CHAPTER 17

# Forcing Images to be Realistic

There are strong links between the optimization based denoisers of Chapter 7 and the learned denoisers of Chapters 15 and 16. The optimization methods search for something that is (a) close to the noisy input and (b) more "like an image" than the noisy input. The learned methods try to predict the result of this search. The optimization based methods use quite complex measures of "like an image", where (so far) the learned methods measure "like" by some norm comparing the denoised and the original image in training.

A key nuisance here is that most arrays are not images, and true images seem to be relatively "rare", in the sense that you will need to sample an awful lot of arrays uniformly and at random to see an image. Worse, simple norms are not adapted to images. Something that is close to an image in an L2 norm, for example, may not be a real image.

Ideally, the denoiser should produce something that is (a) "like" the original version and (b) an image. A remarkable construction – adversarial loss – can be used to force the denoiser to produce objects that are hard to distinguish from images. This construction builds on the image representation built by a learned encoder to build a *classifier* – a device that can map an image to a category label, in this case "real" or "fake".

For the moment, assume you can build a good classifier. At a high level, the construction works as follows. Take a denoiser, which (for the moment) will be frozen (no parameters are allowed to change). Now build a classifier that can tell the difference between what comes out of the denoiser and real images. If this classifier is no better than flipping a coin, then there is nothing to do – what comes out of the denoiser is indistinguishable from a real image. This doesn't usually happen without work, so freeze the classifier, unfreeze the denoiser, and update the denoiser so it now fools the frozen classifier. Of course, the new denoiser is now likely to be producing objects that aren't images, but fool the classifier into believing that they are. So freeze the new denoiser, unfreeze the classifier, and adjust the classifier. Repeat as needed. This chapter fills in some of the details in this recipe.

## 17.1 CLASSIFICATION: IS THIS A REAL IMAGE?

The classifier must distinguish between two *classes* (sometimes, *categories*). One consists of actual images, and the other of what comes out of the denoiser. This section assumes that each image is represented by a known feature vector of fixed dimension (Section 17.2 treats how to obtain this feature vector). The classifier will accept this feature vector, then produce a number. Ideally, that number is positive for for any image that comes out of the denoiser, and negative for any that is a real image patch. Such a classifier could serve as a loss for training a new version of the denoiser, at least until the denoiser gets good enough to fool it.

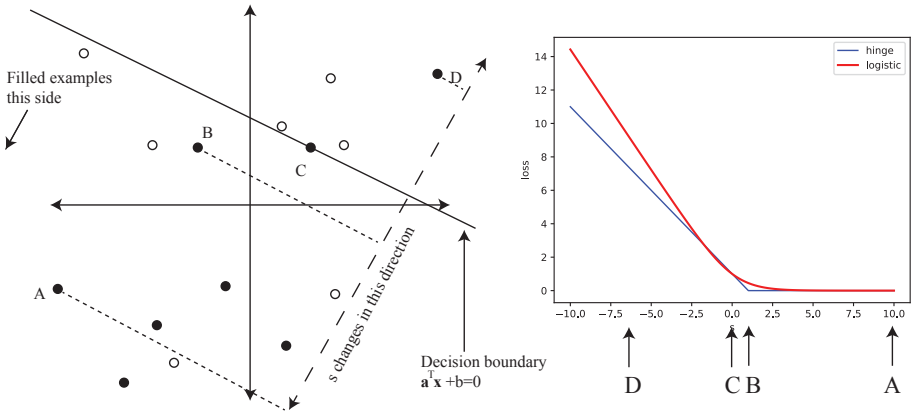


FIGURE 17.1: **Left:** *A visualization of a linear classifier in a 2D feature space (so $f = 2$) to illustrate the constraints on a classification loss. The example labelled D should have large loss, because it is on the wrong side of the boundary and far away from the boundary. The example labelled C should have a medium loss. It is on the boundary, but should be some way to the right side. This is because there are likely future examples close to it, and some of those might be on the wrong side of the boundary. The example labelled B should have a zero loss, because it is far enough on the right side of the boundary.* **Right:** *Plots of the hinge and cross entropy loss for filled examples, keyed to the example labels, to show how these losses meet the constraints.*

### 17.1.1 From Features to Label with a Linear Classifier

Assume you have a feature vector $\mathbf{x}$ that describes an image well. You must map this feature vector to a *label* which identifies the class of the image. In the current case, the label is either "real" or "denoised", but much richer alternatives will be important (Chapter **??**). A straightforward choice is a *linear classifier*, which maps $\mathbf{x}$ to $u(\mathbf{x}; \mathbf{a}, b) = (\mathbf{a}^T\mathbf{x} + b)$, then uses the sign of that value to classify. Equivalently, a linear classifier constructs a hyperplane in the feature space. Data items that map to one side of the hyperplane are real and data items that map to the other side are denoiser outputs. The parameters $\mathbf{a}$ and $b$ are chosen to get the best performance (many more details below). You might object that this mapping is too simple to achieve what is wanted. But the the feature vector is a high dimensional representation of the image, so there is a good chance of finding a linear classifier that separates the two. It will turn out that the feature vector is the product of a learned encoder, meaning *you can adjust the encoder to get the feature vector that works best with a linear classifier.*

The *training error rate* of the classifier is the fraction of training examples that it classifies incorrectly. The *training accuracy* of the classifier is the fraction of training examples that it classifies correctly. These are easy to estimate – take the current values of $\mathbf{a}$ and $b$, use them to classify a random subset of training examples, then compute what fraction are wrong (or right).

