CS-GY 9223, Fall 2020

Textbook reading: Sec 17

Generative Models

Motivation

Much of what we have discussed in the first part of this course has been in the context of making *deterministic, point* predictions: given image, predict cat vs dog; given sequence of words, predict next word; given image, locate all balloons; given a piece of music, classify it; etc. By now you should be quite clear (and confident) in your ability to solve such tasks using deep learning (given, of course, the usual caveats on dataset size, quality, loss function, etc etc).

All of the above tasks have a well defined *answer* to whatever question we are asking, and deep networks trained with suitable supervision can find them. But modern deep networks can be used for several other interesting tasks that conceivably fall into the purview of "artificial intelligence". For example, think about the following tasks (that humans can do quite well), that do not cleanly fit into the supervised learning:

- find underlying laws/characteristics that are salient in a given corpus of data.
- given a topic/keyword (say "water lily"), draw/synthesize a new painting (or 250 paintings, all different) based on the keyword.
- given a photograph of a face (with the left half blacked out), mentally visualize how the rest would look like.
- be able to quickly adapt to new tasks.
- be able to memorize and recall objects.
- be able to plan ahead in the face of uncertain and changing environments;

among many others.

In the latter part of the course we will focus on solving such tasks. Somewhat fortunately, the main ingredients of deep learning (feedforward/recurrent architectures, gradient descent/backpropagation, data representations) will remain the same – but we will put them together into novel formulations.

Tasks such as classification/regression are inherently *discriminative* – the network learns to figure out *the* answer (or label) for a given input. Tasks such as synthesis are inherently *generative* – there is no one answer, and instead the network will need to figure out a *probability distribution* (or, loosely, a *set*) of possible answers to it. Let us see how to train neural nets that learn to produce such distributions.

[Side note: machine learning/statistics has long dealt with modeling uncertainty and producing distributions. Probabilistic models for machine learning is a vast area in itself (independent of whether we are studying neural nets or not). We won't have time to go into all the details – take an advanced statistical learning course if you would like to learn more.]

Setup

Let us lay out the problem more precisely. In terms of symbols, instead of learning weights W that learn a discriminative function mapping of the form:

$$y = f_W(x)$$

we will instead imagine that the space of all x is endowed with some probability distribution $p(x)$. This may be a distribution that is without any conditions (e.g., all face images x are assigned high values of $p(x)$, and the set of all images that are not faces are assigned low values of $p(x)$). Or, this may be a *conditional* distribution $p(x; c)$. (Example: the condition c may denote hair color, and the set of all face images with that particular hair color c will be assigned higher probability versus the rest).

If there was some computationally easy way to represent the distribution $p(x)$, we could do several things:

- we could *sample* from this distribution. This would give us the ability to synthesize new data points.
- we could *evaluate* the likelihood of a given test data point (e.g. answering the question: does this image resemble a face image?)
- we could solve *optimization problems* (e.g. among all potential designs of handbags, find the ones that meet color and cost criteria)
- perhaps learn conditional relationships between different features

etc.

The question now becomes: how do we computationally represent the distribution $p(x)$? Modeling distributions (particularly in high-dimensional feature spaces) is not easy – this is called the *curse of dimensionality* — and the typical approach to resolve this is to parameterize the distribution in some way:

$$p(x) := p_\Theta(x)$$

and try to figure out the optimal parameters Θ (where we will define what "optimal" means later).

Classical machine learning and statistical approaches start off with simple parameterizations (such as Gaussians). Gaussians are nice in many ways: they are exactly characterized by their mean and (co)variance. We can draw samples easily from Gaussians. Central limit theorem = any set of independent samples averaged over sufficiently many draws resembles a Gaussian. Computationally, we like Gaussians.

Unfortunately, nature is far from being Gaussian! Real-world data is diverse; multi-modal; discontinuous; involves rare events; and so on, none of which Gaussians can handle very well.

Second attempt: Gaussian mixture models. These are better (multi-modal) but still not rich enough to capture real datasets very well.

