both short scale features (the white block) and longer scale features (from the dark gray block).*

have passed through many poorly known layers, and so will be wrong. In turn, the autoencoder might not train satisfactorily. Residual blocks (Section 18.3.2) cure this difficulty. Finally, if there are enough layers in the autoencoder, some large values might appear in some features and cause unstable gradients. Batch normalization (Section **??**) will resolve this.

18.3.1 Skip Connections

An important difficulty presented by stacking many convolutional layers is that the features in the final code necessarily depend on a fairly large receptive field. This can make it difficult to produce reconstructions with sharp edges. A feature that depends on a very small neighborhood could provide enough information to place an edge accurately – for example, report the gradient of the image. If the receptive field is large, constructing a very local feature that isn't somewhat smoothed will require a set of weights that ignores many or most of the pixel values in the receptive field, which will be difficult to achieve. However, features with large receptive fields may be necessary to denoise, because they can observe long-range trends in the image.

This argument suggests that the image code should consist of some features that come from small receptive fields and some that come from large receptive fields. There is an analogy with a Laplacian pyramid here. In that pyramid, features that come from large receptive fields are used early in the reconstruction process and

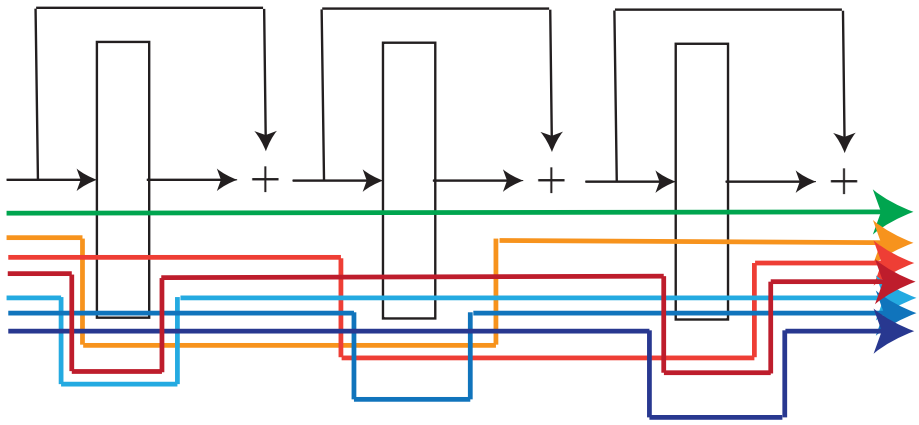

FIGURE 18.2: *Three simple residual layers, drawn in* **black**. *The output is the sum of a term that passes through all layers (***green***); three terms that each pass through two layers (***red***); and three terms that pass through one layer each.*

features that come from small receptive fields are used later. This suggests that the decoder should use features from large receptive fields at early stages, and use those from small receptive fields at later stages.

This is easy to achieve. Pass the tensor of features that comes from the first layer of the encoder to both the next layer of the encoder *and* the last layer of the decoder (Figure 18.1). This trick works with the second layer of the encoder and next to last layer of the decoder as well, etc.

18.3.2 Residual Connections

Imagine training many layers stacked on one another. Look at Section 17.3.4, and notice that the gradient of the first layer's parameters depends on a stack of derivative matrices connecting the output of the last layer to the result of the first. But the derivative matrices may not be particularly helpful, because the parameters for each layer are wrong (which is why we are training). In turn, the gradient of the first layer's parameters may not be helpful. Now think about the last layer. There are no intervening layers, so problems with derivative matrices don't apply. But the derivative is evaluated with that layer applied to a particular set of input values, and depends on these values. These values are wrong, because the previous layers are wrong, so the gradient update at the earliest layer is going to be poor as well. The argument applies to early and late layers, rather than just first and last. If there are few layers, this effect seems not to prevent training. But a deep stack of layers may be very hard to train. Gradient steps may diverge, or require an impractically small learning rate – and so a very large number of steps – to converge.

Residual connections are an extraordinarily powerful method to improve training behavior for very deep convolutional networks. The idea is straightforward. If you can ensure that some information passes through relatively few layers from

input to output, those layers will tend to be easier to train. A clever procedure ensures that there are components of the output that are determined by any subset of the layers.

Start with a simple example. Assume that each layer is a convolutional layer (no ReLUs), and that the input feature tensor is the same size as the output feature tensor. Recall Section 17.3.4 wrote the w'th layer as $\mathcal{L}_w$. Now write $\mathcal{B}$ for some tensor, and $\mathcal{R}_w = \mathcal{L}_w + \mathcal{I}$ (where $\mathcal{I}$ is the identity, so $\mathcal{R}_w(\mathcal{B}) = \mathcal{L}_w(\mathcal{B}) + \mathcal{B}$). Now stack these to get

$$\begin{aligned}
\mathcal{B}_4 &= \mathcal{R}_3(\mathcal{B}_3;\theta_3) \\
\mathcal{B}_3 &= \mathcal{R}_2(\mathcal{B}_2;\theta_2) \\
\mathcal{B}_2 &= \mathcal{R}_1(\mathcal{B}_1;\theta_1) \\
\mathcal{B}_1 &= \mathcal{I}.
\end{aligned}$$

