Statistic

for machine learning

Tran Trong Khiem

AI lab tranning

2024/05/29

Jacobian Matrix

Jacobian Matrix:

- The Jacobian matrix J of a function $\mathbf{F}: \mathbb{R}^n \to \mathbb{R}^m$ is given by:

$$J = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdots & \frac{\partial F_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1} & \frac{\partial F_m}{\partial x_2} & \cdots & \frac{\partial F_m}{\partial x_n} \end{bmatrix}$$

Change of Variable Theorem:

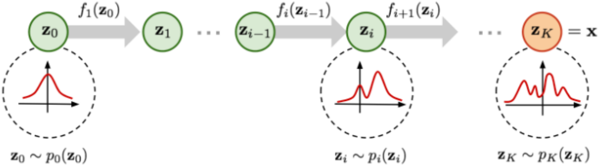
- Given a random variable $z \sim \pi(z)$, a mapping 1-1 function $x = f(z)$
- $p(x) = \pi(z)|\frac{dz}{dx}| = \pi(f^{-1}(x))|(f^{-1})'(x))|$
- The multivariable version has a similar format:

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

What is Normalizing Flows?

Normalizing Flow:

- **Goal**: for **better and more powerful** distribution approximation.
- Transforms a **simple distribution** into a **complex one**
  - by applying a **sequence** of **invertible transformation** functions.



- $z_{i-1} \sim p_{i-1}(z_{i-1})$, $z_i = f_i(z_{i-1})$, we have :
  $p_i(z_i) = p_{i-1}(f_i^{-1}(z_i)) \left| \det \frac{df_i^{-1}}{dz_i} \right|$
- $\log p_i(z_i) = \log p_{i-1}(z_{i-1}) - \log \left| \det \frac{df_i}{dz_{i-1}} \right|$

What is Normalizing Flows?

Inverse function theorem : If $y = f(x)$ and $x = f^{-}(y)$, we have :

- $\frac{df^{-1}(y)}{dy} = \frac{dx}{dy} = (\frac{dy}{dx})^{-1} = (\frac{df(x)}{dx})^{-1}$

Jacobians of invertible function: $\det(M^{-1}) = (\det(M))^{-1}$
We have:

- $x = z_k = f_k \circ f_{k-1} \circ \ldots \circ f_1(z_0)$
- $\log(p(x)) = \log \pi_k(z_k) = \log \pi_0(z_0) - \sum_{i=1}^{K} \log |\det \frac{df_i}{dz_{i-1}}|$
- $z_i = f_i(z_{i-1})$, tranformation function f_i should satisfy :
  - It is **easily invertible.**
  - Its **Jacobian determinant** is **easy to compute.**
- **Loss function:**

$$\mathcal{L}(\mathcal{D}) = -\frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} \log p(x)$$

RealNVP

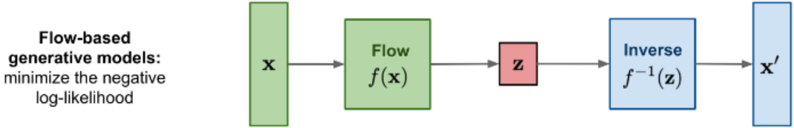
Real-valued Non-Volume Preserving:

- model implements a **normalizing flow.**
- stacking a sequence of **invertible bijective transformation** functions.

In each bijection $f : x \to y$, affine coupling layer

- The first d dimensions **stay same.** : $y_{1:d} = x_{1:d}$
- $y_{d+1:D} = x_{d+1:D} \circ exp(s(x_{1:d})) + t(x_{1:d})$
  - s,t are scale and translation functions map $R^d \to R^{D-d}$
- **Condition 1: "It is easily invertible."**
  - $y_{1:d} = x_{1:d}$
  - $y_{d+1,D} = (x_{d+1:D} - t(x_{1:D})) \circ exp(-s(x_{1:d}))$
- **Condition 2: "Its Jacobian determinant is easy to compute."**
  - $\det J = \exp(\sum_{j=1}^{D-d} s(x_{1:d})_j)$

RealNVP



Pos:

- compute f^{-1} **does not require** computing s^{-1} and t^{-1}
- computing the **Jacobian of f does not involve** computing the Jacobian of s and t.
  - s and t can be modeled by **deep neural networks**.

Batch normalization:

- **goal**: improve the **propagation of training signal**.
- use deep **residual networks** with batch normalization and weight normalization s and t.

