Deep Learning - Foundations and Concepts

Chapter 18. Normalizing Flows

nonlineark@github

April 19, 2025

Outline

Normalizing flows

- We define a distribution $p_z(z)$ over a latent variable z.
- And we use a deep neural network architecture to define an invertible function $x = f(z; w)$ that transforms the latent space into the data space.
- We can ensure that the overall function is invertible if we make each layer of the network invertible:
  - $x = f^A(f^B(f^C(z)))$.
  - $z = g^C(g^B(g^A(x)))$.

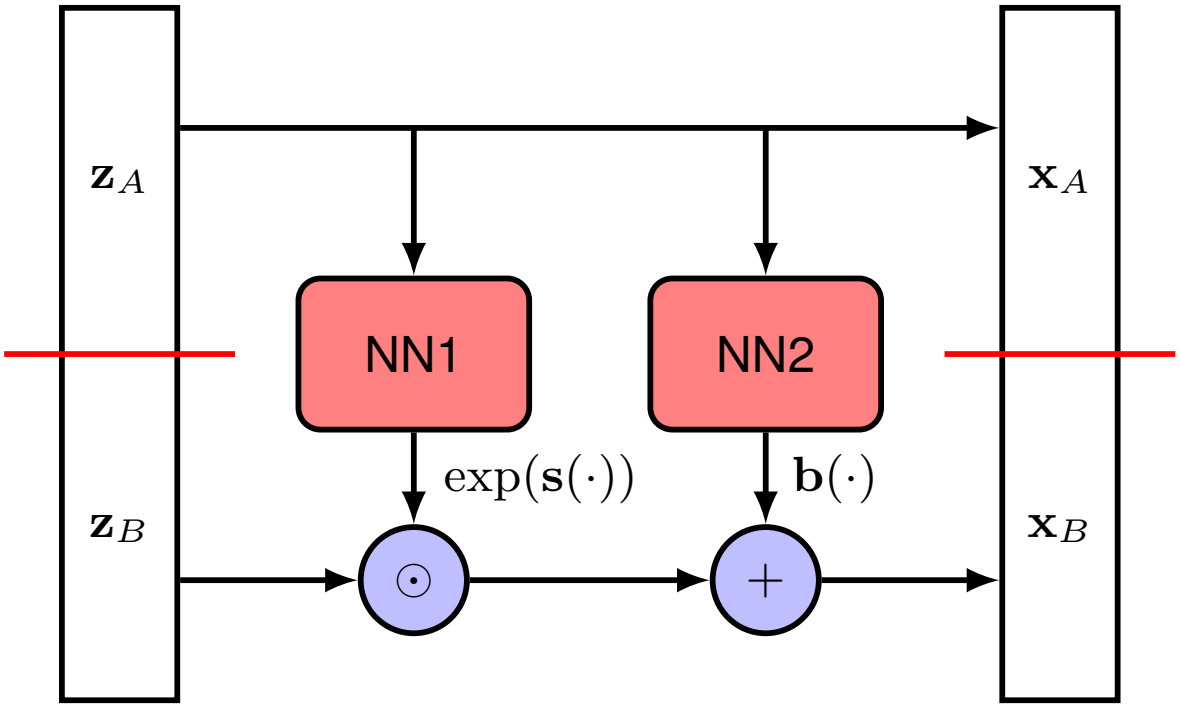
Normalizing flows

This approach to modelling a flexible distribution is called a normalizing flow because:

- The transformation of a probability distribution through a sequence of mappings is somewhat analogous to the flow of a fluid.
- The effect of the inverse mapping is to transform the complex data distribution into a normalized form, typically a Gaussian distribution.

Real NVP

Figure: A single layer of the real NVP normalizing flow model



Real NVP

Real-valued non-volume-preserving (real NVP) partitions the latent variable $z \in \mathbb{R}^D$ and the output vector $x \in \mathbb{R}^D$ into two parts: $z = (z_A, z_B)$ and $x = (x_A, x_B)$, where $z_A, x_A \in \mathbb{R}^d$. The transformation function is given by:

$$
\begin{aligned}
x_A &= z_A \\
x_B &= \exp(s(z_A; w)) \odot z_B + b(z_A; w)
\end{aligned}
$$

where $s(z_A; w)$ and $b(z_A; w)$ are the real-valued outputs of neural networks, and $\odot$ denotes element-wise multiplication of the two vectors.