Expand all this to get

$$\begin{aligned}
\mathcal{B}_4 &= \mathcal{L}_3(\mathcal{L}_2(\mathcal{L}_1(\mathcal{B}_1;\theta_1);\theta_2);\theta_3) + \\
&\quad \mathcal{L}_3(\mathcal{L}_2(\mathcal{B}_1;\theta_2);\theta_3) + \mathcal{L}_3(\mathcal{L}_1(\mathcal{B}_1;\theta_1);\theta_3) + \mathcal{L}_2(\mathcal{L}_1(\mathcal{B}_1;\theta_1);\theta_2) + \\
&\quad \mathcal{L}_3(\mathcal{B}_1;\theta_3) + \mathcal{L}_2(\mathcal{B}_1;\theta_1) + \mathcal{L}_1(\mathcal{B}_1;\theta_1) + \mathcal{B}_1 \\
\mathcal{B}_3 &= \mathcal{L}_2(\mathcal{L}_1(\mathcal{B}_1;\theta_1);\theta_2) + \\
&\quad \mathcal{L}_2(\mathcal{B}_1;\theta_1) + \mathcal{L}_1(\mathcal{B}_1;\theta_1) + \mathcal{B}_1 \\
\mathcal{B}_2 &= \mathcal{L}_1(\mathcal{B}_1;\theta_1) + \mathcal{B}_1 \\
\mathcal{B}_1 &= \mathcal{I} \text{ which is the input image.}
\end{aligned}$$

Notice the output is a series, consisting of a term that went through all three layers; three terms each involving two layers; three terms involving only one layer; and the input (Figure 18.2). The gradient will be the gradient of the series, and so will have terms that have passed through no layers, one layer, two layers and three layers. In the early stages of learning, there should be some improvement in each layer, because the gradient of the term that passes through that layer alone is quite accurate. As learning proceeds, these improvements should cause layers to make sensible reports to the next layer, meaning that terms that pass through two layers will be good, too.

Even though real layers involve non-linearities (at least, a ReLU here and there), this linear model is informative. This is because you could linearize the network about the current operating point, and the gradient of that linearized network is the true gradient at that operating point. Real layers tend also to increase or decrease the feature dimension of the tensor. You can deal with this by writing $\mathcal{V}_w$ for a slightly modified version of the identity operator, modified so that all dimensions of $\mathcal{V}_w(B)$ are the same as all dimensions of $L_w(B;\theta_w)$ *and* $\mathcal{V}_w$ is "very like" the identity.

Here is one way to build $\mathcal{V}_w$. If the tensor that comes out of L_w has the same size as the one that goes in, then $\mathcal{V}_w$ is the identity operator. If L_w makes the feature dimension of the tensor get smaller, then $\mathcal{V}_w$ projects off some dimensions of the input to match. If L_w makes the feature dimension of the tensor get bigger, then $\mathcal{V}_w$ pads the input with zeros to match dimensions. Finally, if L_w subsamples

the tensor so that its spatial dimensions get smaller, so does $\mathcal{V}_w$. You could then produce an $\mathcal{R}_w = \mathcal{L}_w(\cdot;\theta_w) + \mathcal{V}_w$. There is strong evidence in practice that a construction like this – known as a *residual layer* – helps learn very deep networks, even when there are ReLU's in the layers, and even when the dimension of a tensor changes when it passes through $\mathcal{L}_w$. In practice, these improvements in learning manifest when V_w isn't the identity; very often, V_w is a simple convolutional layer, as in Figure 18.3. Most modern convolutional networks are built out of residual layers of one form or another. Such networks are known as *residual networks* or *resnets*.

18.3.3 Batch Normalization

Numbers with large magnitude in a neural network cause problems. Imagine some input to some unit is big and the weight applied to that input is small. Then a single gradient step could cause the weight to change sign, and the ReLU might cause the corresponding output to swing between strongly positive and zero. This can cause training problems, because the gradient will be a poor predictor of what will actually happen to the output. Ideally, relatively few values at the input of any layer will have large absolute values. A new layer, sometimes called a *batch normalization layer*, can be inserted between two existing layers to ensure this happens.

Write $\mathcal{I}$ for the input of this layer, which is a $X \times Y \times F$ tensor of features, and $\mathcal{O}$ for its output, which is a tensor of features of the same dimension. The layer has two vectors of parameters, γ and β, each of dimension F. Write γ_i for the i'th component of γ, etc. Assume we know the mean (m_k) and standard deviation (s_k) of each feature in $\mathcal{I}$ computed over the whole dataset and over the spatial dimensions. Write ϵ for a small positive number chosen to avoid divide-by-zero. The tensor $\mathcal{U}$, with ijk'th component

$$\mathcal{U}_{ijk} = \frac{(\mathcal{I}_{ijk} - m_k)}{(s_k + \epsilon)}$$

will tend to have small magnitude numbers in it, both positive and negative. The mean of each feature in this tensor should be about zero, because it is close to the mean over all tensors. The standard deviation of each feature in this tensor should be about one, because it is close to the standard deviation over all tensors. Now compute

$$\mathcal{O}_{ijk} = \gamma_k \mathcal{U}_{ijk} + \beta_k$$

and notice that $\mathcal{O}$ could be the same as $\mathcal{I}$ (set $\gamma_k = s_k$ and $\beta_k = m_k$). The output of this layer is a differentiable function of γ and β, which can be adjusted to achieve the best performance.