NICE

Non-linear Independent Component Estimation:

- Affine coupling layer without the scale term:
  - $y_{1:d} = x_{1:d}$
  - $y_{d+1:D} = x_{d+1:D} + m(x_{1:d})$
  - m is an **arbitrarily complex function**(e.g : ReLU MLP)

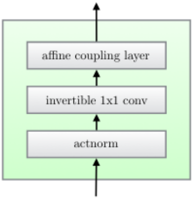
Learn bijective transformations of continuous distribution

- **Goal**: **learning a probability density** from a parametric family of densities $p_\theta, \theta \in \Theta$, have N samples from dataset $\mathcal{D}$, $x \in R^D$
- $\log p_X(x) = \sum_{d=1}^{D} \log(p_{Hd}(f_d(x))) + \log(|\det(\frac{\partial f(x)}{\partial x})|)$
  - $p_H(h)$ is the prior distribution.
  - Where $f(x) = (f_d(x))_{d \leq D}$

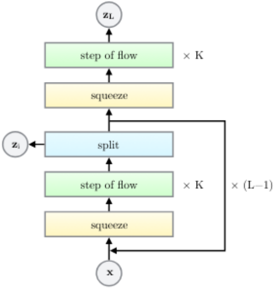
Glow

Proposed Generative Flow:

- consists of a **series of steps of flow**
- Each step of flow consists
  - actnorm
  - invertible 1x1 convolution
  - coupling layer



(a) One step of our flow.



(b) Multi-scale architecture (Dinh et al., 2016).

Actnorm

Problem:

- **Variance** of **activations noise** added by **batch normalization**
  - **inversely proportional** to minibatch **size** per GPU.
- performance is **degraded** for **small per-PU minibatch size**

Actnorm layer (for activation normalizaton):

- performs an **affine transformation** of the **activations.**
  - using a **scale and bias parameter** per channel
  - similar to **batch normalization.**
  - **scale and bias** are treated as regular **trainable parameters.**

Description	Function	Reverse Function	Log-determinant
Actnorm. See Section 3.1.	$\forall i,j : \mathbf{y}_{i,j} = \mathbf{s} \odot \mathbf{x}_{i,j} + \mathbf{b}$	$\forall i,j : \mathbf{x}_{i,j} = (\mathbf{y}_{i,j} - \mathbf{b})/\mathbf{s}$	$h \cdot w \cdot \texttt{sum}(\log \lvert\mathbf{s}\rvert)$

Invertible 1x1 convolution

Nice : proposed a flow containing the equivalent of a **permutation** that **reverses** the ordering of the channels.

- **Replace** this fixed **permutation** with a (learned) **invertible** 1x1 convolution.
  - a 1×1 convolution with an **equal number of input and output** channels is a generalization of a **permutation operation**.
- weight matrix is initialized as a random rotation matrix.
- Input: $h \times \omega \times c$ tensor H. W matrix has size $c \times c$.
  - Each element $x_{ij}(i = 1, ..., h, j = 1, ..., \omega) \in H$ is a vector of c channels.
  - Output: $y_{ij} = Wx_{ij}$ so that $\frac{\partial y_{ij}}{\partial x_{ij}} = W$
  - $\log|\det \frac{\partial Conv2D(H,W)}{\partial H}| = \log(|\det W|^{h*w}|) = (h*w)\log(|\det W|)$
  - LU Decomposition. The cost of computing $\det(W)$ can be reduced from $O(c^3)$ to $O(c)$.

Affine coupling layer

- The design is same as in RealNVP.