### 17.1.2 Logistic Regression

The next step is to choose the parameters of the linear classifier to get good behavior from the classifier. The recipe used in Chapter 15 applies: construct a loss, then use some optimization procedure to minimize the loss. Notice that you can't use training error to adjust the parameters $\mathbf{a}$ and $b$. Gradient descent on error rate won't work, because the gradient is zero almost everywhere. Instead, some approximation to the error rate is required.

One natural approximation is to interpret $u(\mathbf{x}; \mathbf{a}, b)$ in terms of a probability. Use the model

$$u(\mathbf{x}; \mathbf{a}, b) = \log \left[ \frac{P(\text{denoise}|\mathbf{x})}{P(\text{real}|\mathbf{x})} \right].$$

This means a data item with positive $u$ is likely to be from the denoiser, and more likely to be from the denoiser if $|u|$ is larger. A data item with a negative $u$ is likely to be real, and more likely so if $|u|$ is larger. In particular

$$P(\text{denoise}|\mathbf{x}) = \frac{e^u}{1+e^u} \text{ and } P(\text{real}|\mathbf{x}) = \frac{1}{1+e^u}.$$

Call this distribution the *predictive distribution* for the $i$'th example, and write $P(\cdot; u_i)$. Now write $\mathcal{S}$ for the set of examples, where each example has the form $(\mathbf{x}_i, y_i)$, and

$$y_i = \begin{cases} 1 & \text{if } i\text{'th example is real} \\ -1 & \text{otherwise} \end{cases}$$

Then the log-likelihood of the dataset under this model is

$$\mathcal{L}_{lr} = \sum_{i \in \mathcal{S}} \left[ u_i \left( \frac{1-y_i}{2} \right) - \log \left( 1 + e^{u_i} \right) \right]$$

(you should check this; **exercises** ). It would be natural to choose $\mathbf{a}$, $b$ to maximize this likelihood, a procedure known as *logistic regression*

### 17.1.3 The Cross Entropy Loss

The *cross-entropy* between a discrete distribution $p$ and another discrete distribution on the same space $q$ is

$$H_x(p, q) = -\mathbb{E}[p] [\log q] = -\sum_u p_u \log q_u$$

where the sum is over all elements with non-zero terms in $p$ and $q$. Now interpret the label for the $i$'th data item as a model probability distribution, by writing $p_i(\text{real}) = (1 + y_i)/2$ and $p_i(\text{denoise}) = 1 - p_i(\text{real}) = (1 - y_i)/2$. One of these is 1 and the other 0 for each data item, and there is a different distribution for each data item. Write $m_i$ for the $i$'th such example distribution and $P(\cdot; u_i)$ for the distribution predicted by the classifier for the $i$'th item. Notice that the logistic

loss is constructed out of cross-entropy terms, so

$$\begin{aligned}
\mathcal{L}_{lr} &= \sum_{i \in \mathcal{S}} \left[ u_i \left( \frac{1 - y_i}{2} \right) - \log \left( 1 + e^{u_i} \right) \right] \\
&= \sum_{i \in \mathcal{S}} \left[ \left( \frac{1 - y_i}{2} \right) \left[ u_i - \log \left( 1 + e^{u_i} \right) \right] + \left( \frac{1 + y_i}{2} \right) \left[ - \log \left( 1 + e^{u_i} \right) \right] \right] \\
&= \sum_{i \in \mathcal{S}} \left[ p_i(\text{real}) \log P(\text{real} | u_i) + p_i(\text{denoise}) \log P(\text{denoise} | u_i) \right] \\
&= - \sum_{i \in \mathcal{S}} H(m_i, P(\cdot; u_i)) \\
&= \mathcal{L}_{xe}.
\end{aligned}$$

This means that you can interpret the log-likelihood as a comparison between the predicted distribution and the model distribution for each data item.

### 17.1.4 The Logistic Loss

Write $s_i = y_i u_i = y_i(\mathbf{a}^T \mathbf{x} + b)$. The *logistic loss* function is given by

$$\mathcal{L}_{\text{logistic}}(s) = \frac{1}{\log 2} \left[ \log \left( 1 + e^{-s} \right) \right]$$

Then, by recalling that $\log \left( 1 + e^{f} \right) = f + \log \left( 1 + e^{-f} \right)$, you can show that the log-likelihood for logistic regression is

$$\mathcal{L}_{lr} = (\log 2) \sum_{i \in \mathcal{S}} \mathcal{L}_{\text{logistic}}(s_i)$$

(though the $\log 2$ factor is often ignored).

