Notes on "Glow"

Xavier Garcia

March 6, 2019

The purpose of these notes is to explain a few of the confusing parts of the paper "Glow".

1 Normalizing Flows

In this section, we discuss and define normalizing flows, as well as why we should consider them. We begin by first pointing out some of the limitations of popular methods of approximating distributions. Following this, we will define normalizing flows and the type considered in the paper.

1.1 Limitations of traditional generative models

Traditionally, generative models arise as a parametrized family of distributions by either modelling the density directly, or modeling the data distribution as the image of a Gaussian random variable. More precisely, suppose we have data $x_1, ..., x_N, \in \mathbb{R}^n$, which we assume is sampled from a random variable X. The most popular architectures assume that there exists a standard Gaussian $Z \sim \mathcal{N}(0, I_k)$ in $\mathbb{R}^k$ and a continuous function f such that $f(Z) \sim X$. While such a framework is excellent if we are only interested in sampling (see e.g. all the different GAN architectures), it falls apart if we are interested in more subtle, delicate questions, such as:

1. What is the support of X? i.e. Given a picture, what's the probability it is a cat picture?
2. Can we recover analytically tractible formulas for statistics of X? e.g. Can we compute $\mathbb{E}X$ without resorting to Monte Carlo?
3. What about conditional distributions?

These questions are trivially solved for Z, but generally impossible for $f(Z)$. It's thus interesting to consider when can we extend those nice properties of Z onto X. To wit, suppose $k = n$ and f is smooth and invertible. We can now try to reverse process.

$$\begin{aligned}
p_X(x_1,...,x_n) &= \frac{\partial^n}{\partial x_1,...,\partial x_n}\mathbb{P}(X_i \leq x_i \,\forall i) \\
&= \frac{\partial^n}{\partial x_1,...,\partial x_n}\mathbb{P}(f(Z)_i \leq x_i \,\forall i) \\
&= \frac{\partial^n}{\partial x_1,...,\partial x_n}\mathbb{P}(Z_i \leq f^{-1}(x)_i \,\forall i) \\
&= p_Z(f^{-1}(x)) \cdot |\det(Jf^{-1}(x))|
\end{aligned}$$

where Jf^{-1} is the Jacobian of f. Notice that this assumption is far away from the traditional setup, since we usually assume Z is a latent space with $k << n$. While the math seems nice, there are two problems with this setup:

1. We need f to be both smooth and invertible, with smooth inverse.
2. Computing the determinant is expensive.
3. Given these two and any X, the family of functions f_θ should be flexible enough such that there exists a $\theta := \theta(X)$ such that $f_\theta(Z) \approx X$

The key idea we'll undertake is that we will think of $f := f_\theta$ as being the composition of "simple" functions i.e.

$$f := g_M \circ ... \circ g_1$$

akin to how we define neural networks. For the ease of notation, we set $z_0 = z$ and $z_i = g_i(z_{i-1})$ To see why this is a reasonable idea in this context, we point out that the inverse of f has a nice formula in terms of the inverses of the g_i.

$$\begin{aligned}
f^{-1}(x) &= g_1^{-1} \circ ... \circ g_M^{-1}(x) \\
Jf^{-1}|_{f_M(z_0)} &= J(g_1^{-1} \circ ... \circ g_M^{-1}) \\
&= Jg_1^{-1}|_{z_{M-1}} \cdot ... \cdot Jg_M^{-1}|_{z_0}
\end{aligned}$$

where by $Jf|_y$ we mean the Jacobian of f evaluated at y. Additionally, we can also write the Jacobian of f in terms of the f_i:

$$Jf(z) = Jg_M|_{z_{M-1}} \cdot ... \cdot Jg_1|_{z_0}$$

Moreover, we can connect the Jacobian of the inverse of a function to the Jacobian of the function by the following formula: For any (z, x) such that $f(z) = x$, we have:

$$Jf^{-1}|_{f(z)} = (Jf)^{-1}|_x$$

Due to the emergence of products as well as the fact that we'll be interested in log-likelihood, we should consider taking logs:

$$\log |\det Jf|_z| = \sum_{i=1}^{M} \log \left|\det Jg_i|_{z_{i-1}}\right|$$

This suggests for simple functions with a large M, we should be able to approximate as many functions as we want, addressing 3). We call this method of approximating the distribution a normalizing flow.

2 Suitable functions

In this subsection, we look at functions which satisfy 1) and 2). The most natural transformations to look at are affine transformations, i.e. $x \mapsto Ax + b$ for some matrix A and some vector b. These are nice since the inverse is given by $y \mapsto A^{-1}(y - b)$ and the Jacobian is just A. The first two family of functions will consist of this type.

Given the setting of the paper consists of images, we shall work with the assumption that our samples x are actually images, i.e. $x \in \mathbb{R}^{w \times h \times c}$, where w stands for width, h for height, c for channels. Whenever we use two indices e,g, $x_{i,j}$, we mean indexing the w and h dimension, so $x_{i,j} \in \mathbb{R}^c$. If we use only a single index, e.g. x_k, then we are indexing by the channels $x_k \in R^{w \times h}$. This is pretty dumb notation, but it'll do for now. With this setting, the paper consists of three different types of simple functions.