Neither the mean or standard deviation are known, because the parameters of the previous layers are unknown. To estimate them, start with mean 0 and standard deviation 1 for each feature layer. Now choose a batch, and train the network using that batch. Once you have taken enough gradient steps and are ready to work on another minibatch, reestimate the mean as the mean of values of the inputs to the layer, and the standard deviation as the corresponding standard deviations. Now

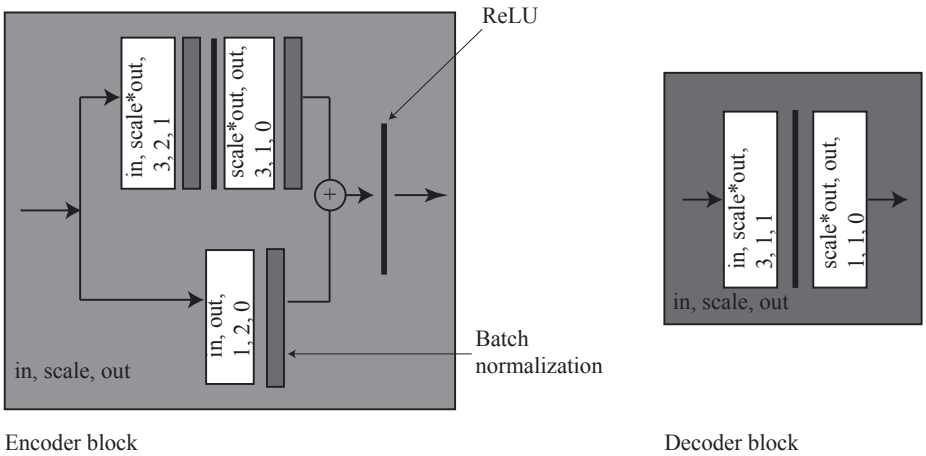

FIGURE 18.3: *I built the autoencoders in the worked examples of Sections 18.4.4 out of two, fairly standard blocks. On the* **left***, an encoder block, with arguments: input dimension; scale; and output dimension. This is a residual block built out of convolutional blocks (light rectangles: arguments are: input dimension; output dimension; kernel size; stride and padding); batch normalization blocks (darker rectangles); ReLU layers (vertical lines). The residual connection goes through a single convolutional layer. In some examples, the stride in this block will be 1 rather than 2. On the* **right***, a decoder block, with arguments: input dimension; scale; and output dimension.*

obtain another batch, and proceed. Remember, γ and β are parameters that are trained, just like the others.

Once the network has been trained, take the mean (resp. standard deviation) of the layer inputs over the training data for mean (resp. standard deviation). Most neural network implementation environments will do all the work for you. **Warning:** A reliable source of errors in using an API is not to "tell" the environment that you want a network with batch normalization layers in it to switch from training to evaluation mode – be careful about this.

Remember this: *Modern convolutional encoders and decoders tend to be very deep. This presents practical problems. Skip connections help decoder outputs maintain fine details. Residual layers improve training by making gradient estimates more useful. Batch normalization helps manage problems caused by large numbers in tensors. Skip connections, residual layers and batch normalization or variants of these constructions are present in most modern convolutional networks.*

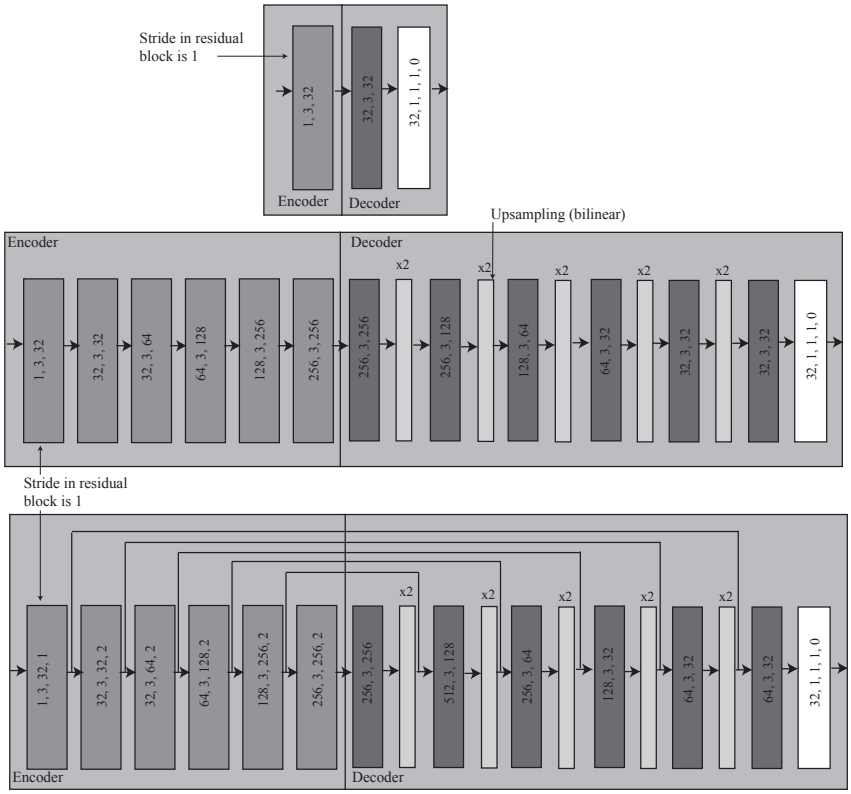

FIGURE 18.4: *The worked example compares three simple convolutional autoencoders, pictured here (small, no-skip and skip, Section 18.4). They are built out of the blocks in Figure 18.3. The skip connections shown simply stack the output of the connected encoder block on the output of the relevant decoder block, which is why the input dimensions are so big. The no-skip comparison version omits the skip connections and uses decoder blocks with smaller input dimensions.*

18.4 A WORKING DENOISING AUTOENCODER

Figure 18.4 sketches three autoencoders, built out of simple blocks shown in Figure 18.3. There is a very shallow autoencoder. An alternative is much deeper, but has no skip connections; and the reference version as deep and has skip connections. These autoencoders, which are quite shallow by current practice, are trained on 500, 000 distinct images of size 128×128. Training batches are selected uniformly and at random from this set.