### 17.1.5 The Hinge Loss

The logistic loss has a helpful geometric interpretation in terms of the hyperplane $\mathbf{a}^T \mathbf{x} + b = 0$. If the $i$'th example is correctly classified and far from the hyperplane, $s_i$ is large and positive, and so $\mathcal{L}_{\text{logistic}}(s_i)$ is very close to zero. As $s_i$ gets closer to zero (and so the example gets closer to the hyperplane on the right side), the logistic loss grows. If $s_i$ is a lot smaller than zero (and so the example is far from the hyperplane and on the wrong side), the loss grows close to linearly in $s_i$ There are other loss functions that have this behavior. The *hinge loss* function

$$\mathcal{L}_{\text{hinge}}(s) = \max 1 - s 0$$

has this behavior as well. Recall $s_i = y_i(\mathbf{a}^T \mathbf{x}_i + b)$. The hinge loss for a dataset is

$$\sum_i \mathcal{L}_{\text{hinge}}(s_i).$$

If the example is correctly classified and far from the hyperplane, $s$ is larger than 1, and so the hinge loss is zero. If the example is correctly classified and close to the

hyperplane, $s$ is less than 1, and so the hinge loss is positive and gets bigger as the example gets closer to the hyperplane. If the example is incorrectly classified, the loss is positive, and the loss grows as the example gets further from the hyperplane (Figure **??**).

For both the hinge and the logistic loss, there is some cost to having an example close to the hyperplane even if it is on the right side. This effect helps ensure that the classifier performs well on test examples. You should expect future test examples to occur near to training examples. If (say) a training example is on the right side of the hyperplane, but is close to it, there is some possibility that some other, future example that is near the training example might also be on the wrong side of the hyperplane. This means it is a good idea for the loss to have a *margin* – a training example that is on the right side, but close to the hyperplane, should have loss greater than zero, and the loss should get bigger for examples that are closer to the hyperplane.

## 17.2 BUILDING A CLASSIFIER

The feature representation $\mathbf{x}$ used to classify the image patches could come from a fixed encoder recovered from an autoencoder. There is no good reason to do this, and several good reasons not to. It is mildly inconvenient to train an encoder on one problem, and use it on another. Worse, an encoder trained to do one thing may not be good at another. The encoder parameters were chosen to be good at denoising images, rather than to be good at distinguishing between real and denoised patches.

Much more natural is to build a classifier that accepts an image and predicts a value. If that value is positive, the classifier has labelled the patch a denoised patch; if negative, a real patch. Mostly, you know how to do this already. It is straightforward to repurpose the tools of Chapter **??** to do so. The classifier consists of a learned encoder that accepts an image and produces an $f \times 1$ dimensional vector which is passed to a linear classifier that produces a number. Sections 17.1.2, 17.1.3 and 17.1.5 offer options for scoring the numbers produced from a training set. The losses are (mostly) differentiable functions of the parameters $\mathbf{a}$ and $b$, so the machinery of Sections 15.2.3, 15.2.1 (and variants in Section 16.3) apply. The main open question is good ways to turn a block of encoded features into a vector.

### 17.2.1 Pooling

Chapters **??** and 16 showed procedures to produce a learned image representation: Apply a sequence of layers to an image, typically, convolutional, then ReLU, then convolutional, then ReLU, and so on. There could be stride in these layers, so that the block of features gets spatially smaller as it moves up the encoder.

For the result to be a vector, it must be $f \times 1 \times 1$. This could be achieved with stride alone, but an alternative is a *pooling layer* – a layer that reduces the spatial extent of the data block by forming summaries of local windows. Windows may overlap (depending on the API), but often don't. Quite usual is halve the spatial dimension of the image by pooling over non-overlapping $2 \times 2$ windows, so mapping from $f \times 2a \times 2b$ to $f \times a \times b$. In *average pooling*, the summary is the mean of the elements in the window in each feature layer, and in *max-pooling*, the summary is the maximum of the elements in the window in each feature layer. These pooling

layers have no learnable parameters (unlike, say, a convolutional layer with stride 2). Pooling layers differ by how they react to unusual (outlying) responses from feature detectors. Average pooling will tend to suppress them, whereas max-pooling will tend to emphasize them; there is some evidence that emphasizing them, and so max-pooling, is better on the whole for some classification purposes.

The layers, stride, padding and pooling are arranged so that the $c \times d \times d$ image results in a $g \times s \times s$ block. It is straightforward to turn this into a $g \times 1 \times 1$ block by average pooling over the two spatial dimensions.

### 17.2.2 Fully Connected Layers

You could regard the $g \times 1 \times 1$ block as a vector (in some APIs, you need to reshape, but this is housekeeping) and simply pass it to a linear classifier. Alternatively, you could transform this vector with a *fully connected layer*, which maps a vector $\mathbf{u}$ to a vector $\mathcal{C}\mathbf{u} + \mathbf{d}$, where the parameters $\mathcal{C}$ and $\lceil$ are learned, and $\mathcal{C}$ does not need to be square.

