CHAPTER 16

Making an Autoencoder that Works

16.1 ENVIRONMENTS AND APIS

It is just possible to build an autoencoder in a simple programming environment using the description of the previous chapter, but doing so will be very hard work and the result will work poorly even with a great deal of fiddling. I do not encourage the exercise – though it is informative, the knowledge will be hard-won. An autoencoder is an example of a neural network, a category containing a wide range of of enormously useful approximation procedures. 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.

A very high component of the computation workload for autoencoders (and neural networks generally) is, essentially, linear algebra. Fairly quickly it will become obvious that specialized hardware support is highly desirable. It is a remarkable fact that a *GPU* (or, occasionally, graphics processing unit) supports the operation 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.

It is now common in the research community 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`.

- **Papers with code**

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`.

16.2 NETWORK TRICKS

If you rushed off to build an autoencoder after reading Section 16.1, you likely encountered some practical problems.

16.2.1 Useful Practical Tricks

Input scale: The mechanics of learning can get difficult if large numbers appear in any data blocks. 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**).

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 15.3.1; **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.

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.

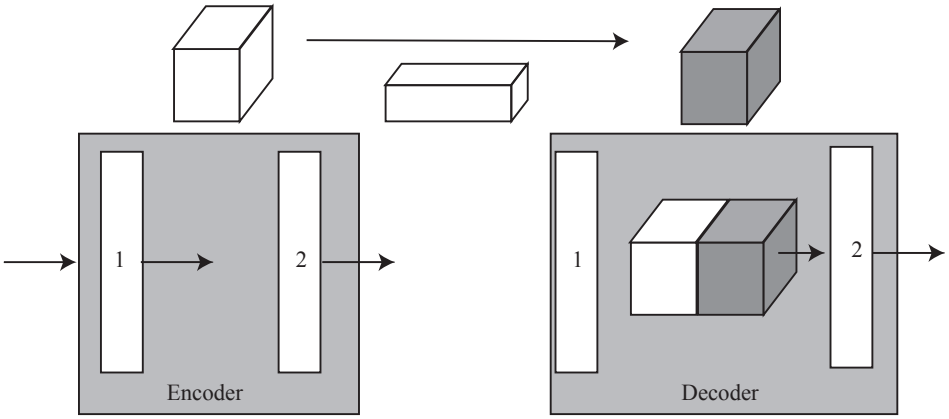

FIGURE 16.1: *In a very simple two layer autoencoder with skip connections, the first encoder layer produces a block of features (***above***, white block) which is passed to the second layer. The second layer downsamples to a smaller, deeper block (***above***, long white block). The decoder receives this, passes it through the first decoder layer and upsamples to obtain a block which has the same spatial extent as the white block (***above***, gray block). The white block and gray block are stacked 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 gray block).*

16.2.2 Skip Connections

An important difficulty presented by stacking many convolutional layers is that any feature produced by the encoder necessarily depends 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 necessar to denoise, because they can observe long-range trends in the image.

This argument suggests encoding the image with many blocks of features, each depending on different sized receptive fields. An easy way to do this is to pass the block 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 16.1). This trick works with the second layer of the encoder and next to last layer of the decoder as well, etc.

16.2.3 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 $X \times Y \times F$ block of features, and $\mathcal{O}$ for its output, which is a block 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 data block $\mathcal{U}$, 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 block should be about zero, because it is close to the mean over all blocks. The standard deviation of each feature in this block should be about one, because it is close to the standard deviation over all blocks. 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 minibatch, and train the network using that minibatch. 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 obtain another minibatch, 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. 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.

16.2.4 Residual Connections

An autoencoder built according to the recipes above and trained with enough data will perform tolerably, but you will notice some annoying effects. The autoencoders of Figure 15.4 and Figure 15.5 use relatively few convolutional layers by current standards. The encoder architecture is in Figure 16.3 and Figure 16.4. 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. If there are many layers in the encoder and decoder, the autoencoder might not train satisfactorily

Imagine training many layers stacked on one another. Look at Section 15.2.3, 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 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 will not 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, and most easily described in an abstract way. Recall Section 15.2.3 wrote the w'th layer as $L_w(\cdot;\theta_w)$. Now write V_w for a slightly modified version of the identity operator, modified so that all dimensions of $V_w(B)$ are the same as all dimensions of $L_w(B;\theta_w)$ *and* V_w is "very like" the identity.

Here is one way to build V_w. If the data block that comes out of L_w has the same size as the one that goes in, then V_w is the identity operator. If L_w makes the feature dimension of the data block get smaller, then V_w projects off some dimensions of the input to match. If L_w makes the feature dimension of the data block get bigger, then V_w pads the input with zeros to match dimensions. Finally, if L_w subsamples the data block so that its spatial dimensions get smaller, so does V_w.