Figure 18.5 shows training information for two of these autoencoders trained with SGD, trained for five rounds of about 100,000 images and with a simple learning rate schedule. Rates are $[2e-3, 1e-3, 7e-4, 3e-4, 1e-4]$. Write $\mathcal{R}$ for

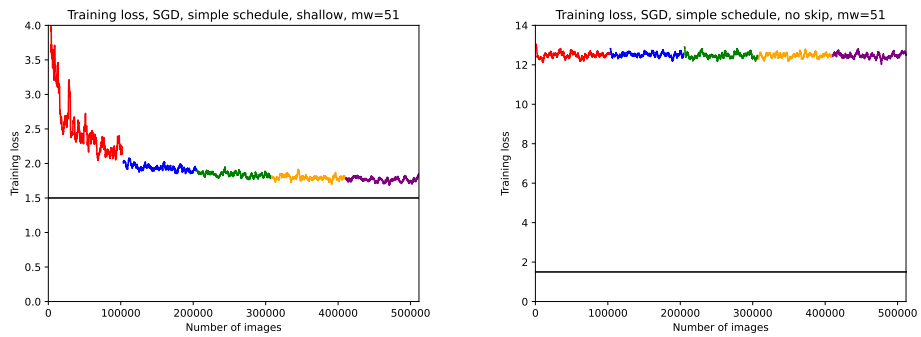

FIGURE 18.5: *The progress of training for a learned network is often plotted in a* learning curve*, a plot of training loss against some measure of progress (these figures use the number of images seen in training). On the* **left***, training for the shallow autoencoder and on the* **right***, for the autoencoder without skip connections. The learning rate changes approximately every 100, 000 images (the color changes), and the curve has been smoothed – you see a moving average of 50 records. Note the shallow autoencoder appears to be doing well, and the autoencoder without skip connections is hardly training. The horizontal line at 1.5 is to make it easier to compare curves.*

residual image; then the loss plotted is

$$100\frac{\sum_{ij} R_{ij}^2}{128 \times 128} + 30\frac{\sum_{ij} |R_{ij}|}{128 \times 128}$$

so if the root mean squared value of $\mathcal{R}$ is 0.1 – which is quite a large error in denoising an image – then the loss value will be about 4. Figure 18.6 (**left**) shows the same information for the autoencoder with skip connections. Comparing this to the curve for the shallow autoencoder suggests (misleadingly, as you will see) that there is no major benefit to a deep autoencoder. Worryingly, Figure 18.5 suggests the autoencoder with no skip connections appears not to be training, and Figure 18.6 suggests that a learning rate schedule could be quite important. Large scale training might require optimization improvements.

18.4.1 The Gradient is a Poor Choice of Direction

Everyone is surprised the first time they learn that the best direction to travel in when you want to minimize a function is not, in fact, backwards down the gradient. The gradient *is* uphill, but repeated downhill steps along the gradient are often not particularly efficient. I will show this very important point in several ways because different people have different ways of understanding this point.

Here is an example in algebra. Consider $f(x, y) = (1/2)(\epsilon x^2 + y^2)$, where ϵ is a small positive number. The gradient at (x, y) is $(\epsilon x, y)$. For simplicity, use a

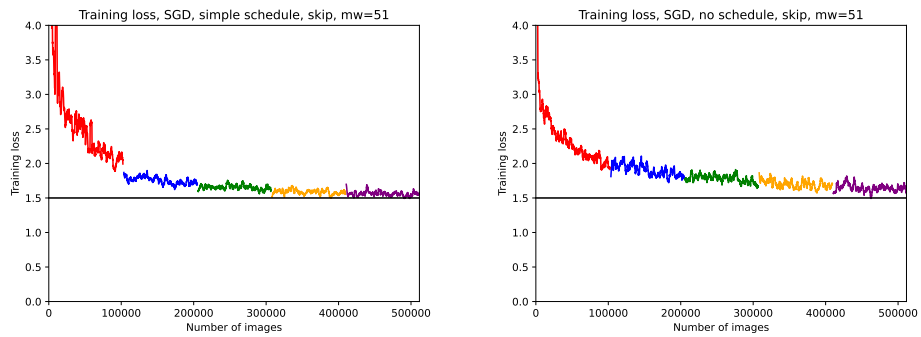

FIGURE 18.6: *Learning rate schedules are helpful for SGD. Learning curves for the autoencoder with skip connections using the learning rate schedule in the text* **left**, *and without a learning rate schedule on the* **right**. *The learning rate changes approximately every 100, 000 images (the color changes), and the curve has been smoothed – you see a moving average of 50 records. I have preserved the color changes when there is no learning rate schedule, to make it easier to compare curves. Notice that the training loss reached after 500, 000 images is quite a lot lower when the learning rate is scheduled. Notice also that the lowest training loss reached for the autoencoder with skip connections is rather a lot better that the best training loss for the other two (compare Figure 18.5), whatever the algorithm.*

fixed learning rate η, so

$$\begin{bmatrix} x^{(r)} \\ y^{(r)} \end{bmatrix} = \begin{bmatrix} (1-\epsilon\eta)x^{(r-1)} \\ (1-\eta)y^{(r-1)} \end{bmatrix}.$$

Start at, say, $(x^{(0)}, y^{(0)})$ and repeatedly go downhill along the gradient; you will travel very slowly to your destination. You can show that

$$\begin{bmatrix} x^{(r)} \\ y^{(r)} \end{bmatrix} = \begin{bmatrix} (1-\epsilon\eta)^r x^{(0)} \\ (1-\eta)^r y^{(0)} \end{bmatrix}.$$

The problem is that the gradient in y is quite large (so y must change quickly) and the gradient in x is small (so x changes slowly). In turn, for steps in y to converge requires $|1-\eta| < 1$; but for steps in x to converge requires only the much weaker constraint $|1-\epsilon\eta| < 1$. Choose the largest η you dare for the y constraint. The y value will very quickly have small magnitude, though its sign will change with each step. But the x steps will move closer to the right spot only extremely slowly.

