CHAPTER 21

Forcing Predictions to be Realistic

There are strong links between the optimization based denoisers of Chapter 9 and the learned denoisers of Chapters 16 and 17. 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".

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.

21.1 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.

21.1.1 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 20.2.2 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 a denoised image, and more likely to be denoised 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

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

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 θ_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 θ_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 θ_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 λ is some weight you choose to get good performance. ***This approach does not work*** (try it!).

21.1.2 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 θ_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 θ_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 22.6 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**).

21.1.3 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 22.6. Figure 21.1 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 21.2 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 21.1 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 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.

21.1.4 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 outputs 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 **??**).

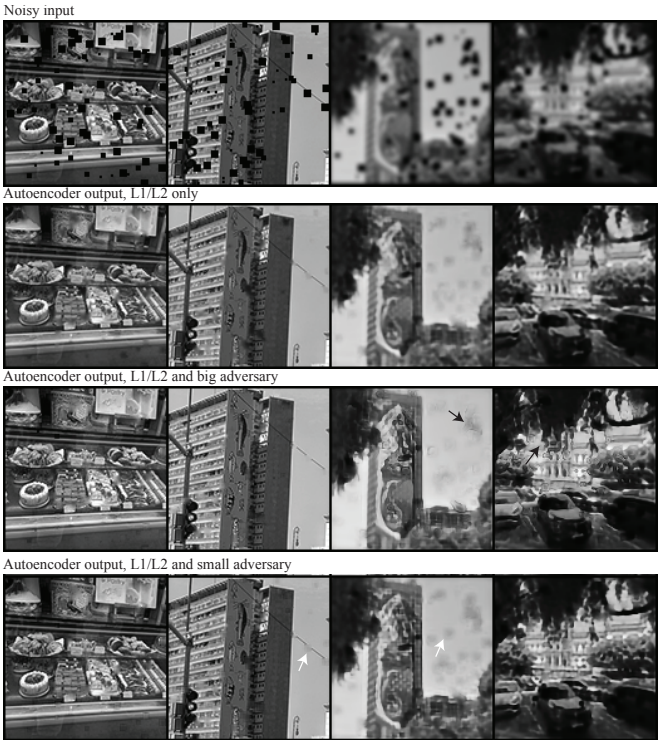

FIGURE 21.1: *An adversary improves the behavior of a denoising autoencoder by encouraging it to produce outputs that can't be distinguished from real images. Notice that the main effect is to make edges sharper (the differences are mainly at edges). Mostly, but not always, this improves the outputs. 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.*

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

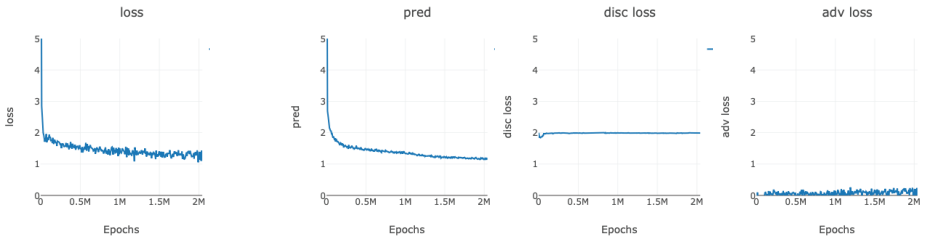

FIGURE 21.2: *Plots of the overall loss, L1/L2 error (labelled "Pred"), classifier training loss ("disc loss") and adversarial term ("adv loss") observed during training the autoencoder of Section 21.1.3, as a function of the number of training images. Notice how the L1/L2 error declines (this doesn't – 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 21.1 confirms.*

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 20.3.3, 16.2.1 and 16.2.3 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.

21.1.5 Application to other Predictions

TODO: fill this in