2.1 Activation normalization

The first type of function we consider is actually one of the simplest possible. Namely, we normalize across channels i.e. for $s, b \in \mathbb{R}^c$, called scale and bias respectively, define the act-norm layer by

$$\text{act-norm}(x; s, b) = s \odot x + b$$

where $\odot$ corresponds to element-wise product. These are chosen as a proxy for batch normalization, as the author's model only allows for a batch size of one. The inverse is given by $y \mapsto (y - b)/s$, where the division in element-wise, and the Jacobian is given by

$$\det J\text{act-norm}(x; s, b) = hw \prod_{i=1}^{n} s_i$$

2.2 Invertible 1x1 convolutions

We could turn the act-norm function into a 1×1 convolution by summing over the values in the channel. Mathematically, this would look like the following:

$$s^T \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^{1 \times w \times h}$$

This would force some interaction between the values of x, giving some added flexibility. Unfortunately, this would also eliminate the invertibility of the operation, since we are compressing dimensions i.e. the operation would look like $(w, h, c) \mapsto (1, w, h)$. Thus, in order to recover the right dimension count, we consider c 1×1 convolutions $s_1, ..., s_c$ and stack them to get the following:

$$\left(\begin{array}{c} s_1^T \\ \hline \vdots \\ \hline s_c^T \end{array}\right) \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^{c \times w \times h}$$

If we set

$$W = \left(\begin{array}{c} s_1^T \\ \hline \vdots \\ \hline s_c^T \end{array}\right)$$

then as before, we can compute the inverse in the usual way by computing the matrix inverse, and similarly for the Jacobian.

Unfortunately, computing the inverse and the determinant of this Jacobian can be on the order of $\mathcal{O}(c^3)$. We can solve this problem by restricting our search to either upper or lower triangular matrices. These are particularly nice because the inverse of an upper (respectively lower) triangular matrix is also upper (respectively lower) triangular, hence making the inverse computation cleaner. Additionally, computing the determinant is just the product of the diagonal entries, reducing the cost to $\mathcal{O}(n)$. However, this would be quite drastic of a reduction of predictive power. To skirt around this issue, we can decompose W into it's LU decomposition:

$$W = PL(U + \text{diag}(s))$$

where P is a permutation matrix, L is a lower triangular matrix with ones on the diagonal and U is a upper triangular matrix with 0s in the diagonal. With this reduction, we have that $\log\det W = \sum_i \log s_i$, and hence much cheaper computation of order $\mathcal{O}(n)$. To avoid recomputing the LU decomposition, we shall instead fix P and consider L, diag(s) and U as the trainable parameters.

2.3 Affine Coupling Layer

Both of the previous family of functions are still, at their core, affine transformation. If we only considered compositions of those functions, we'd end up with an affine transformation at the end. One way to extend these functions would be to consider functions of the form

$$x \mapsto s(x)x + b(x).$$

This way x also controls the scale and bias. Unfortunately, this would severely limit the kind of s and b we can choose, since we need to be able to invert this and as well as have a tractible Jacobian. Notice that the previous functions were also of this form with $s(x) \equiv s$ for some constant s, similarly with b. The advantage of that formulation is that decoupling x from $s(x)$ and $b(x)$ makes the problem tractible, while having no decoupling makes it intractible. We shall settle for a partial decoupling. Suppose we break up x into two parts, say

$$x = \begin{pmatrix} \tilde{x}_1 \\ \tilde{x}_2 \end{pmatrix}$$

then we could look at the function $\tilde{x}_1 \mapsto s(\tilde{x}_2)\tilde{x}_1 + b(\tilde{x}_2)$. Notice that as a function $\tilde{x}_1$, this is an affine transformation, and hence enjoys the benefits that comes with that property. We extend this as a function of x, called Affine Coupling Layer (ACL), as follows:

$$\operatorname{acl}(x; s, b) : \begin{pmatrix} \tilde{x}_1 \\ \tilde{x}_2 \end{pmatrix} \mapsto \begin{pmatrix} s(\tilde{x}_2)\tilde{x}_1 + b(\tilde{x}_2) \\ \tilde{x}_2 \end{pmatrix}$$

Since $\tilde{x}_2$ is fixed, we can still invert this function without having to invert s or b, but we still get the benefits of non-linearity. The only thing left is to compute the Jacobian. At a glance, it looks like the fun stops here. After all, even though we managed to cheat our way into non-linearity, we are still going to have to differentiate s and b with respect to $\tilde{x}_2$, which should lead to a nasty Jacobian computation, as well as restrictions as to what kind of s and b we can consider to have a tractible problem. Nevertheless, we invoke the principle of optimism in the face of uncertainty and carry forward:

$$J\operatorname{acl}(x; s, b) = \begin{pmatrix} s(\tilde{x}_2) & \text{Bad News} \\ 0 & I \end{pmatrix}$$

where I is the identity matrix of the appropiate dimension and Bad News is the term involving derivatives of s and b. This term looks like it should spell out bad news for us. Nevertheless, a miracle happens:

$$\det J\operatorname{acl}(x; s, b) = \prod_i s_i(\tilde{x}_2).$$

This follows from the traditional matrix block determinant formula i.e. if A is $n \times n$,B is $n \times m$, C is $m \times n$ and D is $m \times m$, then we have:

$$\det \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \det(A - BD^{-1}C)\det(C)$$

Setting $A = s(\tilde{x}_2)$, $B =$ Bad News, $C = 0$, and $D = I$ yields the result. Incredibly, this frees us up to choose any function for s and b, so long as s is nonzero and preferably continuous.

For this paper, the three types of functions are composed one after another, forming one of the g_i from before.