Another way to see this problem is to reason geometrically. Figure 18.7 shows this effect for this function. The gradient is at right angles to the level curves of the function. But when the level curves form a narrow valley, the gradient points across the valley rather than down it. The effect isn't changed by rotating and translating the function (Figure **??**).

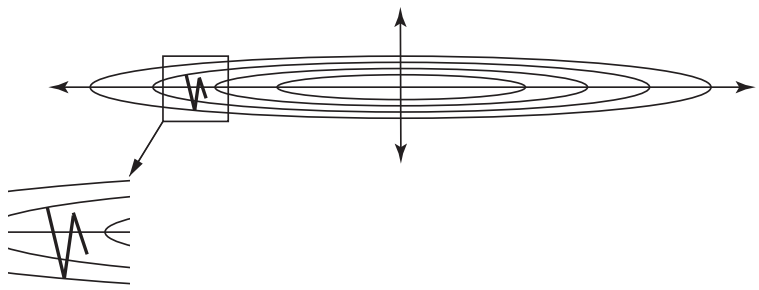

FIGURE 18.7: *A plot of the level curves (curves of constant value) of the function $f(x, y) = (1/2)(\epsilon x^2 + y^2)$. Notice that the value changes slowly with large changes in x, and quickly with small changes in y. The gradient points mostly toward the x-axis; this means that gradient descent is a slow zig-zag across the "valley" of the function, as illustrated. It might be possible to fix this problem by changing coordinates, but doing so would require second derivative information because the problem is caused by the directional derivative in some directions being big and in other directions being small.*

18.4.2 Alternative Directions

You may have learned that Newton's method resolves this problem. This is all very well, but to apply Newton's method would require knowing the matrix of second partial derivatives. A network can easily have millions to billions of parameters, and there is no hope of working with matrices of these dimensions.

One useful insight into the problem is that fast changes in the gradient vector are worrying. For example, consider $f(x) = (1/2)(x^2 + y^2)$. Imagine you start far away from the origin. The gradient won't change much along reasonably sized steps. But now imagine yourself on one side of a valley like the function $f(x) = (1/2)(x^2 + \epsilon y^2)$ (Figure 18.7); as you move along the gradient, the gradient in the x direction gets smaller very quickly, then points back in the direction you came from. You are not justified in taking a large step in this direction, because if you do you will end up at a point with a very different gradient. Similarly, the gradient in the y direction is small, and stays small for quite large changes in y value. You would like to take a small step in the x direction and a large step in the y direction.

You can see that this is the impact of the second derivative of the function (which is what Newton's method is all about). But Newton's method isn't available. Ideally, a method travels further in directions where the gradient doesn't change much, and less far in directions where it changes a lot. There are several ways to achieve this.

Parameters need to be discouraged from "zig-zagging" as in the example above. In these examples, the problem is caused by components of the gradient changing sign from step to step. It is natural to try and smooth the gradient. A method called *momentum* forms a moving average of the gradient.

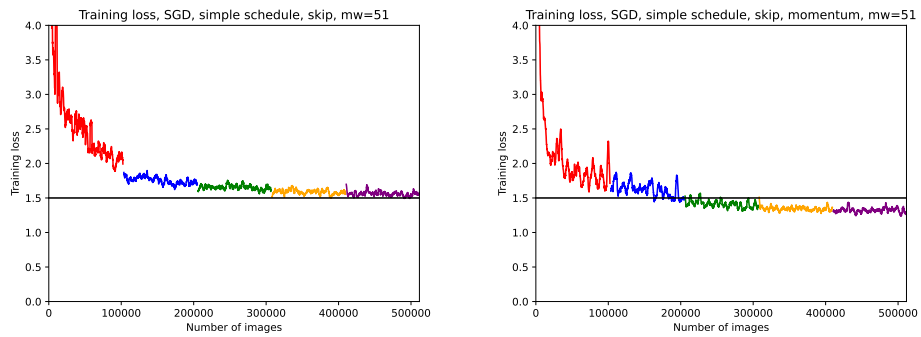

FIGURE 18.8: *Momentum can significantly improve training with SGD. Learning curves for the autoencoder with skip connections using the learning rate schedule in the text* **left**, *and with momentum ($\mu = 0.9$) and a learning rate schedule on the* **right**. *The curve has been smoothed – you see a moving average of 50 records. Note the improvements due to momentum.*

Procedure: 18.1 *Momentum*

Construct a vector $\mathbf{v}$, the same size as the gradient, and initialize this to zero. Choose a positive number $\mu < 1$ (the momentum). Then iterate

$$\begin{aligned} \mathbf{v}^{(r+1)} &= \mu \mathbf{v}^{(r)} + \nabla_\theta E \\ \theta^{(r+1)} &= \theta^{(r)} - \eta \mathbf{v}^{(r+1)} \end{aligned}$$

Larger μ means the final update sees a moving average of the gradient over a longer window.