Notice that applying a linear classifier $\mathbf{a}^T\mathbf{x} + b$ to the output of a fully connected layer is not particularly interesting, because the result is $\mathbf{a}^T\mathbf{C}\mathbf{u} + \mathbf{a}^T\mathbf{d} + b$, which is just a different linear classifier. Similarly, applying a fully connected layer to another fully connected layer directly is not interesting. Instead, each fully connected layer is followed by a ReLU.

It is usual to take the $g \times 1 \times 1$ block, turn it into a vector if your API wants that, then pass it through a fully connected layer and then a ReLU layer at least once and possibly multiple times before applying a linear classifier. Experience teaches that it is helpful to pass high dimensional features to a linear classifier. This creates a minor tension, because big fully connected layers have a lot of parameters in them and can create issues with both inference and learning speed.

### 17.2.3 Training a Classifier

The encoder architecture produces an $f \times 1 \times 1$ block, and the classifier dots that vector with a parameter vector $\mathbf{a}$, adds $b$, and reports the result. This process is another layer, like the convolutional layers. Fold $\mathbf{a}$ and $b$ into the parameter vector $\theta$. The result is a function that accepts an example image $\mathcal{I}_i$ and produces a number. Write $F(\mathcal{I}_i, \theta)$ for the number that comes out of the classifier.

Choose one of the logistic or hinge losses, and write $\mathcal{C}$ for your chosen loss. Then the loss of applying the classifier to all training examples is

$$\sum_{i \in \text{train}} \mathcal{C}(F(\mathcal{I}_i, \theta), y_i)$$

and stochastic gradient descent can be applied to choose $\theta$ as in Section 15.2.1.

### 17.2.4 Worked Example

Figure 17.2 shows the architecture of a very simple classifier I used to classify real vs. denoised. I trained this classifier using a cross-entropy loss; the optimizer was Adam (Section 16.3.6); and I used batches of 128 images. I used 100, 000 images from the ImageNet training set (Section **??**), which I mapped to gray level

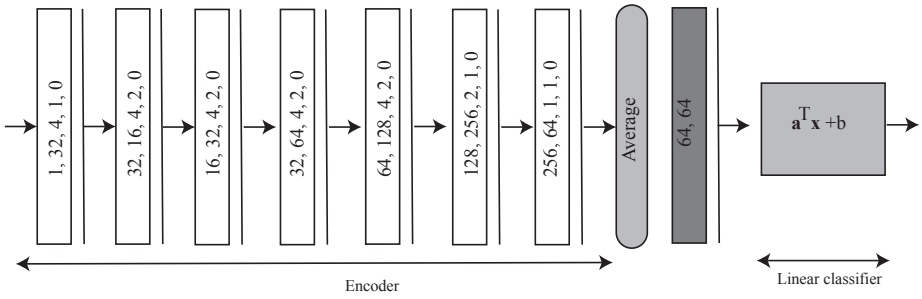


FIGURE 17.2: *A (very simple) classifier built out of convolutional layers, a pooling layer, a fully connected layer, and a linear classifier. Each light block represents a convolutional layer (arguments are, in order: input dimension, output dimension, kernel size, stride, padding); vertical lines are ReLU layers; the gray block with rounded corners pools over all spatial dimensions to produce a vector; and the dark gray block is a fully connected layer.*

images at $128 \times 128$ resolution. I obtained denoised images by applying the noise of Section 17.3.3 to training images, then denoising them with the autoencoder from that section (the one that uses skip connections). I used 20, 000 images from that set as test examples, and constructed denoised test examples as in training examples. This classifier is about as simple as it could be, and still quite easily tells test denoise images from test real images. The behavior of the classifier is summarised in Figure 17.3. Various modifications should lead to an improved classifier (**exercises** ). There is a very good chance of telling accurately whether an image has been through the autoencoder described in the text or not using a simple classifier – the error rate averaged over the whole validation set is 0.06 (so about one in 20 images will be misclassified).

A more devious example is the classifier of Figure 17.4. This classifier computes a feature vector for each image patch; applies a linear classifier to each such feature vector; and then reports (a) the hinge loss and (b) the score, averaged over all patches. It is worth taking a moment to check these statements against the figure. This classifier has the useful property that it checks whether individual image patches look good. The value of this property will become apparent (Section 8.2.2). This isn't a particularly good classifier for this application (validation error rate of 0.28),

## 17.3 FORCING IMAGES TO BE REALISTIC

You now have a classifier that can tell the difference between the output of an autoencoder and a real image. In principle, the value it makes could be used as a loss and provide a gradient. In practice, using these gradients effectively requires considerable care, but with care, they can produce important improvements in accuracy.