Now write $R_w(\cdot;\theta_w) = V_w + L_w(\cdot;\theta_w)$. Imagine stacking three such layers to get

$$\begin{aligned} B_4 &= R_3(B_3;\theta_3) \\ B_3 &= R_2(B_2;\theta_2) \\ B_2 &= R_1(B_1;\theta_1) \\ B_1 &= \mathcal{I}. \end{aligned}$$

In the simplest case, where V_w happens to be the identity, this is

$$\begin{aligned} B_4 &= B_3 + L_3(B_3;\theta_3) \\ B_3 &= B_2 + L_2(B_2;\theta_2) \\ B_2 &= B_1 + L_1(B_1;\theta_1) \\ B_1 &= \mathcal{I} \end{aligned}$$

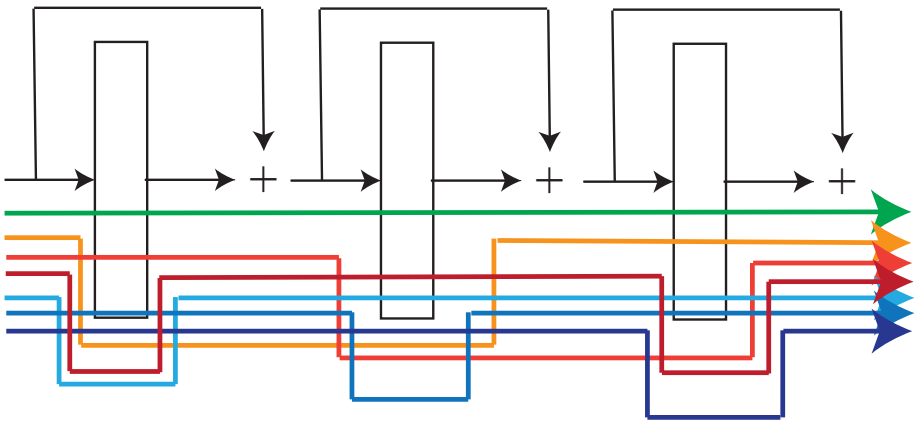

FIGURE 16.2: *Three simple residual layers, drawn in* **black**. *The output is the sum of a term that passes through all layers (***green***); three terms that each pass through two layers (***red***); and three terms that pass through one layer each.*

equivalently

$$\begin{aligned} B_4 &= \mathcal{I} + L_1(\mathcal{I};\theta_1) + L_2(B_2;\theta_2) + L_3(B_3;\theta_3) \\ B_3 &= \mathcal{I} + L_1(\mathcal{I};\theta_1) + L_2(B_2;\theta_2) \\ B_2 &= \mathcal{I} + L_1(\mathcal{I};\theta_1) \end{aligned}$$

so the output block consists of a series consisting of:

- a term that passes from the input to the output directly;
- a term that passes from the input to the output through one layer;
- a term that passes from the input to the output through two layers;
- and a term that passes through all three layers.

as Figure 16.2 shows.

This series reveals why the residual connections could help learning. 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. There is strong evidence in practice that residual layers help learn very deep networks. 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 16.3.

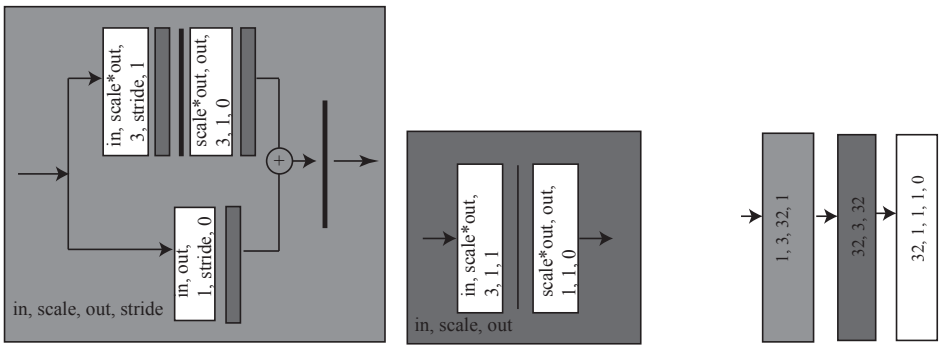

FIGURE 16.3: *Autoencoders in the worked examples of Sections 17.3.3 and 17.3.3 are built out of two, fairly standard blocks. On the* **left**, *an encoder block, with arguments: input dimension; scale; output dimension; and stride. This is built out of convolutional blocks (light rectangles: arguments are: input dimension; output dimension; kernel size; stride and padding); batch normalization blocks (darker rectangles); ReLU layers (vertical lines). There is a residual connection, going through a single convolutional layer. In the* **center**, *a decoder block, with arguments: input dimension; scale; and output dimension. On the* **right**, *a very simple autoencoder. The encoder is a single encoder block and the decoder is one decoder block followed by a convolutional layer.*

16.3 OPTIMIZATION AND TRAINING

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.