Notice that, in this case, the update is an average of all past gradients, each weighted by a power of μ. If μ is small, then only relatively recent gradients will participate in the average, and there will be less smoothing. Larger μ lead to more smoothing. A typical value is $\mu = 0.9$. It is reasonable to make the learning rate go down with epoch when you use momentum, but keep in mind that a very large μ will mean you need to take several steps before the effect of a change in learning rate shows. Correctly implementing weight decay requires care when momentum is present (**exercises**).

A modification of momentum called *adam* rescales gradients, tries to forget large gradients, and adjusts early gradient estimates to correct for bias.

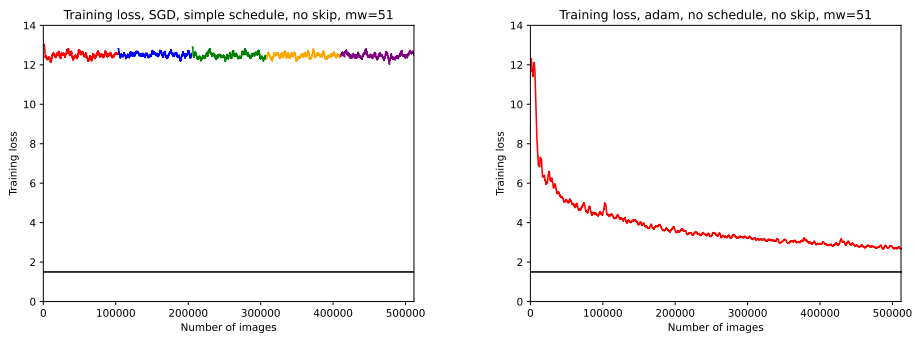

FIGURE 18.9: *Adam can improve training behavior for recalcitrant models. Learning curves for the autoencoder with no skip connections using the learning rate schedule in the text* **left**, *and with Adam on the* **right**. *The curve has been smoothed – you see a moving average of 50 records. This autoencoder just refused to train with SGD, and has trained with Adam, though the loss isn't wonderful.*

Procedure: 18.2 *Adam*

Write $g_i^{(r)}$ for the i'th component of the gradient $\nabla_\theta E$ computed at the r'th iteration. Choose three numbers β_1, β_2 and ϵ (typical values are 0.9, 0.999 and 1e-8, respectively), and some stepsize or learning rate η, then iterate

$$\begin{aligned}
\mathbf{v}^{(r+1)} &= \beta_1 * \mathbf{v}^{(r)} + (1-\beta_1) * \nabla_\theta E \\
c_i^{(r+1)} &= \beta_2 * c_i^{(r)} + (1-\beta_2) * (g_i^r)^2 \\
\hat{\mathbf{v}} &= \frac{\mathbf{v}^{(r+1)}}{1-\beta_1^t} \\
\hat{c}_i &= \frac{\hat{c}_i^{(r+1)}}{1-\beta_2^t} \\
\theta_i^{(r+1)} &= \theta_i^{(r)} - \eta \frac{\hat{v}_i}{\sqrt{\hat{c}_i} + \epsilon}
\end{aligned}$$

Remember this: *The gradient is seldom the best direction to follow. A variety of methods adjust the search direction to get better behavior. Most widely used is Adam, followed by momentum, but these are examples from a very large range of options (check your API for more).*

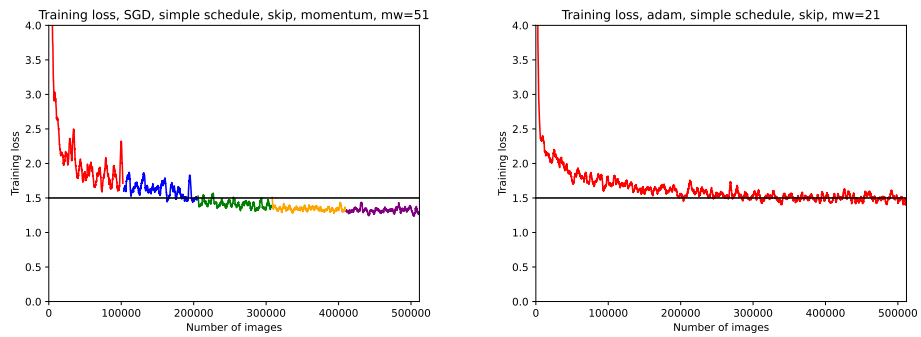

FIGURE 18.10: *Adam is often, but not always, the best available training procedure. Learning curves for the autoencoder with skip connections and momentum ($\mu = 0.9$)* **left**, *and with Adam on the* **right**. *The curve has been smoothed – you see a moving average of 50 records. Note that here SGD with a simple learning rate schedule is behaving better; but look at Figure [**?**].*

18.4.3 Hyperparameters: Weights, etc.

There are a variety of *hyperparameters* – values that need to be chosen before training, like choice of optimizer, initial learning rate, learning rate scheduling strategy, weight of SSD loss against L1 loss, and so on – that need to be set to train a model. Generally, the choice of hyperparameters for a complex model can be a difficult problem, requiring specialized machinery. Problems include: there may be many hyperparameters; there isn't much theoretical insight into what values they should take or how they might interact; and evaluating any particular set of hyperparameters is likely expensive (train the model, then evaluate it).

At this point, I will concentrate on rules of thumb. One widely used rule of thumb is that Adam is used to train models used to publish papers – because one gets fast descent, so you can fiddle with the results till the deadline – but either vanilla SGD or SGD with momentum is used to train production models – because one tends to get more robust models, but slowly. Another useful rule of thumb is that it is usually obvious when an initial learning rate is too large. The model will typically diverge quickly – you'll get nan's in the parameters and then the losses. If the learning rate is much too small, you will get no or minimal descent. In turn, you can set an initial learning rate quite easily by trying a value larger than you think will work (1e-2 has a following), then reducing it till the model does not diverge.

