Lecture 20: Reversible and Autoregressive Models

Roger Grosse

As we saw last time, GANs are very good at producing convincing samples for complex data distributions. But they're an implicit generative model, which means they don't explicitly represent the density, i.e. we can't evaluate the probability of an observation. This means we can't measure the likelihood, which has two implications:

- We can't train the model using maximum likelihood.
- Without measuring likelihoods, it's hard to tell if the network is really modeling the distribution well. E.g., it could be memorizing the training data, or ignoring part of the distribution. If we could compute likelihoods, we could, e.g., measure if there's a big difference between likelihoods on the training and test sets.

In this lecture, we'll cover two kinds deep generative model architectures for which we *can* measure the likelihood, and hence can train them using maximum likelihood. The first kind is reversible architectures, where the network's computations can be inverted in order to recover the input which maps to a given output. We'll see that this makes the likelihood computation tractable.

The second kind of architecture is autoregressive models. This isn't new: we've already covered neural language models and RNN language models, both of which are examples of autoregressive models. In this lecture, we'll introduce two tricks for making them much more scalable, so that we can apply them to high-dimensional data modalities like high-resolution images and audio waveforms.

1 Reversible Models

Recall the GAN generator architecture from last lecture: we would first sample a code vector from a fixed, simple distribution such as uniform or spherical Gaussian. The generator (which is a deterministic feed-forward network) maps the code vector to the observation space. Hopefully, the distribution of the network's outputs should approximate the data distribution. We noted that this was an implicit generative model, since it's intractable to determine the density $p(\mathbf{x})$ for any observation $\mathbf{x}$. But if we modify the generator architecture to be reversible, then it's possible to compute the density.

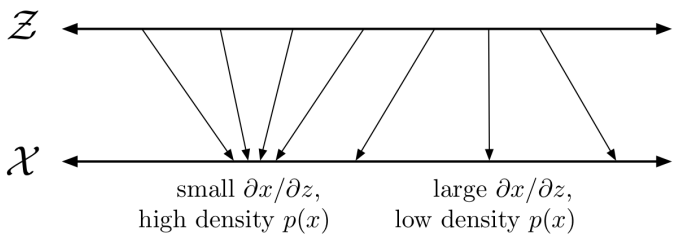
Mathematically, this is based on the **change-of-variables formula** for probability density functions. Suppose we have a bijective, differentiable mapping $f : \mathcal{Z} \to \mathcal{X}$. ("Bijective" means the mapping must be 1–1 and cover all of $\mathcal{X}$.) Since f is bijective, we can think of it as representing a

change-of-variables transformation. For instance, $\mathbf{x} = f(\mathbf{z}) = 12\mathbf{z}$ could represent a conversion of units from feet to inches. If we have a density $p_Z(\mathbf{z})$, the change-of-variables formula gives us the density $p_X(\mathbf{x})$:

$$p_X(\mathbf{x}) = p_Z(\mathbf{z}) \left| \det \left(\frac{\partial \mathbf{x}}{\partial \mathbf{z}} \right) \right|^{-1}, \tag{1}$$

where $\mathbf{z} = f^{-1}(\mathbf{x})$. Let's unpack this. First, $\partial \mathbf{x}/\partial \mathbf{z}$ is the Jacobian of f, which is the linearization of f around $\mathbf{z}$. Then we take the absolute value of the matrix determinant. Recall that the absolute value of the determinant of a matrix gives the factor by which the associated linear transformation expands or contracts the volume of a set. So the determinant of the Jacobian determines how much f is expanding or contracting the volume locally around $\mathbf{z}$. We then take the inverse of the determinant, which means if f expands the volume, then the density $p_X(\mathbf{x})$ shrinks, and vice versa. Heuristically, this is justified by the following picture:

If we consider the linear transformation $\mathbf{x} \mapsto \mathbf{A}\mathbf{x}$ for some matrix $\mathbf{A}$, and apply it to a set with volume V, we'll get a set with volume $V|\det \mathbf{A}|$.



Now suppose the mapping f is the function computed by a generator network (i.e. its outputs as a function of its inputs). It's tempting to apply the change-of-variables formula in order to compute $p_X(\mathbf{x})$. But in order for this to work, three things need to be true:

1. The mapping f needs to be differentiable, so that the Jacobian $\partial \mathbf{x}/\partial \mathbf{z}$ is defined.

2. We need to be able to compute $\mathbf{z} = f^{-1}(\mathbf{x})$, which means f needs to be invertible, with an easy-to-compute inverse.