16.3.1 The Gradient as a Poor Choice of Direction: Algebra

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 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

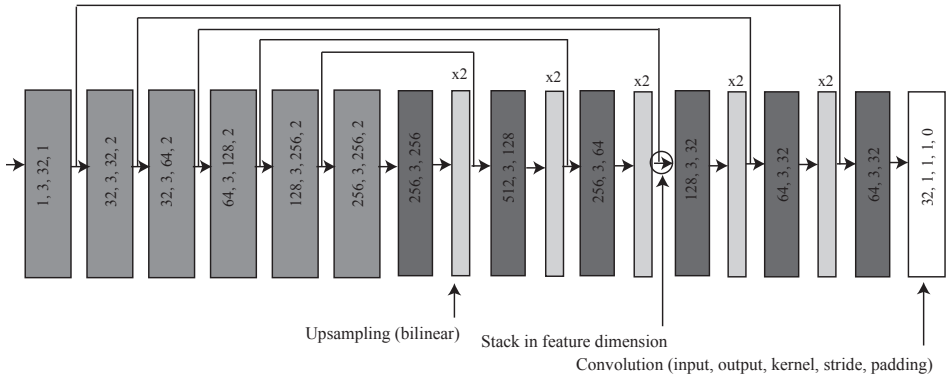

FIGURE 16.4: *The autoencoder with skip connections in the worked examples of Sections 17.3.3 and 17.3.3 is built out of a sequence of encoder blocks, followed by a sequence of decoder blocks (blocks in Figure 16.3). The skip connections are indicated by in the drawing. Note that the connection simply stacks 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.*

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.

16.3.2 The Gradient as a Poor Choice of Direction: Geometry

Another way to see this problem is to reason geometrically. Figure 16.5 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 16.6).

16.3.3 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 16.5); 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

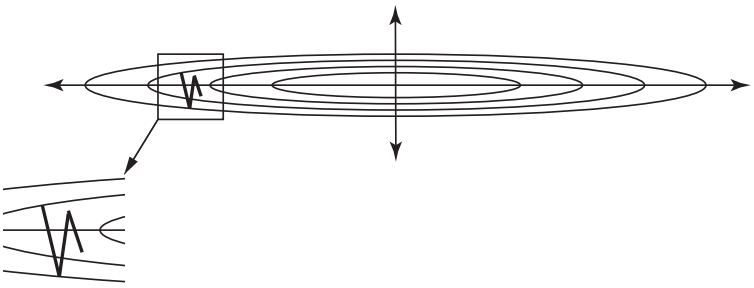

FIGURE 16.5: *A plot of the level curves (curves of constant value) of the function* $f(x, y) = (1/2)(\epsilon x^2 + y^2)$*. Notice that the value changes slowly with large changes in* x*, and quickly with small changes in* y*. The gradient points mostly toward the* x*-axis; this means that gradient 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.*

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.

16.3.4 Momentum

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. Momentum forms a moving average of the gradient. Construct a vector $\mathbf{v}$, the same size as the gradient, and initialize this to zero. Choose a positive number $\mu < 1$. Then iterate

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

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**).