Enter neural networks. We will start with some simple distribution (say a standard Gaussian) and call it $p(z)$. We will generate random samples from p; call it z. We will then pass z through a neural network:

$$x = f_\Theta(z)$$

parameterized by Θ. Therefore, the random variable x has a different distribution, say $p(x)$. By adjusting the weights we can (hopefully) deform $p(z)$ to obtain a $p(x)$ that matches any distribution we like. Here, z is called the *latent* variable (or sometimes the *code*), and f is called the *generative model* (or sometimes the *decoder*).

How are $p(x)$ and $p(z)$ linked? Let us for simplicity assume that f is one-to-one and invertible, i.e., $z = f_\Theta^{-1}(x)$. Then, we can use the *Change-of-Variables* formula for probability distributions. In one dimension, this is fairly intuitive to understand: in order to conserve mass, the area of the intervals must be the same, i.e., $p(x)dx = p(z)d(z)$ and hence the probability distributions must obey:

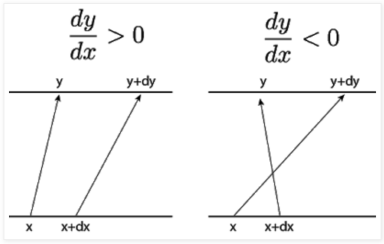

Figure 1: Change of variables

$$p(x) = p(z)|\frac{dx}{dz}|^{-1}$$

When both x and z have more than one dimension, we have to replace areas by (multi-dimensional) volumes and derivatives by partial derivatives. Fortunately, volumes correspond to determinants! Therefore, we can get an analogous formula by replacing the absolute value by the *determinant of the Jacobian of the mapping* $x = f(z)$:

$$p(x) = p(z)|\frac{\partial x}{\partial z}|^{-1}$$

This gives us a closed-form expression to evaluate any $p(x)$, given the forward mapping. However, note that for this formula to hold, the following conditions must be true:

- f must be one-to-one and easily invertible.
- f needs to be differentiable, i.e., the Jacobian must be well-defined.
- The determinant of the Jacobian must be easy to invert.

Reversible Models

As a warmup, a simple approach that ensures all of the above conditions are called *reversible models*. Recall the *residual* block that we discussed in the context of CNNs: this is similar. Residual blocks implement:

$$x = z + F_\Theta(z)$$

where F_Θ is some differentiable network that has equal input and output size. (You can use ReLUs too but strictly speaking we should use differentiable nonlinearities such as sigmoids). Typically, F_Θ is a dense shallow (single- or two-layer network).

Reversible models use the above block as follows. We will consider two *auxiliary random variables* u and v as the same size as x and z, and define two paths:

$$x = z + F_\Theta(u),$$
$$v = u.$$

The variable u is called an *additive coupling layer*. If you don't like adding an extra variable for memory reasons (say), you can just split your features into two halves and proceed.

The advantage of this model is that the inverse of this forward model is easy to calculate! Given any x and v, the inverse of this model is given by:

$$u = v,$$
$$z = x - F_\Theta(u).$$

What about the determinant of the Jacobian? Turns out that reversible blocks have very simple expressions for the determinant. For each layer, the Jacobian is of the form:

$$\begin{pmatrix} \frac{\partial x}{\partial z} & \frac{\partial x}{\partial u} \\ \frac{\partial v}{\partial z} & \frac{\partial v}{\partial u} \end{pmatrix} = \begin{pmatrix} I & \frac{\partial F_\theta}{\partial u} \\ 0 & I \end{pmatrix}$$

which is an upper-triangular matrix with diagonal equal to 1. Such matrices have *determinant equal to 1 always* (and such transformations are hence called "volume preserving"). In other words, each reversible block maps a set to another set of the same volume.

Having defined a single reversible block, we can now chain multiple such reversible blocks into a deeper architecture by alternating the roles of x and v. Let's say we have a second such block F_Ψ applied to v and u. Then, we get the following two-layer architecture:

$$x' = z + F_\Theta(u)$$
$$z' = u + F_\Psi(x')$$

(Exercise: can you compute the inverse of this two-layer block?)