Real NVP

The overall transformation is easily invertible:

$$
\begin{aligned}
z_A &= x_A \\
z_B &= \exp(-s(x_A; w)) \odot (x_B - b(x_A; w))
\end{aligned}
$$

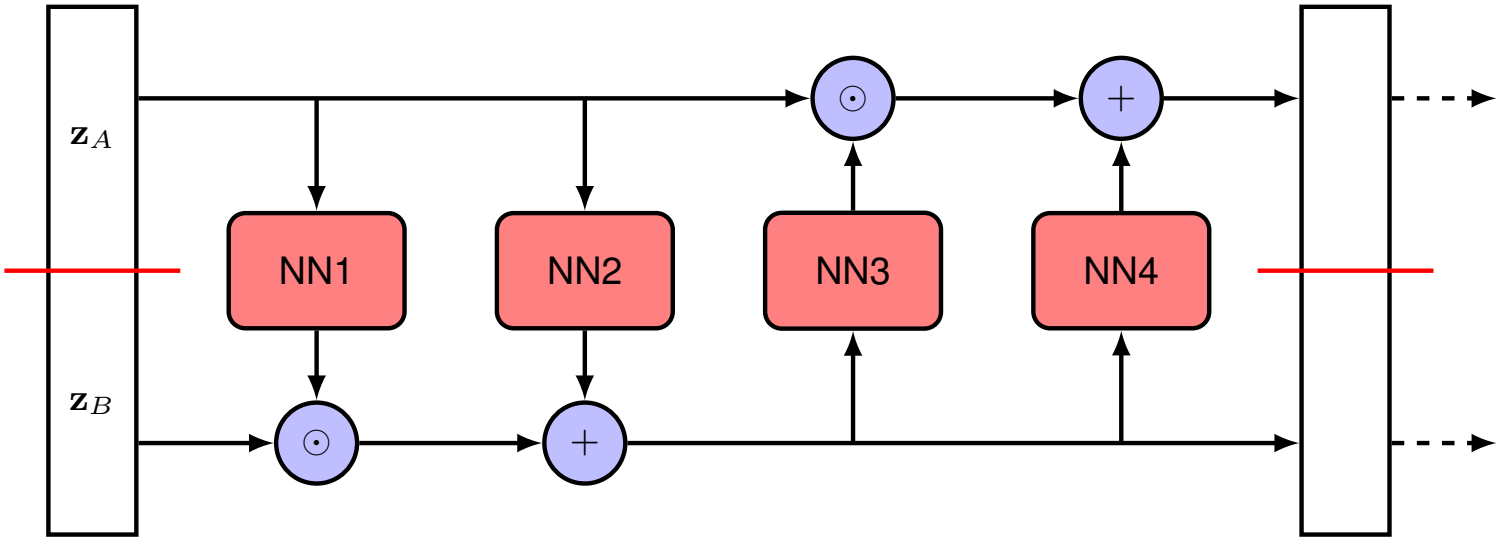
The Jacobian matrix of this inverse mapping is given by:

$$
J = \begin{pmatrix} I_d & 0 \\ \frac{\partial z_B}{\partial x_A} & \operatorname{diag}(\exp(-s(x_A; w))) \end{pmatrix}
$$

We see that the Jacobian matrix is a lower triangular matrix, and its determinant is simply given by the product of the elements of $\exp(-s(x_A; w))$.

Real NVP

Figure: A double-layer structure where the roles of z_A and z_B are reversed in the second layer



Coupling flows

The real NVP model belongs to a broad class of normalizing flows called coupling flows:

$$x_A = z_A$$
$$x_B = h(z_B, g(z_A; w))$$

where:

- $h(z_B, g)$ is a function of z_B that is efficiently invertible for any given value of g and is called the coupling function.
- The function $g(z_A; w)$ is called a conditioner and is typically represented by a neural network.