3. We need to be able to compute the (log) determinant of the Jacobian.

With regards to (1), networks with ReLU nonlinearities technically aren't differentiable because ReLU is nondifferentiable at 0. In practice, we can ignore this issue because the inputs to the activation function are very unlikely to be exactly zero, so with high probability, the Jacobian will be defined. Or, if we're still worried, we could just pick a differentiable activation function. But the other two points are much harder to deal with.

Fortunately, there's a simple and elegant kind of network architecture called a **reversible architecture** which is efficiently invertible and for which we can compute the log determinant efficiently. (In fact, the determinant turns out to be 1.) This architecture is based on the **reversible block**, which is very similar to the residual block from Lecture 17. Recall that residual blocks implement the following equation:

$$\mathbf{y} = \mathbf{x} + \mathcal{F}(\mathbf{x}), \tag{2}$$

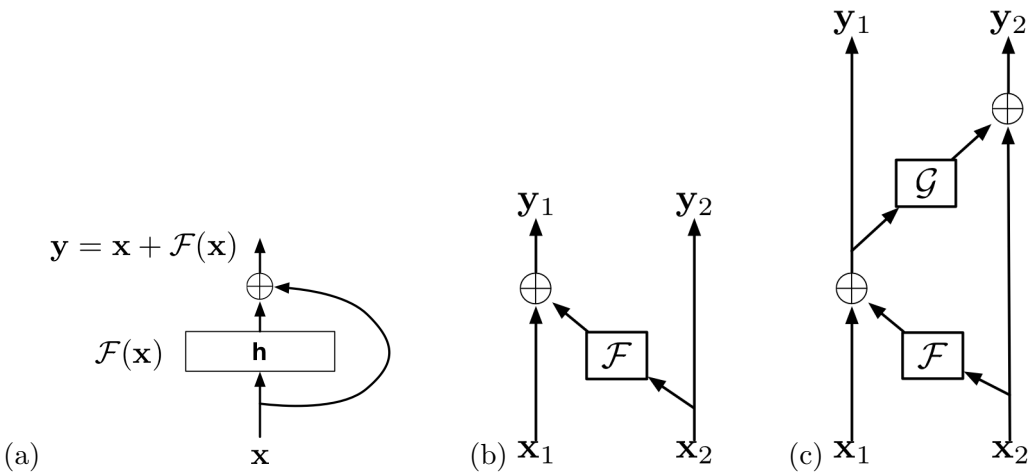


Figure 1: **(a)** A residual block. **(b)** A reversible block. **(c)** A composition of two reversible blocks.

where $\mathcal{F}$ is some function, such as a shallow network. Reversible blocks are similar, except that we divide the units into two groups; the residual function for the first group depends only on the other group, and the second group is left unchanged. Mathematically,

$$\begin{aligned} \mathbf{y}_1 &= \mathbf{x}_1 + \mathcal{F}(\mathbf{x}_2) \\ \mathbf{y}_2 &= \mathbf{x}_2 \end{aligned} \tag{3}$$

This is shown schematically in Figure 1. The reversible block is easily inverted, i.e. if we're given $\mathbf{y}_1$ and $\mathbf{y}_2$, we can recover $\mathbf{x}_1$ and $\mathbf{x}_2$:

$$\begin{aligned} \mathbf{x}_2 &= \mathbf{y}_2 \\ \mathbf{x}_1 &= \mathbf{y}_1 - \mathcal{F}(\mathbf{x}_2) \end{aligned} \tag{4}$$

Here's what happens when we compose two residual blocks, with the roles of $\mathbf{x}_1$ and $\mathbf{x}_2$ swapped:

$$\begin{aligned} \mathbf{y}_1 &= \mathbf{x}_1 + \mathcal{F}(\mathbf{x}_2) \\ \mathbf{y}_2 &= \mathbf{x}_2 + \mathcal{G}(\mathbf{y}_1) \end{aligned} \tag{5}$$

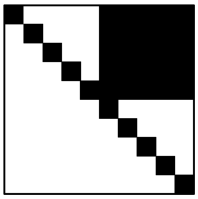
This is shown schematically in Figure 1. To invert the composition of two blocks:

$$\begin{aligned} \mathbf{x}_2 &= \mathbf{y}_2 - \mathcal{G}(\mathbf{y}_1) \\ \mathbf{x}_1 &= \mathbf{y}_1 - \mathcal{F}(\mathbf{x}_2) \end{aligned} \tag{6}$$