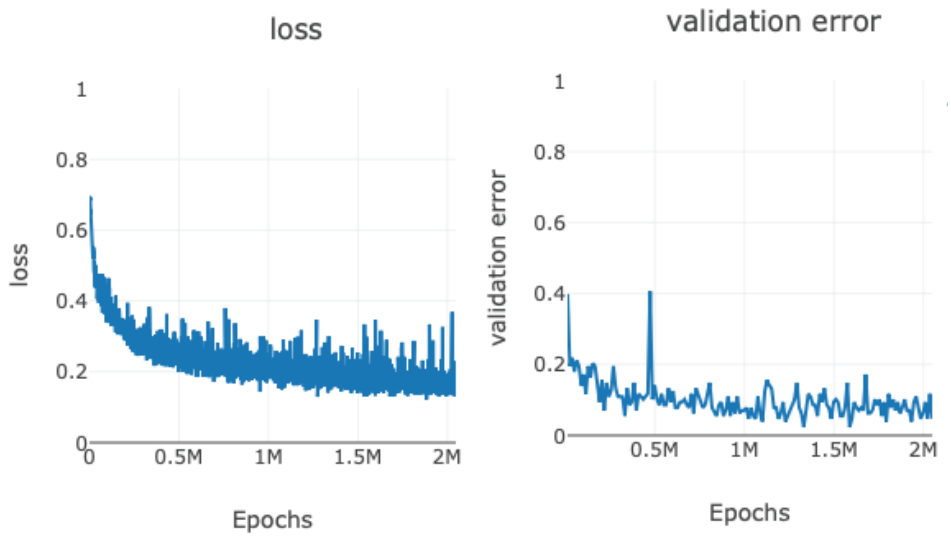


FIGURE 17.3: *A plot of training loss (***left***) and validation error rate (***right***) for the classifier of Figure 17.2, plotted against the number of training images the classifier has seen. These are fairly characteristic of a simple classifier. The loss mostly goes down, but there is some noise, particularly in the early stages of training (where a randomly selected batch may show the classifier effects it hasn't seen before). The validation error mostly goes down, then slows. The validation error is somewhat noisy, because it is measured on batches of 128 images rather than the whole validation set, so there is some chance of an odd batch. Eventually, the validation error rate must stall (it can't go below 0!) but – in this case – is small. There is a very good chance of telling accurately whether an image has been through the autoencoder described in the text or not using a simple classifier – the error rate averaged over the whole validation set is* 0.06 *(so about one in 20 images will be misclassified).*

### 17.3.1 The Obvious Strategy doesn't Work

Imagine we have a classifier that is good at telling whether an image has been through a particular autoencoder or not. You might use this classifier as a loss, by the following argument. Recall Section **??** interpreted $u(\mathbf{x}; \mathbf{a}, b)$ in terms of a probability with the model

$$u(\mathbf{x}; \mathbf{a}, b) = \log \left[ \frac{P(\text{denoise}|\mathbf{x})}{P(\text{real}|\mathbf{x})} \right].$$

so that a data item with positive $u$ is likely to be real, and more likely to be real if $|u|$ is larger. In turn, one could interpret $-u(\mathbf{x}; \mathbf{a}, b)$ as a loss. A more realistic set of reconstructions would have a smaller value of

$$\sum_{i \in \text{ae outputs}} u(\mathbf{x}_i; \mathbf{a}, b).$$

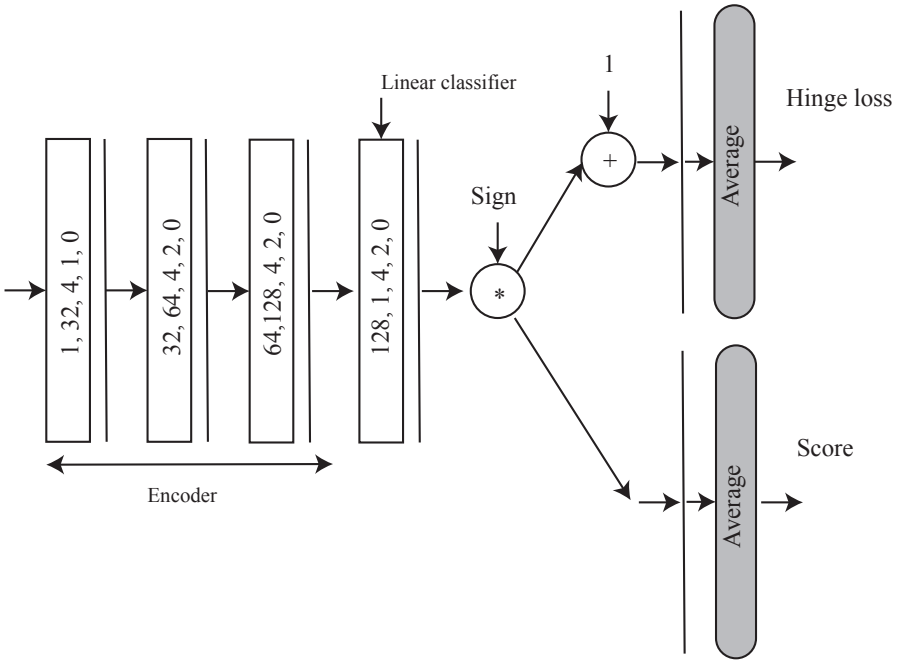


FIGURE 17.4: *This classifier looks at each image patch of a particular size, computes a score for that patch, and reports a loss (or score) averaged across the whole image. The patches overlap, and the size of the patches can be computed from the parameters of the convolutional layers (***exercises** *). Each light block represents a convolutional layer (arguments are, in order: input dimension, output dimension, kernel size, stride, padding); vertical lines are ReLU layers; the gray block with rounded corners pools over all spatial dimensions to produce a vector; and the dark gray block is a fully connected layer. This classifier is not particularly accurate – the error rate averaged over the whole validation set is* 0.28 *(so about one in four images will be misclassified) – but classifiers built like this will turn out to be useful.*