Masked autoregressive flows

In masked autoregressive flows (MAFs), the transformation function from the latent space into the data space is given by:

$$x_d = h(z_d, g_d(x_{1:d-1}; w_d))$$

where:

- h is the coupling function, which is chosen to be easily invertible with respect to z_d.
- g_d is the conditioner, which is typically represented by a deep neural network.
- $x_{1:d-1}$ denotes $x_1, \ldots, x_{d-1}$.

Masked autoregressive flows

The overall transformation is easily invertible:

$$z_d = h^{-1}(x_d, g_d(x_{1:d-1}; w_d))$$

- The inverse can be performed efficiently on modern hardware since $z_1, \ldots, z_D$ can be evaluated in parallel.
- The Jocobian matrix is a lower-triangular matrix whose determinant can also be evaluated efficiently.
- However, sampling from this model must be done sequentially because the values of $x_1, \ldots, x_{d-1}$ must be evaluated before x_d can be computed.

Inverse autoregressive flows

To avoid this inefficient sampling, we can instead define an inverse autoregressive flow (IAF) given by:

$$x_d = h(z_d, \tilde{g}_d(z_{1:d-1}; w_d))$$

The inverse is given by:

$$z_d = h^{-1}(x_d, \tilde{g}_d(z_{1:d-1}; w_d))$$

We see that:

- Sampling is now efficient since the evaluation of the elements $x_1, \ldots, x_D$ can be performed in parallel.
- The Jocobian matrix is again a lower-triangular matrix whose determinant can be evaluated efficiently.
- The inverse calculations are intrinsically sequential and slow.

Neural differential equations

Consider a residual network:

$$z^{(t+1)} = z^{(t)} + f(z^{(t)}; w)$$

where t labels the layers in the network. Now we consider a small increment ϵ in the layer variable t:

$$z^{(t+\epsilon)} = z^{(t)} + \epsilon f(z^{(t)}; w)$$

Taking the limit $\epsilon \to 0$, the hidden-unit activation vector becomes a function $z(t)$ of a continous variable t, and we can express the evolution of this vector through the network as a differential equation:

$$\frac{\mathrm{d}z(t)}{\mathrm{d}t} = f(z(t); w)$$

where t is often referred to as time.

Neural differential equations

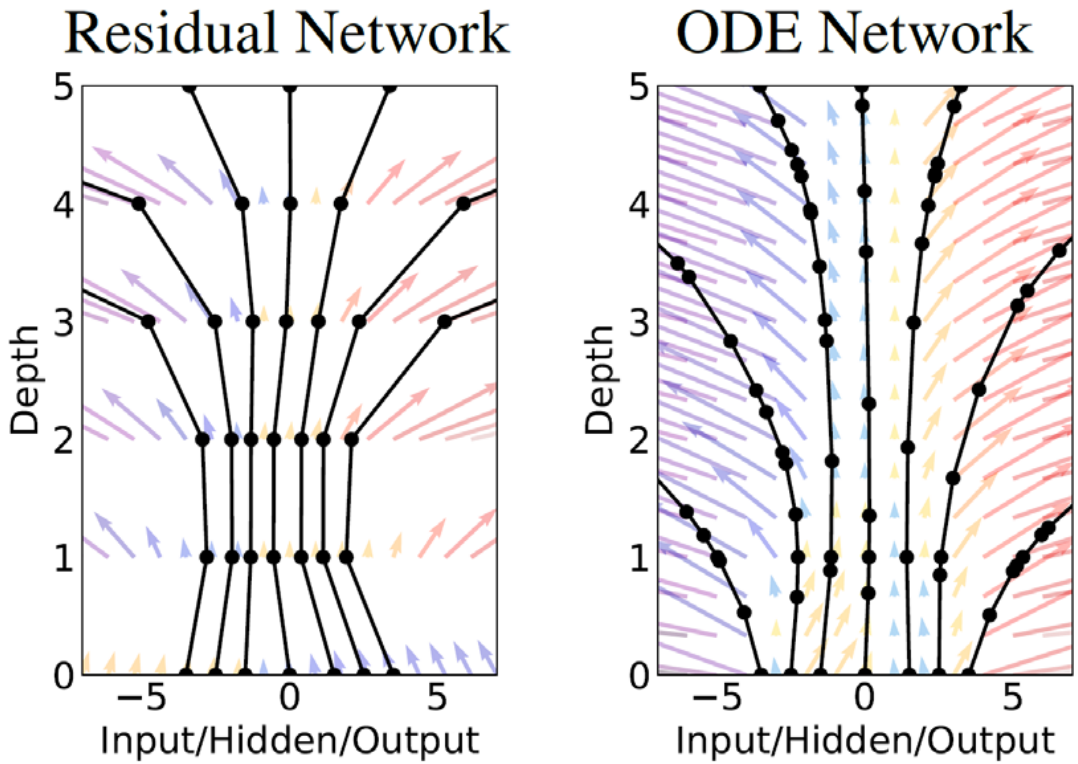
If we denote the input to the network by the vector $z(0)$, then the output $z(T)$ is obtained by integration of the differential equation:

$$z(T) = z(0) + \int_0^T f(z(t); w)\mathrm{d}t$$

This integral can be evaluated using standard numerical integration packages.

Neural differential equations

Figure: Comparison of a conventional layered network with a neural differential equation



Neural ODE backpropagation

Let us assume that we are given a data set comprising values of the input vector $z(0)$ along with an associated output target vector and a loss function L that depends on the output vector $z(T)$.

One approach to training a neural ODE would be to use automatic differentiation to differentiate through all of the operations performed by the ODE solver during the forward pass. Although this is straightforward to do, it is costly from a memory perspective and is not optimal in terms of controlling numerical error.

Neural ODE backpropagation

For any two time values $0 \leq t_1 \leq t_2 \leq T$ and a network parameter vector w, we define a map $g_w^{t_1:t_2} : \mathbb{R}^D \to \mathbb{R}^D$, also known as the phase flow map of our neural ODE:

- There is a solution for our neural ODE whose graph passes through the point (t_1, z_1) when the network parameter vector is w. Let's denote this solution by $\phi(t)$.
- Then $g_w^{t_1:t_2}(z_1)$ is given by the value of this particular solution at t_2: $g_w^{t_1:t_2}(z_1) = \phi(t_2)$.

Neural ODE backpropagation

Now our training objective is defined as:

$$L(z_T) = L(g_w^{0:T}(z_0))$$

It can be shown that:

$$\frac{\partial g_w^{0:T}(z(0))}{\partial w} = \int_0^T \frac{\partial g_w^{t:T}(z(t))}{\partial z} \frac{\partial f(z(t);w)}{\partial w} \mathrm{d}t$$

$$\frac{\partial L(w)}{\partial w} = \int_0^T \left(\frac{\partial L(z(T))}{\partial z_T} \frac{\partial g_w^{t:T}(z(t))}{\partial z}\right) \frac{\partial f(z(t);w)}{\partial w} \mathrm{d}t$$

$$= \int_0^T \frac{\partial (L \circ g_w^{t:T})(z(t))}{\partial z} \frac{\partial f(z(t);w)}{\partial w} \mathrm{d}t$$

Neural ODE backpropagation

Let $a(t) = \frac{\partial (L \circ g_w^{t:T})(z(t))}{\partial z}$, it can be shown that:

$$\frac{\mathrm{d}a(t)}{\mathrm{d}t} = -a(t)\frac{\partial f(z(t); w)}{\partial z}$$

This is called the adjoint equation. The initial condition is given by:

$$a(T) = \frac{\partial L(z(T))}{\partial z_T}$$

Further references.

Neural ODE flows

If we define a base distribution over the input vector $p(z(0))$ then the neural ODE propagates this forward through time to give a distribution $p(z(t))$ for each value of t, leading to a distribution over the output vector $p(z(T))$. It can be shown that the transformation for the density can be evaluated by integrating a differential equation given by:

$$\frac{\mathrm{d}\log p(z(t))}{\mathrm{d}t} = -\mathrm{tr}\left(\frac{\partial f(z(t); w)}{\partial z}\right)$$

Further references.

Neural ODE flows

Figure: Illustration of a continuous normalizing flow