So we've shown how to invert a reversible block. What about the determinant of the Jacobian? Here is the formula for the Jacobian, which we get by differentiating Eqn. 3 and putting the result into the form of a block matrix:

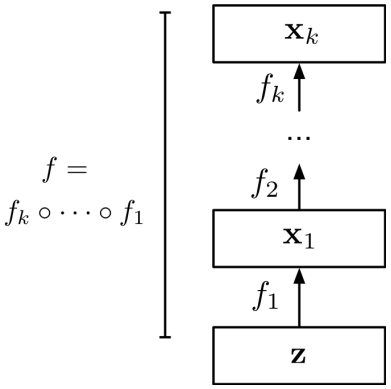
$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{pmatrix} \mathbf{I} & \frac{\partial \mathcal{F}}{\partial \mathbf{x}_2} \\ \mathbf{0} & \mathbf{I} \end{pmatrix}$$

(Recall that $\mathbf{I}$ denotes the identity matrix.) Here's the pattern of nonzero entries of this matrix:



This is an upper triangular matrix. Think back to linear algebra class: the determinant of an upper triangular matrix is simply the product of the diagonal entries. In this case, the diagonal entries are all 1's, so the determinant is 1. How convenient! Since the determinant is 1, the mapping is **volume preserving**, i.e. ~~it maps any given set to another set of the same volume.~~ In our context, this just means the determinant term disappears from the change-of-variables formula (Eqn. 1).

All this analysis so far was for a single reversible block. What if we build a reversible network by chaining together lots of reversible blocks?



Fortunately, inversion of the whole network is still easy, since we just invert each block from top to bottom. Mathematically,

$$f^{-1} = f_1^{-1} \circ \cdots \circ f_k^{-1}. \tag{7}$$

For the determinant, we can apply the chain rule for derivatives, followed by the product rule for determinants:

$$\begin{aligned}
\left|\frac{\partial \mathbf{x}_k}{\partial \mathbf{z}}\right| &= \left|\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_{k-1}} \cdots \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1}\frac{\partial \mathbf{x}_1}{\partial \mathbf{z}}\right| \\
&= \left|\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_{k-1}}\right| \cdots \left|\frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1}\right| \left|\frac{\partial \mathbf{x}_1}{\partial \mathbf{z}}\right| \\
&= 1 \cdot 1 \cdots 1 \\
&= 1
\end{aligned} \tag{8}$$

Hence, the full reversible network is also volume preserving.

Because we can compute inverses and determinants, we can train a reversible generative model using maximum likelihood using the change-of-variables formula. This is the idea behind **nonlinear independent components estimation (NICE)**[1]. (This paper introduced the idea of training

[1] Dinh et al., 2014. NICE: Non-linear independent components estimation.

reversible architectures with maximum likelihood.) The change-of-variables formula gives us:

$$\begin{aligned} p_X(\mathbf{x}) &= p_Z(\mathbf{z}) \left| \det \left(\frac{\partial \mathbf{x}}{\partial \mathbf{z}} \right) \right|^{-1} \\ &= p_Z(\mathbf{z}) \end{aligned} \tag{9}$$

Hence, the maximum likelihood objective over the whole dataset is:

$$\prod_{i=1}^{N} p_X(\mathbf{x}^{(i)}) = \prod_{i=1}^{N} p_Z(f^{-1}(\mathbf{x}^{(i)})) \tag{10}$$

Remember, p_Z is a simple, fixed distribution (e.g. independent Gaussians), so $p_Z(\mathbf{z})$ is easy to evaluate. Note that this objective only makes sense because of the volume constraint.

If f weren't constrained to be volume preserving, then f^{-1} could map every training example very close to $\mathbf{0}$, and hence $p_Z(f^{-1}(\mathbf{x}^{(i)}))$ would be large for every training example. The volume preservation constraint prevents this trivial solution.