Description	Function	Reverse Function	Log-determinant
Actnorm. See Section 3.1.	$\forall i, j: \mathbf{y}_{i,j} = \mathbf{s} \odot \mathbf{x}_{i,j} + \mathbf{b}$	$\forall i, j: \mathbf{x}_{i,j} = (\mathbf{y}_{i,j} - \mathbf{b})/\mathbf{s}$	$h \cdot w \cdot \mathtt{sum}(\log \lvert\mathbf{s}\rvert)$
Invertible 1×1 convolution. $\mathbf{W}: [c \times c]$. See Section 3.2.	$\forall i, j: \mathbf{y}_{i,j} = \mathbf{W}\mathbf{x}_{i,j}$	$\forall i, j: \mathbf{x}_{i,j} = \mathbf{W}^{-1}\mathbf{y}_{i,j}$	$h \cdot w \cdot \log \lvert\det(\mathbf{W})\rvert$ or $h \cdot w \cdot \mathtt{sum}(\log \lvert\mathbf{s}\rvert)$ (see eq. (10))
Affine coupling layer. See Section 3.3 and (Dinh et al., 2014)	$\mathbf{x}_a, \mathbf{x}_b = \mathtt{split}(\mathbf{x})$ $(\log \mathbf{s}, \mathbf{t}) = \mathtt{NN}(\mathbf{x}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{y}_a = \mathbf{s} \odot \mathbf{x}_a + \mathbf{t}$ $\mathbf{y}_b = \mathbf{x}_b$ $\mathbf{y} = \mathtt{concat}(\mathbf{y}_a, \mathbf{y}_b)$	$\mathbf{y}_a, \mathbf{y}_b = \mathtt{split}(\mathbf{y})$ $(\log \mathbf{s}, \mathbf{t}) = \mathtt{NN}(\mathbf{y}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{x}_a = (\mathbf{y}_a - \mathbf{t})/\mathbf{s}$ $\mathbf{x}_b = \mathbf{y}_b$ $\mathbf{x} = \mathtt{concat}(\mathbf{x}_a, \mathbf{x}_b)$	$\mathtt{sum}(\log(\lvert\mathbf{s}\rvert))$

Models with Autoregressive Flows

The **autoregressive** constraint is a way to **model sequential data**.

- $x = [x_1, ..., x_D]$, each **output** only **depends** on the data observed in the **past**, but not on the **future ones**.
- $p(x) = \prod_{i=1}^{D} p(x_i | x_1 ... x_{i-1}) = \prod_{i=1}^{D} p(x_i | x_{1:i-1})$

If a **flow transformation** in a **normalizing flow** is framed as an **autoregressive** model.

- **each dimension in a vector** variable is **conditioned** on the **previous dimensions**.
- this is an **autoregressive flow**.

MADE

Masked Autoencoder for Distribution Estimation:

- **specially designed architecture** to enforce the **autoregressive property** in the autoencoder efficiently.
- using an **autoencoder** to **predict the conditional** probabilities.

Autoencoders :

- We have dataset $\{x^{(t)}\}_{t=1}^{T}$, each x is D dimentions, dimention x_d belong in $\{0, 1\}$
- **Goal**: learn **hidden representations** of the inputs.
    - **statistical structure** of the **distribution** that generated them.
    - attempts to learn a **feed-forward, hidden representation** h(x) of x.
    - from $h(x)$, can obtain a **recontruction** $\hat{x}$,which is as **close as possible to x.**
    - $h(x) = g(b + Wx)$
    - $\hat{x} = sigm(c + Vh(x))$

MADE(cont.)

Autoencoders:

- W and V are matrices, b and c are vectors, g is **non-linear activation** function, $sigm(x) = \frac{1}{1+\exp(-x)}$
- W is present **connections** from the input to the **hidden layer**.
- V represents the **connections** from the **hidden** to the output.
- The cross-entropy loss:

$$l(x) = -\sum_{d=1}^{D}(x_d \log(\hat{x}_d) + (1 - x_d)\log(1 - \hat{x}_d))$$

- **main disadvantage**: The **representation** it learns can be **trivial**.
  - hidden units can each learn to **"copy"** a single input dimension.

MADE(cont.)

Masked Autoencoders

- **Goal**: **modify** the **autoencoder** so as to satisfy the **autoregressive property**.
  - $\hat{x}_d$ must depend only on the preceding inputs $x_{<d}$
  - no **computational path** between output $\hat{x}_d$ and input unit $x_d, \ldots, x_D$
  - each of **computational** paths, at least one connection (in matrix W or V) must be 0.
- **Zeroing connections** : elementwise -multiply each W and V a binary **mask matrix**.
  - $h(x) = g(b + (W \circ M^W)x)$
  - $\hat{x} = sigm(c + (V \circ M^V)h(x))$
  - M^W and M^V are the masks for W and V.
  - Assign $1 \le m(k) \le D - 1$ for the k^{th} hidden unit's number.
    - maximum number of input units to which it can be connected.
    - Disallow $m(k) = 0$ or $m(k) = D$.

MADE(cont.)