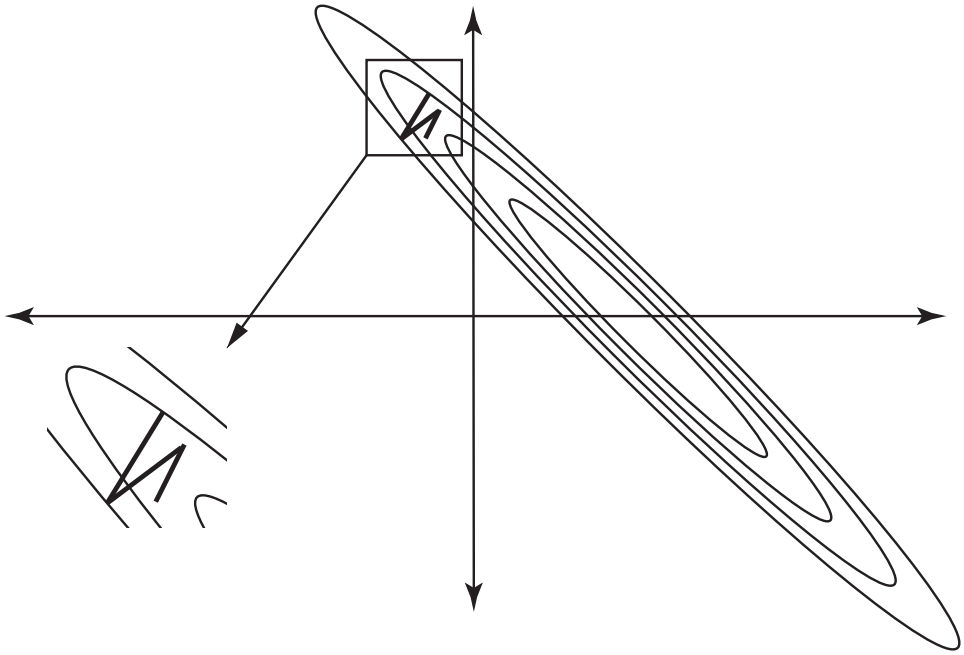

FIGURE 16.6: *Rotating and translating a function rotates and translates the gradient; this is a picture of the function of figure 16.5, but now rotated and translated. The problem of zig-zagging remains. This is important, because it means that choosing a good change of coordinates is likely very hard.*

16.3.5 Adagrad and RMSprop

This is a method to keep track of the size of each component of the gradient. In particular, there is a running cache $\mathbf{c}$ which is initialized at zero. Choose a small number α (typically 1e-6), and a fixed η. Write $g_i^{(r)}$ for the i'th component of the gradient $\nabla_\theta E$ computed at the r'th iteration.Then iterate

$$\begin{aligned} c_i^{(r+1)} &= c_i^{(r)} + (g_i^{(r)})^2 \\ \theta_i^{(r+1)} &= \theta_i^{(r)} - \eta \frac{g_i^{(r)}}{(c_i^{(r+1)})^{\frac{1}{2}} + \alpha} \end{aligned}$$

Notice that each component of the gradient has its own learning rate, set by the history of previous gradients.

RMSprop is a modification of Adagrad, to allow it to "forget" large gradients that occurred far in the past. Again, write $g_i^{(r)}$ for the i'th component of the gradient $\nabla_\theta E$ computed at the r'th iteration. Now choose another number, Δ,

(the *decay rate*; typical values might be 0.9, 0.99 or 0.999), and iterate

$$\begin{aligned} c_i^{(r+1)} &= \Delta c_i^{(r)} + (1-\Delta)(g_i^{(r)})^2 \\ \theta_i^{(r+1)} &= \theta_i^{(r)} - \eta \frac{g_i^{(r)}}{(c_i^{(r+1)})^{\frac{1}{2}} + \alpha} \end{aligned}$$

16.3.6 Adam

This is a modification of momentum that rescales gradients, tries to forget large gradients, and adjusts early gradient estimates to correct for bias. Again, 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

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

16.3.7 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 15.2.1 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 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).

16.4 WORKED EXAMPLE

Figures 16.3 and 16.4 sketch three autoencoders. The reference version, with skip connections, is shown in Figure 16.4; the caption for that figure describes a version without skip connections. Finally, Figure 16.3 is a shallow autoencoder. These autoencoders, which are quite shallow by current practice, are trained on 500, 000 distinct images of size 128×128. 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.

Skip connnections make a significant difference to autoencoder performance. Figures 16.8 and **??** compare the output of these autoencoders. Figure **??** shows a detail panel for each of the inputs and outputs of Figure 16.8.

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 16.8 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