You could do this whether you use the hinge loss or the cross-entropy loss to train the classifier (**exercises** ).

Write $u(\mathcal{I};\theta_c)$ for the value the classifier with parameters $\theta_c$ produces when given image $\mathcal{I}$. For the moment, fix the classifier and consider training the autoencoder. Write $(\mathcal{N}_i, \mathcal{C}_i)$ for training pairs of noisy image $\mathcal{N}_i$ and clean real image $\mathcal{C}_i$, $\mathcal{A}(\mathcal{N}_i;\theta_a)$ for the image produced by the autoencoder with parameters $\theta_a$ given input $\mathcal{N}_i$. There is some loss $\mathcal{L}$ that compares the autoencoder output to the clean image, which might be some combination of L1 and L2 losses. In principle, you could train an autoencoder; then train a classifier to get $\theta_c$; then freeze the classifier and continue to train the autoencoder using the loss

$$\sum_i \mathcal{L}(\mathcal{A}(\mathcal{N}_i;\theta_a), \mathcal{C}_i) + \lambda \sum_i -u(\mathcal{A}(\mathcal{N}_i;\theta_a);\theta_c)$$

where $\lambda$ is some weight you choose to get good performance. ***This approach does not work*** (try it!).

### 17.3.2 A More Subtle Approach that Works

The problem is a basic property of classifiers. Real images and autoencoder outputs could be different in many ways. The classifier finds the most effective direction in feature space to distinguish between them, *not* all of the directions. For example, imagine the autoencoder places a red pixel in the top left hand corner of every image *and* a blue pixel in the bottom right hand corner of every image. The classifier might very well identify the red pixel, but ignore the blue pixel. In this case, if you use a fixed classifier to polish the autoencoder, the autoencoder will likely stop putting red pixels in the top left hand corner – because the classifier notices them, and objects – but will not fix the blue pixel and might even insert red pixels somewhere else. To prevent this, you will need to adjust the classifier once you have adjusted the autoencoder, and repeat.

The important property here is that the largest difference between the autoencoder outputs and the real images should be small. Equivalently, the best classifier should perform poorly. Imagine you start with the best classifier. If you then train the autoencoder for some steps, *that classifier is no longer the best* (because the autoencoder has explicitly been trained to fool it). To have some hope of achieving this property, the classifier needs to change once the autoencoder has changed.

Write $\mathcal{R}_i$ for a set of real images (which could be just the clean examples from the dataset for the autoencoder). Label all real images with $y_i = 1$, and all autoencoder outputs with $y_i = -1$. Recall $s_i(\theta_c) = y_i u(\mathcal{I}_i; \theta_c)$, and write $\mathcal{L}_c(s_i)$ for some classification loss. The process should look like iterating:

- Fix an autoencoder, then adjust a classifier slightly using the outputs of this autoencoder; that is, use the autoencoder to produce a set of outputs, label them, take some steps to minimize

$$\sum_{i \in \text{data}} \mathcal{L}_c(s_i(\theta_c))$$

  (the training loss of the classifier) as a function of $\theta_c$.

- Fix the classifier, and adjust the autoencoder to fool the classifier; that is, take some steps to reduce

$$\sum_i \mathcal{L}(\mathcal{A}(\mathcal{N}_i; \theta_a), \mathcal{C}_i) + \lambda \sum_i -u(\mathcal{A}(\mathcal{N}_i; \theta_a); \theta_c)$$

  as a function of $\theta_a$.

At this point, the classifier isn't really a loss, because each time you train the autoencoder *you are using a different classifier*. The main point of the classifier is to supply a helpful gradient to the autoencoder. You should visualize this as a competition between the classifier and the autoencoder. The term

$$\sum_i -u(\mathcal{A}(\mathcal{N}_i; \theta_a); \theta_c)$$

is often referred to as an *adversarial loss* and the classifier as an *adversary*. Getting all this to work well can be surprisingly tricky, because it is important that neither "win" the competition.

A really bad classifier presents problems. Imagine the classifier simply cannot tell the difference between real images and autoencoder outputs: the gradient is unlikely to be helpful. It is possible, but unlikely, that this occurs because the autoencoder produces things that are indistinguishable from real images. It is much more likely that the classifier is simply not strong enough. This could occur either because the classifier architecture can't build sufficiently strong features (in the case of the red pixel autoencoder example, imagine the classifier looks only at the central pixel of the image) or because it is very poorly trained. This effect can manifest in training, because the autoencoder could improve faster than the classifier and eventually the classifier is no longer able to distinguish between real and fake images. It is common to say that the autoencoder has beaten the classifier.