For the class of model described here, the learning rate scheduling in Section 17.3.1 is quite sufficient (though if your API provides others, it is always amusing to try them). The number of steps to take and the constant to scale the learning rate by are typically set by experiment. It is usually a good idea to pass through all training data at least once before scaling the learning rate. The scale is typically either 0.1 or 0.3 (roughly, the square root of 0.1).

For parameters like the relative weights of different losses, typically only quite large changes in value matter very much. However, choosing these parameters takes

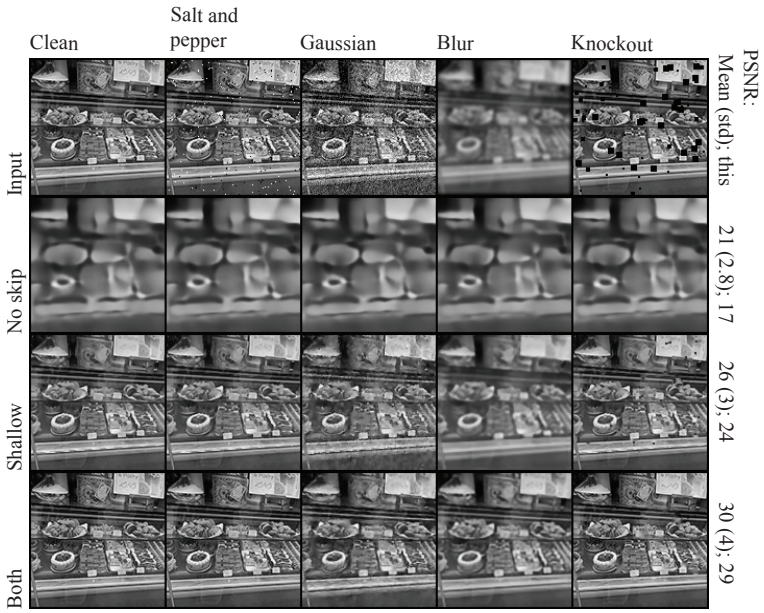

FIGURE 18.11: *Comparing autoencoder reconstructions for various kinds of noise and various kinds of autoencoder. Details in the text.* Image credit: *Images is my photograph of an enticing shop window.*

some care, because the intention is to produce the best behavior *on application data* rather than on training data. But the model will be slightly better on training data than on other data, because it has been trained to do well on training data. This means you must evaluate the model on data that wasn't used in training. It is natural to split the available data into a training set, a validation set, and a test set. Now train models on the training set for several different parameter values, and evaluate each on the validation set. Choose the model with the best performance on the validation set. Notice that you do not know how well this model works on application data, because it has been chosen to be good on the validation data (and so the estimate of how well it works is likely somewhat biased). Finally, evaluate this model by applying it to the test set. This recipe applies whatever the measure used for "goodness" of the model (choose from the cases in Section 7.1.2, for example).

18.4.4 Example Results

In training, each sees a total of 2M images. They are trained with a mixed L1/L2 loss. During training, noise is applied to input images using: additive Gaussian noise of randomly chosen magnitude; salt and pepper noise of fixed magnitude; gaussian blurring with a blur kernel of random $\sigma < 3$; and "knockout", where randomly chosen blocks of pixels of randomly chosen size up to 9×9 are set to zero.

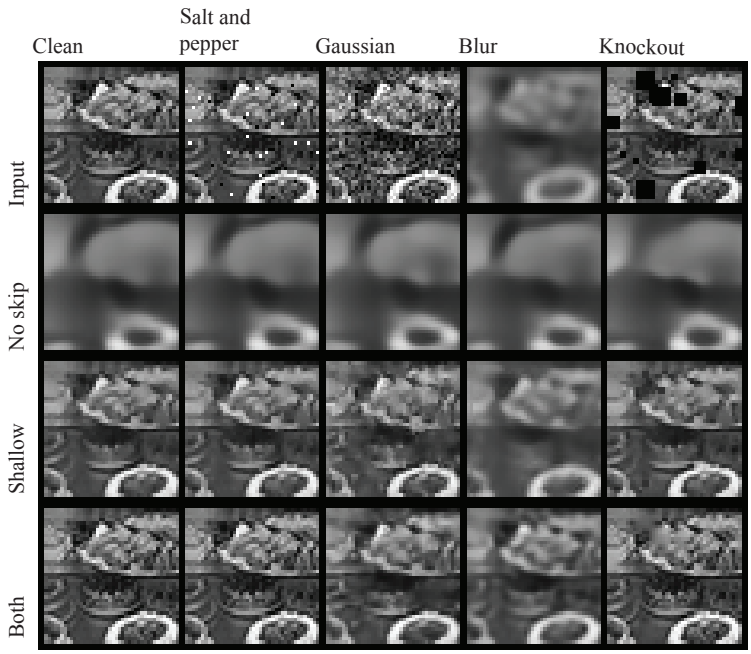

FIGURE 18.12: *Detail panels comparing autoencoder reconstructions for various kinds of noise and various kinds of autoencoder. Details in the text.* Image credit: *Images is my photograph of an enticing shop window.*

Skip connnections make a significant difference to autoencoder performance. Figures 18.11 and **??** compare the output of these autoencoders. Figure **??** shows a detail panel for each of the inputs and outputs of Figure 18.11.

The PSNR's are shown for a variety of cases. PSNR is computed by averaging over five outputs: one resulting from a clean version of the image, and one resulting from an input with an instance of each of the four types of noise. Figure 18.11 shows a PSNR computed like this for the example image and the mean and standard deviation of PSNR computed for 500 test images.