Masked Autoencoders:

- We need to encode that the d^{th} output unit only connect to $x_{<d}$:

$$M^{W}_{k,d} = \begin{cases} 1 & \text{if } m(k) \geq d, \\ 0 & \text{otherwise}, \end{cases} \tag{1}$$

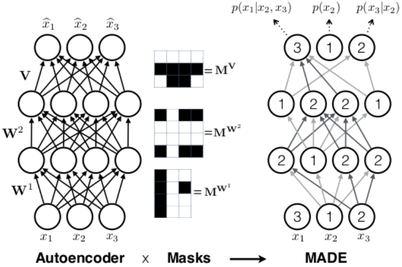
  for $d \in \{1, \ldots, D\}$ and $k \in \{1, \ldots, K\}$

- The output weights can only connect the d^{th} output to hidden units with $m(k) < d$

$$M^{V}_{d,k} = \begin{cases} 1 & \text{if } d > m(k) \\ 0 & \text{otherwise} \end{cases}$$

- Advantage :
  - Naturally generalizes to **deep architectures**.

Deep MADE



One can generalize this rule to any layer l, as follows:

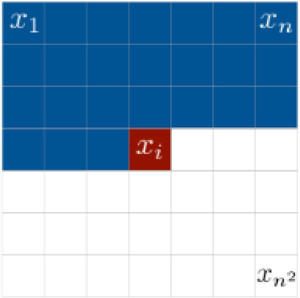
$$M^{(l)}_{k',k} = \begin{cases} 1 & \text{if } m_l(k') \geq m_{l-1}(k) \\ 0 & \text{otherwise} \end{cases}$$

We need to use the **connectivity constraints** of the last hidden layer $m_L(k)$ instead of the first:

$$M^{V}_{d,k} = \begin{cases} 1 & \text{if } d > m_L(k) \\ 0 & \text{otherwise} \end{cases}$$

PixelRNN

- **PixelRNN** is a deep generative model for images.
  - The image is **generated one pixel at a time**.
  - Each **new pixel** is **sampled conditional** on the **pixels** that have been **seen before**.
- Let's consider an image of size $n \times n$, $x = \{x_1, x_2, ... x_{n^2}\}$, the model starts **generating pixels** from the **top left corner** :



Model

Goal:

- **estimate** a **distribution over natural images**
- can be used to **tractably compute** the likelihood of images and to **generate new ones**.

Generating an Image Pixel by Pixel:

- **Goal**: assign a probability p(x) to each image x formed of $n \times n$ pixels.

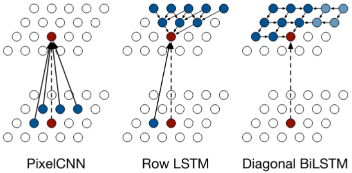
$$p(x) = \prod_{i=1}^{n^2} p(x_i \mid x_1, \ldots, x_{i-1})$$

- Each pixel x_i is in turn **jointly determined** by **three values**.
  - one for each of the color channels Red, Green and Blue(RGB).
  - $p(x_{i,R} \mid x_{<i})\, p(x_{i,G} \mid x_{<i}, x_i, R)\, p(x_{i,B} \mid x_{<i}, x_{i,R}, x_{i,G})$
- Pixels as **Discrete Variables** : each channel variable $x_{i,*}$ takes one of 256 distinct values.

Pixel Recurrent Neural Networks

Row LSTM:

- **processes** the image **row by row** from top to bottom.
  - computing features for a whole row **at once**
  - the computation is performed with a **one-dimensional convolution**



- For a pixel x_i the layer captures a **roughly triangular context** above the pixel.
  - The kernel of the **one-dimensional convolution** has size $k \times 1$ where $k \geq 3$.
  - the **larger the value of k** the **broader the context** that is captured.

Pixel Recurrent Neural Networks

Row LSTM

- The hidden and cell states h_i and c_i are obtained as follows:

$$[o_i, f_i, i_i, g_i] = \sigma(K^{ss} * h_{i-1} + K^{is} * x_i)$$

$$c_i = f_i \odot c_{i-1} + i_i \odot g_i$$

$$h_i = o_i \odot \tanh(c_i)$$

- x_i of size $h \times n \times 1$ is row i of the input map.
  - $*$ represents the **convolution operation**.
  - $\odot$ element-wise multiplication.
  - The weights K^{ss} is the kernel weights for the **state-to-state**.
  - The weights K^{is} is the kernel weights for the **input-to-state**.

Pixel Recurrent Neural Networks

Diagonal BiLSTM

- **designed** to both **parallelize** the computation.
  - capture the **entire available context** for any image size.
  - **skew** the **input map** into a space that makes it easy to **apply convolutions** along diagonals.
  - **skewing operation** offsets each row of the input map by one position with respect to the previous row.
  - results in a map of size $n \times (2n - 1)$
  - The **input-to-state** component is simply a 1×1 convolution K^{is}