2 Autoregressive Models

Autoregressive models are another kind of deep generative model with tractable likelihoods. We've already seen two examples in this course: the ~~neural language model~~ (Lecture 7) and ~~RNNs~~ (Lectures 15-17). Here, the observations were given as sequences $(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)})$, and we decomposed the likelihood into a product of conditional distributions:

$$p(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(t)}) = \prod_{t=1}^{T} p(\mathbf{x}^{(t)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(t-1)}). \tag{11}$$

So the maximum likelihood objective decomposes as a sequence of prediction problems for each term in the sequence given the previous terms. Assuming the observations were discrete (as they are in all the autoregressive models considered in this course), the prediction at each time step can be made using a neural network which outputs a probability distribution using a softmax activation function.

So far, we've mostly considered using short sequences or short context lengths. The neural language model from Assignment 1 used context windows of length 3, though the architecture could work better with contexts of length 10 or so. Machine translation at the word level involves outputting sequences of length 20 or so (the typical number of words in a sentence).

But what if accurately modeling the distribution requires much longer term dependencies? One example is autoregressive models of images. A grayscale image is typically represented with pixel values which are integers from 0 to 255 (i.e. one byte each). We can treat this as a sequence using the raster scan order (Figure 2). But even a fairly small image would correspond to a very long sequence, e.g. a 100×100 image would correspond to a sequence of length 10,000. Clearly, images have a lot of global structure, so the predictions would need to take into account all of the pixels which were already generated. As another example, consider learning a generative model of audio waveforms. An audio waveform is stored as a sequence of integer-valued samples, with a sampling rate of at least 16,000 Hz (cycles/second) in order to have reasonably good sound quality. This means

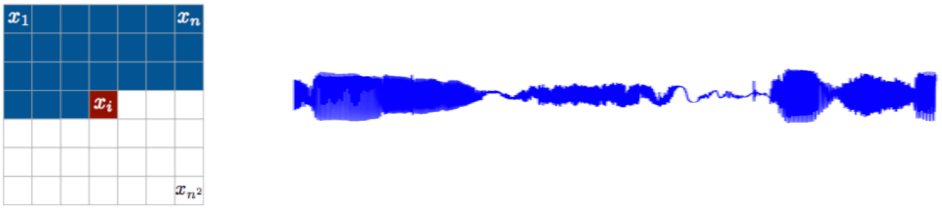


Figure 2: Examples of sequence modeling tasks with very long contexts. **Left:** Modeling images as sequences using raster scan order. **Right:** Modeling an audio waveform (e.g. speech signal).

that to predict the next term in the sequence, if we want to account for even 1 second of context, this requires a context of length 16,000.

One way to account for such a long context is to use an RNN, which (through its hidden units) accounts for the entire sequence that was generated so far. The problem is that computing the hidden units for each time step depends on the hidden units from the previous time step, so the forward pass of backprop requires a `for`-loop over time steps. (The backward pass requires a `for`-loop as well.) With thousands of time steps, this can get very expensive. But think about the neural language model architecture from Lecture 7. At training time, the predictions at each time step are done independently of each other, so all the time steps can be processed simultaneously with vectorized computations. This implies that training with very long sequences could be done much more efficiently if we could somehow get rid of the recurrent connections.

Causal convolution is an elegant solution to this problem. Observe that, in order to apply the chain rule for conditional probability (Eqn. 11), it's important that information never leak backwards in time, i.e. that each prediction be made only using observations from earlier in the sequence. A model with this property is called **causal**. We can design convolutional neural nets (CNNs) to have a causal structure by masking their connections, i.e. constraining certain of their weights to be zero, as shown in Figure 3. At training time, the predictions can be computed for the entire sequence with a single forward pass through the CNN. Causal convolution is a particularly elegant architecture in that it allows computations to be shared between the predictions for different time steps, e.g. a given unit in the first layer will affect the predictions at multiple different time steps.

It's interesting to contrast a causal convolution architecture with an RNN. We could turn the causal CNN into an RNN by adding recurrent connections between the hidden units. This would have the advantage that, because of its memory, the model could use information from all previous time steps to make its predictions. But training would be very slow, since it would require a `for`-loop over time steps. A very influential recent paper[2] showed that both strategies are actually highly effective for modeling images. Take a moment to look at the examples in that paper.