There are some fairly reliable conclusions that one can draw from this quite simple example. First, notice that autoencoders can produce learned representations of images that are highly effective at denoising. Second, the autoencoder without skip connections produces heavily blurred images: it is really difficult for fine spatial detail to make its way through all those layers. Third, the shallow (skip only) network does rather well for local noise effects (salt and pepper noise; additive Gaussian noise) but is less effective for effects that require a longer scale view of the image (deblurring; knockout). Comparing the shallow network to the deeper network with skip connections suggests strongly (and correctly) that deeper image descriptions tend to be better, as long as there is a mechanism like skip connections. Finally, notice that deblurring is hard.

Remember this: *A good denoising autoencoder needs to be deep; needs to have skip connections; and will be hard to train without residual connections. Learning rate schedules can help training, as can Adam and momentum. Use at least a mixed L1-L2 loss to control blurriness in the reconstruction. Most of these points will remain good for other uses of the encoder/decoder idea*

18.4.5 Standing on Others' Shoulders

Sir Isaac Newton's point remains good, even if no giants happen to be available (look it up!). A great virtue of the computer vision research community is that it is now common to publish code, networks and datasets openly. This means that, for much cutting edge research, you can easily find a code base that implements a network; and all the parameter values that the developers used to train a network; and a trained version of the network; and the dataset they used for training and evaluation. Companies often publish weights and models, too. There are a variety of online platforms that share models, code and datasets and provide reference releases. Platforms (and environments) are at various stages of monetization, but as of writing all had free options.

- **HuggingFace** serves as a repository of models, example code and documentation. You can find this at `https://huggingface.co/docs/hub/index`; there is a nice getting started page at `https://huggingface.co/docs/hub/repositories-getting-started`.
- **Github** is another repository of shared code, models and datasets, at `https://github.com`.

There is very good evidence that, if you have something that is trained to perform well on one task – say, a denoising autoencoder that specializes in inpainting moderately sized holes in images – you can modify it to perform sufficiently related tasks rather well, rather than training something from scratch. For this example, you would obtain an autoencoder (model, weights and all); an appropriate dataset (so pairs of blurred and clean images); and then you would apply descent on the new dataset, typically with quite small learning rates. If the two tasks are sufficiently related (which can be tricky to predict), this strategy – usually called *finetuning* – will produce a model that does well at your intended task with relatively little training. This makes it hugely useful to be able to download and use models other people made.

Remember this: *It is usual and helpful to obtain other peoples' models, and adjust them as required. This is a great way to learn new tricks.*

18.5 YOU SHOULD

18.5.1 remember these terms:

Use environments and open source code and data 337
The scale of inputs and outputs present important nuisances 338
Skip connections, residual layers and batch normalization are crucial tools for managing difficulties created by large numbers of layers 343
Gradient descent can be slow; there are many schemes for adjusting the direction you follow 349
Some general requirements for a well-behaved autencoder 353
Download and use published models 353

18.5.2 remember these facts:

Use environments and open source code and data 337
The scale of inputs and outputs present important nuisances 338
Skip connections, residual layers and batch normalization are crucial tools for managing difficulties created by large numbers of layers 343
Gradient descent can be slow; there are many schemes for adjusting the direction you follow 349
Some general requirements for a well-behaved autencoder 353
Download and use published models 353

18.5.3 remember these procedures:

Momentum 348
Adam 349

18.5.4 be able to:

- Put together a simple denoising autoencoder using your preferred API.
- Explain why residual layers, normalization and skip connections can be helpful.
- Choose an optimization procedure to train your autoencoder, and apply it.

EXERCISES

QUICK CHECKS

18.1. A sigmoid

$$\text{sigmoid}(x) = \frac{e^x}{1 + e^x}$$

maps $[-\infty, \infty]$ to $[0, 1]$. What is the derivative of a sigmoid? When sigmoid(x) is close to 1 or to 0, what can you say about the derivative? Why might this be a nuisance?

18.2. Write

$$f(x) = \text{ReLU}(x) - \text{ReLU}(x - 1).$$

Show $f(x) = 0$ for $x \leq 0$ and $f(x) = 1$ for $x \geq 1$. What is the gradient of $f(x)$ in these cases?

18.3. Section 18.3.2 has: "Even though real layers involve non-linearities (at least, a ReLU here and there), this linear model is informative. This is because you could linearize the network about the current operating point, and the gradient of that linearized network is the true gradient at that operating point." Explain – why can you get away with linearizing about an operating point when a ReLU isn't differentiable?

18.4. Section **??** has: " **Warning:** A reliable source of errors in using an API is not to "tell" the environment that you want a network with batch normalization layers in it to switch from training to evaluation mode – be careful about this." Explain.

18.5. Sophisticated learning rate schedules mostly decrease the learning rate, but sometimes *increase* it. Why might occasionally increasing the learning rate be a good idea?

PROGRAMMING EXERCISES

18.6. In this exercise, you will build a series of denoising autoencoders.

- **(a)** Build an example denoising autoencoder, using the version of Section 18.4 with skip connections as a guide.
- **(b)** How sensitive is denoising performance of this autoencoder to the difference between training noise and test noise? To check this, train with all four noise types of Figure 18.11, and test on deblurring. Now omit deblurring from the noise types when you train, and test again on deblurring. What conclusions can you draw?
- **(c)** Can you improve the autoencoder's performance by making it deeper?
- **(d)** Can you improve its performance by adjusting the losses?
- **(e)** Can you find a better performing autoencoder online?