Turns out that each such block is volume preserving, and hence the determinant of the overall Jacobian (no matter how many blocks we stack) are all equal to unity. We can think of each layer as *incrementally* changing the distribution until we arrive at the final result. Such a model that implements this type of incremental change is called a "flow" model. (The specific form above was called NICE – short for Nonlinear Independent Components Estimation).

We finally come to training this model. Different objective functions can be used: a common one is *maximum likelihood*: given a dataset of n samples $x_1, x_2, \ldots, x_n$ we optimize for the parameters that maximize the overall likelihood:

$$L(\Theta) = \prod_{i=1}^{n} p_X(x_i) = \prod_{i=1}^{n} p_Z(f^{-1}(x_i))$$

where p_Z is the base distribution. (Note that the Jacobian disappears.) In practice, sums are easier to optimize than products, and therefore we use the log-likelihood instead.

Normalizing Flows

Reversible blocks are nice from a compute standpoint, but have architectural limitations due to the volume preserving constraint.

Normalizing Flows (NF) generalize the above technique, and allow the mapping to be non-volume preseerving (NVP). The idea is to assume an arbitrary series of maps: $f_1, f_2, \ldots, f_L$ (where L is the depth), so that:

$$x = f_L \odot \ldots \odot f_2 \odot f_1(z).$$

Define $z_0 := z$ and z_i as the output of the i-th layer. Applying the change-of-variables formula to any intermediate layer, we have the distributional relationship:

$$\log p(z_i) = \log p(z_{i-1}) - \log |\det \frac{\partial z_i}{\partial z_{i-1}}|.$$

and recursing over i, we have the log likelihood:

$$\log p(x) = \log p(z) - \sum_{i=1}^{L} \log |\det \frac{\partial z_i}{\partial z_{i-1}}|.$$

This is a bit more complicated to evaluate, but in principle it can be done.

To make life simpler, in NF, we use the same principles as we did for reversible architectures:

- easy inverses for each layer
- easy Jacobian determinant

but this time, instead of creating an *additive* coupling layer u, we use an *affine* coupling layer:

$$\begin{aligned} x &= z \odot \exp(F_\Theta(u)) + F_\Psi(u), \\ v &= u. \end{aligned}$$

where F_θ and F_Ψ are trainable functions, and $\odot$ is applied component wise. The inverse of the affine coupling layer is simple:

$$\begin{aligned} u &= v, \\ z &= (x - F_\Psi(u)) \odot \exp(-F_\Theta(u)). \end{aligned}$$

Moreover, the Jacobian has the following structure:

$$J = \begin{pmatrix} \frac{\partial x}{\partial z} & \frac{\partial x}{\partial u} \\ \frac{\partial v}{\partial z} & \frac{\partial v}{\partial u} \end{pmatrix} = \begin{pmatrix} \text{diag}(\exp(F_\Theta(u))) & \frac{\partial F_\theta}{\partial u} \\ 0 & I \end{pmatrix}$$

which is an upper-triangular matrix, but with an easy-to-calculate determinant:

$$det(J) = \exp(\sum_{i=1}^{d} F_\theta^i(u)).$$

Extensions: Autoregressive models, PixelRNN, WaveNet, GLOW, etc

The above generative architectures are feedforward, dense, and useful for static structured data (such as images). For sequential data (such as music), we can develop similar models using RNN-type architectures.

The differences between the methods lie in the details, but the basic idea is that the features in the output x are not simultaneously generated (as in a feedforward network), but rather, generated one after the other. Moreover, since certain types of sequence data (such as voice or music) usually respect causality, the architectures are restricted to be *auto-regressive*, i.e., the probability distribution of *every* generated sample x is decomposed as:

$$P(x) = \pi_{i=1}^{d} p(x[i]|x[0:i])$$

(where we are abusing Python notation here). Various typical assumptions (e.g. Markov-ness) are made to simplify this, just as how we did for an RNN. But fundamentally, since there are d terms in the above product one would have to "unroll" the operation into a depth-d network, which can be rather challenging.

WaveNet, used for audio signal generation, reduces the depth in a smart way: it uses a technique called *dilated convolution* that effectively reduces the depth to $\log d$ by grouping together symbols and effectively using parallelism. We won't get into any further detail here.