[2]van den Oord et al., 2016, "Pixel recurrent neural networks". `https://arxiv.org/abs/1601.06759`

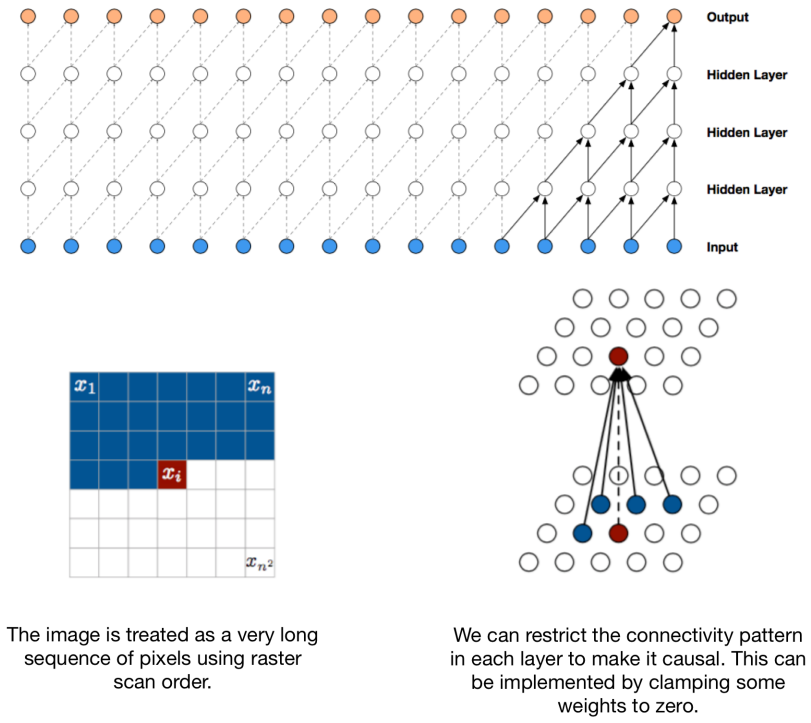


Figure 3: **Top:** a causal CNN applied to sequential data (such as an audio waveform). Source: van den Oord et al., 2016, "WaveNet: a generative model for raw audio". **Bottom:** applying causal convolution to modeling images. Source: van den Oord et al., 2016, "Pixel recurrent neural networks".

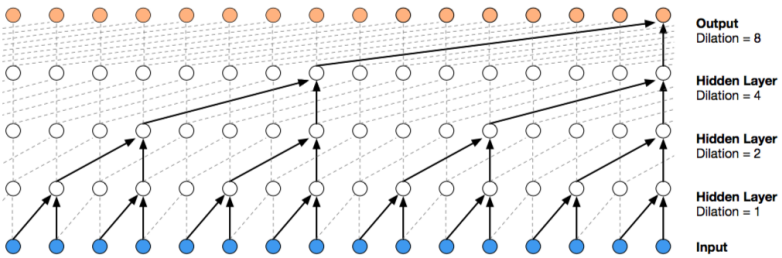


Figure 4: The dilated convolution architecture used in WaveNet. Source: van den Oord et al., 2016, "WaveNet: a generative model for raw audio".

The problem with a straightforward CNN architecture is that the predictions are made using a relatively short context because the output units have a small receptive field. Fortunately, there's a clever fix for this problem, which you've already seen in Programming Assignment 2: **dilated convolution**. Recall that this means that each unit receives connections from units in the previous layer with a spacing larger than 1. Figure 4 shows part of the dilated convolution architecture for WaveNet[3], an autoregressive model for audio. ~~The first layer has a dilation of 1, so each unit has a receptive field of size 1. The next layer has a dilation of 2, so each unit has a receptive field of size 2.~~ The dilation factors are spaced by factors of 2, i.e., $\{1, 2, \ldots, 512\}$, so that the 10th layer has receptive fields of size 1024. Hence, it gets exponentially large receptive fields with only a linear number of connections. This 10-layer architecture is repeated 5 times, so that the receptive fields are approximately of size 5000, or about 300 milliseconds. This is a large enough context to generate impressively good audio. You can find some neat examples here:

`https://deepmind.com/blog/wavenet-generative-model-raw-audio/`

Compared with other autoregressive models, causal, dilated CNNs are quite efficient at training time, despite their large context. However, all autoregressive models, including both CNNs and RNNs, share a common disadvantage: they are very slow to generate from, since the model's own samples need to be fed in as inputs, which means it requires a `for`-loop over time steps. So if efficiency of generation is a big concern, then GANs or reversible models would be much preferred. Learning a generative model of audio which is of similar quality to WaveNet, yet also efficient to generate from, is an active area of research.

[3] van den Oord et al., 2016, "WaveNet: a generative model for raw audio". `https://arxiv.org/abs/1609.03499`