A really good classifier also presents problems. Imagine the classifier is very good. It may not supply a useful gradient to the autoencoder, because any small change to the autoencoder will likely still produce objects that are distinctively not images. This usually happens because the classifier improves faster than the autoencoder, but could also occur because the autoencoder is fundamentally incapable of producing objects that are like real images. For example, Figure 41.2 shows that autoencoders without skip connections have real trouble producing sharp edges. Using an adversary during training isn't going to make this problem go away. It is common to say that the classifier has beaten the autoencoder. All this means that it is difficult to use a pretrained classifier successfully (**exercises** ).

### 17.3.3 Worked Example

I trained the best of the simple autoencoders (three layer encoder, three layer decoder, skip connections) of Section using a straightforward adversary. The autoencoder and adversary are sketched in Figure 41.2. Figure 17.5 compares this autoencoder trained with and without an adversary. You should notice that edges are sharper, which isn't always a good thing. In some cases, the adversary encourages the reconstruction of a block of pixels that was knocked out to have a sharp edge, making it more noticeable. Mostly the effect is helpful, however. Figure 17.6 plots various loss terms coomputed for training batches as the autoencoder is being trained. You may find it startling that the adversarial term *rises* as training continues. But remember, this isn't a loss – it's not the value of one function. Instead, it is the value of a different function at each step (because the classifier keeps changing). Usually, it is a good sign that this term increases, because it means that the classifier is finding it harder to distinguish between real images and autoencoder outputs.

Figure 17.5 summarizes the results. In this figure, the **top row** shows noised inputs to a denoising autoencoder; the **second row** shows outputs from an autoencoder with skip connections trained with an L1/L2 loss but without an adversary; the **third row** shows outputs from the same autoencoder, now trained with a big adversary (the adversary sees the whole image); finally, the **last row** shows outputs from the same autoencoder, now trained with a small adversary (which sees only

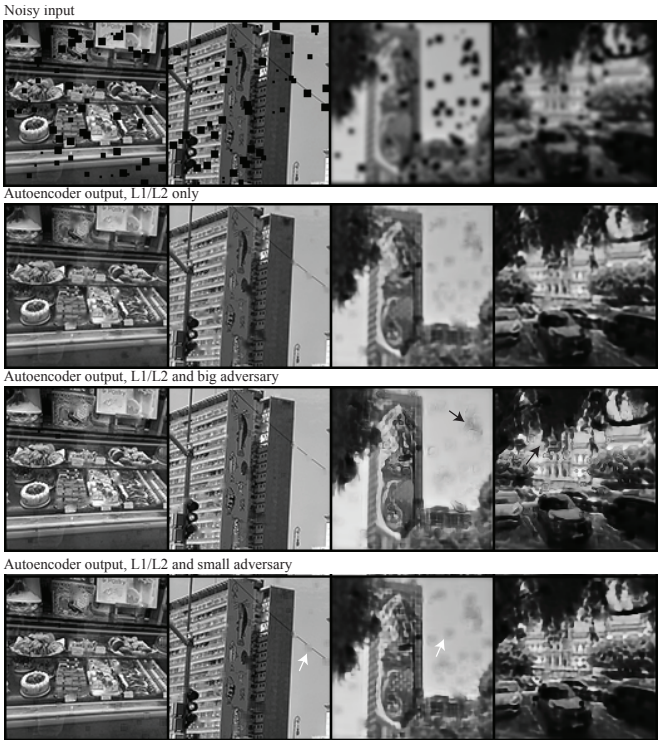


FIGURE 17.5: *An adversary improves the behavior of a denoising autoencoder by encouraging it to produce outputs that can't be distinguished from real images. Notice that the main effect is to make edges sharper (the differences are mainly at edges). Mostly, but not always, this improves the outputs. Dark arrows point to some cases where the adversary has arguably made the output worse. Notice that, when the adversary can see only local patches, the autoencoder cannot recover from large dark knocked out patches (light arrows). Notice also how severely noisy the image on the* **far right** *is; the autoencoders recover from this moderately well.*

patches of the image).

I have not shown the PSNR for this autoencoder, but you should *expect* that it is worse than the PSNR for an autoencoder trained without a discriminator. The PSNR measures the similarity between output and ground truth in a version of the L2 norm. The whole point of a discriminator is to encourage the autoencoder to produce something that is more like a real image, even if the L2 norm is worse.