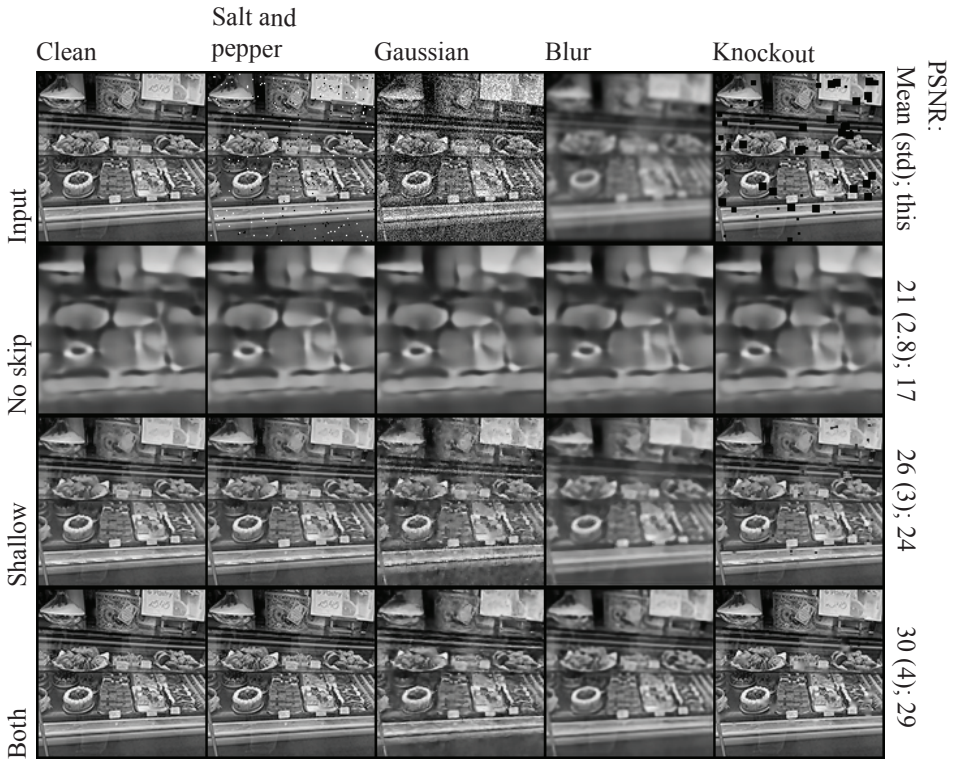

FIGURE 16.7: *Comparing autoencoder reconstructions for various kinds of noise and various kinds of autoencoder. Details in the text.* Image credit: *Images is my photograph of an enticing shop window.*

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.

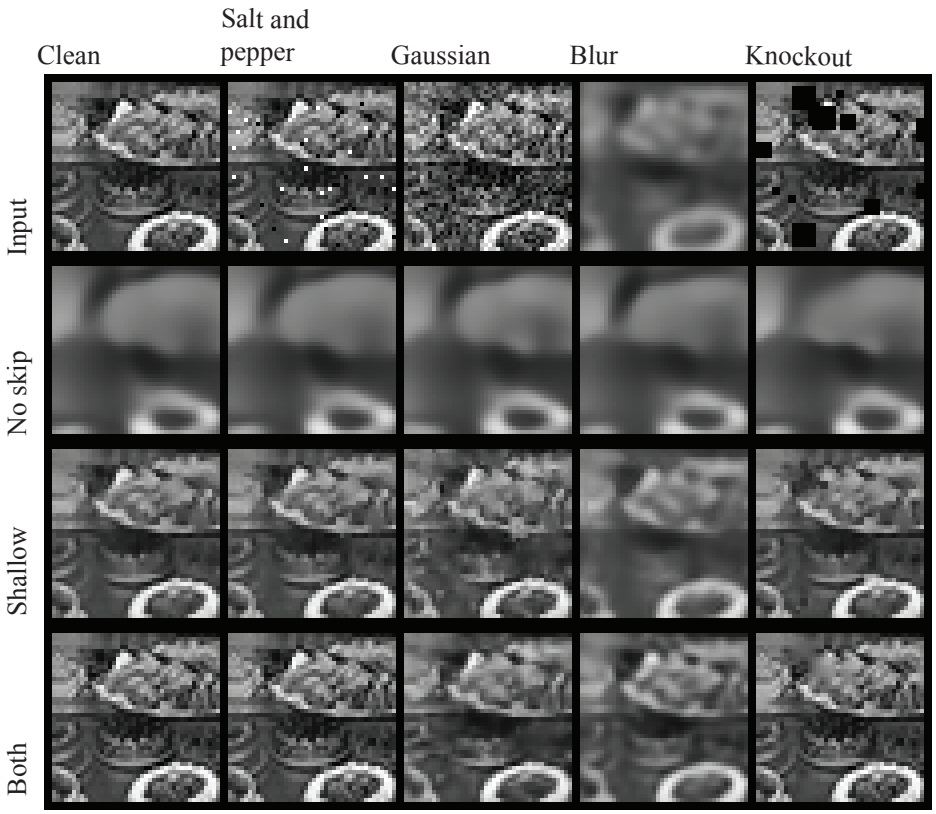

FIGURE 16.8: *Detail panels comparing autoencoder reconstructions for various kinds of noise and various kinds of autoencoder. Details in the text.* Image credit: *Images is my photograph of an enticing shop window.*