Normalizing Flows II

Basic Introduction and Recap

- Generative model, unsupervised learning, applications wide: image generation, reinforcement learning, video generation, astrophysics etc.
- Used as an alternative to GANs, VAEs – exact log-likelihood evaluation, exact posterior inference, more stable training process
- Transformation of simple probability distribution into a more complex one using a series of invertible transformations, probability distribution changes as the change of variables formula:

$$\log p_x(x) = \log p_z(z_K) - \sum_{k=1}^{K} \log \left| \det \frac{\partial f_k(z_{k-1})}{\partial z_{k-1}} \right|$$

- The generative direction takes the simple, base distribution to the more complicated density, and the normalizing direction is given by the inverse of this, which 'flows' backwards
- More formal construction – measure theoretic definition of probability distributions

Applications

- Density Estimation and Sampling
  - Evaluation of the log likelihood requires evaluation of the Jacobian of *F*, however in order to sample we require evaluation of the generative direction, i.e. F^{-1}.
  - Computation of the inverse is often complicated, so for density estimation we often model the flow in the normalizing direction
  - Usually MLE is used, other forms of estimation have been used including adversarial losses
- Variational Inference
  - Latent variable models – computation of the posterior is intractable in practice
  - Use the ELBO – calculation of gradients in gradient descent for calculation of this is tricky
  - Instead, can use a 'reparameterization trick' - the issue is calculating $\nabla_\theta E_{q(y|x,\theta)}[h(y)]$, so with normalizing flows parameterise $q(y|x, \theta) = p_Y(y|\theta)$ where a flow *g* with parameters θ is used, such that $p_Z(z)$ is not a function of the parameters, so:

$$\mathbb{E}_{p_Y(y|\theta)}[h(y)] = \mathbb{E}_{p_Z(z)}[h(g(z|\theta))]$$

  - and the RHS is an easier function to calculate gradients of

Methods

- We want to choose transformations that are i) invertible, ii) complex enough to match our distribution but also iii) computationally efficient, in terms of calculating the transformation, its inverse, and the Jacobian
- 1. Elementwise flows – apply a bijective function to every element of the variable – activation function
- 2. Linear flows - $\mathbf{g}(x) = \mathbf{A}\mathbf{x} + \mathbf{b}$, where **A** is a matrix – allows for correlation between the variables, however Jacobian is det(**A**), which can become expensive in high dimensions – can use several appropriate transformations to make this easier
  - Diagonal – diagonal matrix is v easy to calculate the det of, but no correlation between the variables, only elementwise
  - Triangular – equally easy to get det, more expressive but inversion more expensive
  - Orthogonal – guaranteed to be det +/-1, use the fact that we can get orthogonal matrix as a product of reflections, general form of reflection matrix
  - Factorisation – **A = PLU**, permutation x lower triangular x upper triangular, **L** is only 1's on the diagonal – det **A** is the product of the elements on the diagonal of **U** – **P** is difficult to parameterise, but fix random **P** – limits transformation expressiveness, use QR decomposition

Normalizing Flows II

- Convolution – masked autoregressive convolutions

- 3. Planar and Radial flows
  - Planar – expand and contract the distribution along certain specific distributions, of the form $\mathbf{g(x)} = \mathbf{x} + \mathbf{u}h(\mathbf{w}^T\mathbf{x} + b)$, Jacobian determinant is $1 + h'(\mathbf{w}^T\mathbf{x} + b)\mathbf{u}^T\mathbf{w}$ – problem is that inverse has no closed form, may also not exist depending on choice of **u, w,** h
  - Radial – flow changes to the form $\mathbf{g(x)} = \mathbf{x} + \frac{\beta}{\alpha + |x - x_0|}(x - x_0)$ – same idea, now around a specific point, also has the same problems
- 4. Coupling and Autoregressive Flows
  - Coupling - disjoint partition of the data and a bijection parameterised by θ, then $y_A = h(x_A; \Theta(x_B)); \; y_B = x_B$ is the flow, where $\Theta(x_B)$ is the conditioner – this can be arbitrarily complex, and the inverse and Jacobian is easily calculable
  - Often a NN used for the conditioner – can sometimes be constant. Question about how to partition the data – random partition or something more specific?
  - Autoregressive flows – non-linear generalization of multiplication by a triangular matrix, so we get $y_t = h(x_t; \Theta_t(x_{1:t-1}))$, where the Θ_t are conditioners, functions from R^{t-1} to the set of all parameters, Θ_1 is a constant – Jacobian is triangular by construction, so the determinant is just $det\, J = \prod_{t=1}^{D} \frac{\partial y_t}{\partial x_t}$
  - With appropriate masks, one can calculate the flow in one pass directly, this is called a Masked Autoregressive Flow (MAF) – can also have Inverse Autoregressive Flow (IAF)
  - Universality – several autoregressive flows have a universality property, i.e. they can learn any target distribution given sufficient data (proof – lots of functional analysis)
  - Coupling flows – flows commonly chosen include additive coupling, nonlinear squared flow, continuous mixture CDFs and splines (look up forms)
  - Neural Autoregressive Flows – coupling function is modelled by a NN, will be bijective if all weights are positive and all activation functions are strictly monotone
- 5. Residual Flows
  - $\mathbf{g(x)} = \mathbf{x} + F(\mathbf{x})$ – **g** is the residual connection; *F* is the residual block which is a feedforward neural network
  - Jacobian determinants of these networks difficult to calculate directly
  - Invertibility has a sufficient condition, which is related to the Lipschitz constant of *F*
- 6. Infinitesimal Flows
  - Model *F* with ODEs continuously rather than the discrete version
  - Can also use SDEs – modelled a la Brownian motion
  - Uses MCMC methods for modelling diffusion

<u>Pyro</u>

- Pytorch library that contains a comprehensive library of learnable distributions, libraries to import are:
- `import pyro`
- `import pyro.distributions as dist`
- `import pyro.distributions.transforms as T`
- The transformed distribution class transforms a base distribution of noise through a variety of transformations
- Learnable transformation – transform module, trainable parameters
- Spline transformation, multivariate is spline_coupling – there is also a conditional spline helper function