### 17.3.4 Training Issues

Making all this work can be tricky. There is good experience showing that adversarial terms can be very helpful, but ensuring that a model behaves acceptably can be difficult. You should notice the method I have described applies quite widely, rather than just to autoencoders. Imagine you have a network that produces out-

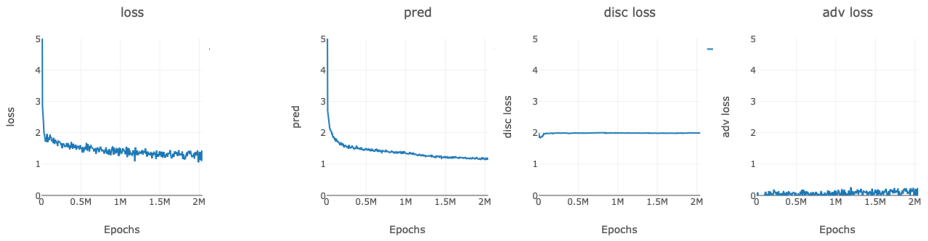


FIGURE 17.6: *Plots of the overall loss, L1/L2 error (labelled "Pred"), classifier training loss ("disc loss") and adversarial term ("adv loss") observed during training the autoencoder of Section 17.3.3, as a function of the number of training images. Notice how the L1/L2 error declines (this doesn't – and shouldn't – always happen). The discriminator is not very good, because the classifier training loss is always quite high. Nonetheless, it contributes to training quite helpfully, the increase in the adversarial term suggests and as Figure 17.5 confirms.*

puts that are "like" images, in the sense the outputs are big, most outputs are unacceptable (in the same way that most arrays aren't images), you can get examples of acceptable outputs, and it is quite difficult to write a cost that ensures your output is acceptable. This is an extremely common situation (Chapters **??** and **??**). Then you could adjust all the discussion above to use an adversary to control the outputs. It is now common practice to refer to such networks as *generator*s.

**Balance** is a major source of problems. Typically, either the generator or the classifier wins, and the classifier no longer provides meaningful gradient. There are a number of helpful strategies, but none works for every problem. If the generator wins, you can take more (or larger) steps to adjust the classifier in each round. If the classifier wins, you can take more (or larger) steps to adjust the generator in each round. You can try making the classifier dumber (if it wins) or smarter (if it loses) by changing the number of layers, or the depth of the layers, or the size of fully connected layers. Usually, you know when to intervene by watching the training loss of the classifier and the value of the adversarial term. What you are looking for is quite noisy behavior, with a slow rise in the adversarial term that may then flatten out. If the training loss of the classifier is high and doesn't go down, it is likely losing. If the adversarial term is high and doesn't go down, the generator is likely losing.

**Bad behavior in the classifier** can be a nuisance. For a convolutional or a fully connected layer, reshape the input into a vector $\mathbf{x}$. You can then write the output of the layer as $\mathcal{A}\mathbf{x} + \mathbf{b}$, for some matrix $\mathcal{A}$ and vector $\mathbf{b}$. Now imagine that the matrix $\mathcal{A}$ has the property that a small change in $\mathbf{x}$ produces a large change in the output. Equivalently, there is some direction $\delta\mathbf{x}$ such that $|\mathcal{A}\delta\mathbf{x}\|$ is big even when $\|\delta\mathbf{x}\|$ is small. If the generator can find this direction, it can cause the classifier to radically change its labels while changing the image by a very small amount. Further, the adversary may supply very small gradients to the generator (because a small change in input results in a large change in output).

**Spectral normalization** is a successful method to control this bad behavior. Spectral normalization ensures that the matrix $\mathcal{A}$ associated with a convolutional or fully connected layer has the property that

$$\max_{\mathbf{u}} \frac{\|\mathcal{A}\mathbf{u}\|}{\|\mathbf{u}\|} = 1$$

and is available in most APIs.

**ReLUs affect gradients** in ways that aren't helpful for an adversary. When you are training a classifier, the gradients that matter are gradients of the classifier output with respect to its parameters (check Sections 17.2.3, 15.2.1 and 15.2.3 if you're not sure). But if you use that classifier as an adversary, the gradient of the output with respect to the input is also important, because this is what goes to the generator. If some feature arriving at a ReLU is negative, the ReLU sets that feature to zero. The gradient of the output with respect to the input then has no component to encourage that feature to go positive – all information is lost because the gradient of a constant zero value is zero. This is not an issue training a classifier *if* you want to use the classifier as a classifier.

**The Leaky ReLU** is a method to improve the gradients an adversary provides a generator. A leaky ReLU with constant $c$ maps $x$ to

$$\text{leakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ cx & \text{otherwise} \end{cases}$$

common practice sets $c = 0.2$. Adversaries built with leaky ReLU's, rather than ReLU's, tend to behave better.

**Spatial scale** is important. It is straightforward to build a classifier that evaluates whether image patches (rather than the whole image) are realistic (**exercises** ). The size of the patch chosen have important consequences. If the patch is too small, then the adversary may not be able to resolve bad behavior by the generator. For example, an adversary might need to see moderately sized windows to tell if the generator produces blurry edges, but relatively small windows to tell if the generator refuses to produce some colors. If the patch is too large – for example, the whole image – there is a chance the adversary notices and benefits from some bias in the original set of real images. For example, if every example of a real image shows a man-made scene, the adversary might bias the autoencoder to make straight lines even if the original image doesn't have them.