16/1/2026

AI Factory

Generative and Self-Supervised ML

Flow Models

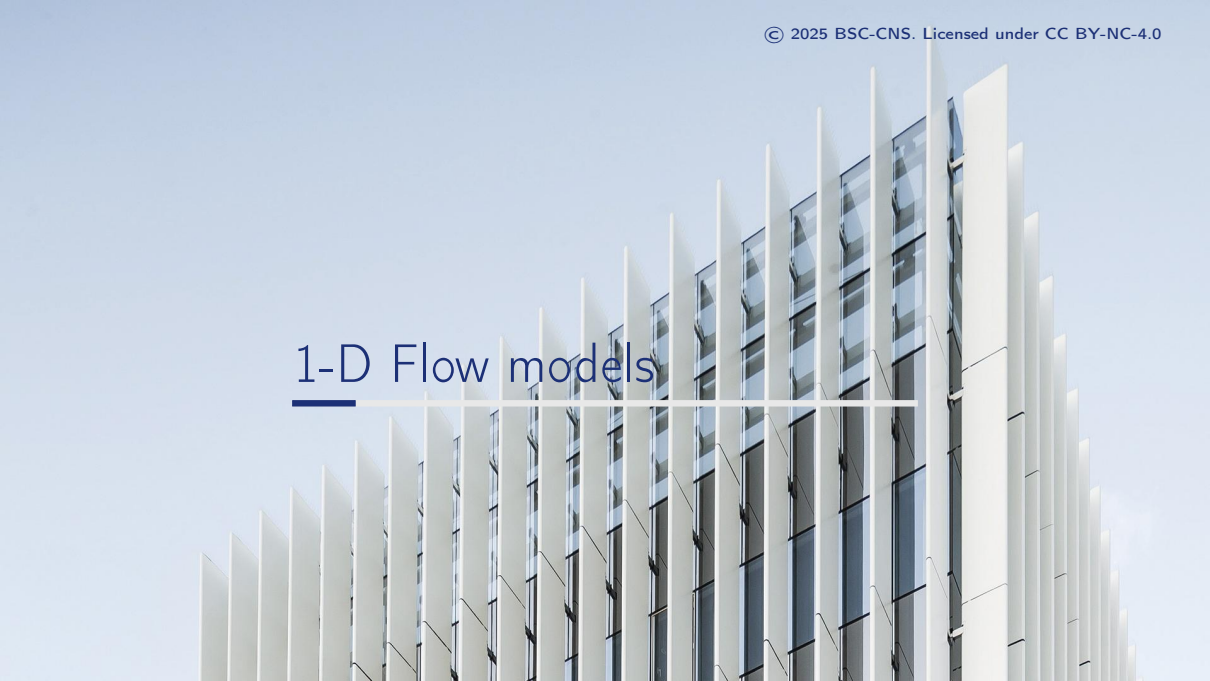
Javier Béjar - UPC

BSC AI Factory Partners

Affiliated entities

Flow models

- Autoregressive models are limited to discrete variables
- We want to be able to model probability density functions for continuous variables
- We want also the usual:
  - A good fit to the training data (generalization)
  - Be able to compute $p_\theta(x)$ for new examples x (efficiently better)
  - Able to sample from $p_\theta(x)$



1-D Flow models

Flows

- Flows are a general method for fitting density probability models
- They are based on transforming a probability density into another through a set of transformation functions
- These sets of transformations maintain the properties of being a density probability function
- The original density is adapted to the target density distribution

Flows - General Idea

- A flow is composed by a chain of transformations $f_\theta = f_1, \ldots, f_n$
- Each transformation preserves the density properties
- The result will be a value z that is a transformation of x through the chain of functions $z = f_\theta(x)$
- z will belong to a probability distribution $p_Z(z)$ of our choice
- To require a specific target distribution for z will force the function f_θ to be a distribution
- When z belongs to a Normal distribution $\mathcal{N}(0, 1)$ this is called a Normalizing Flow

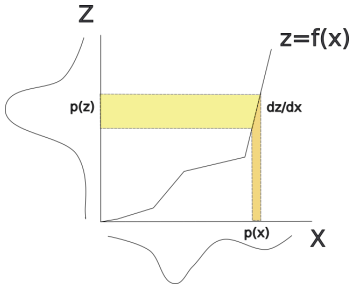
Flow functions

- The requirements for being a flow function are that it has to be monotone, invertible and differentiable
- The goal is:
  - to map uniquely any element from one density to another (monotone)
  - to obtain samples of $p(x)$ from samples of $p(z)$, so $x = f_\theta^{-1}(z)$ (invertible)
  - to be able to train the parameters of the function using gradient descent (differentiable) using maximum likelihood

$$\max_\theta \sum_i \log p_\theta(x_i)$$

- We do not have access directly to $p_\theta(x)$, so we make a transformation performing a change of variable $z = f_\theta(x)$

$$
\begin{aligned}
p_\theta(x)dx &= p_Z(z)dz = p_Z(f_\theta(x))dx \\
p_\theta(x) &= p_Z(f_\theta(x)) \left| \frac{\partial f_\theta(x)}{\partial x} \right|
\end{aligned}
$$



- Given that we have an expression for p_Z

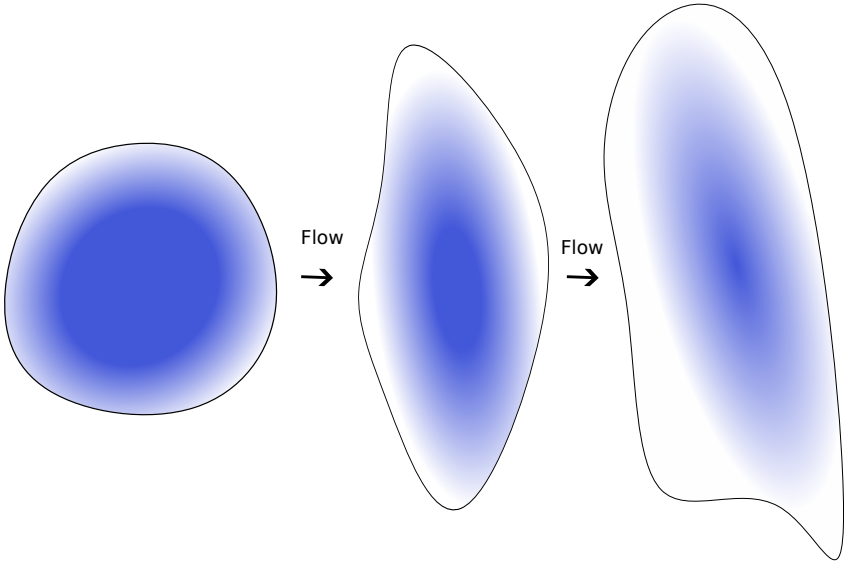
$$
\max_\theta \sum_i \log p_\theta(x_i) = \max_\theta \sum_i \log p_z(f_\theta(x_i)) + \log \left| \frac{\partial f_\theta(x_i)}{\partial x} \right|
$$

Flows using neural networks

- We have some constraints that must be followed:
  - Activation function must be differentiable and invertible (e.g. sigmoid, tanh, Leaky ReLUs, but nor ReLUs)
  - The input and output must have the same dimensionality
- The composability of flows allows having different processing layers, so we can use deep models

N-D Flow models

Changing N variables



- Autoregressive flows implement a transformation of the form

$$z_i = f_\theta(x_i|x_{<i}) \quad x_i = f_\theta^{-1}(z_i|x_{<i})$$

- Fitting is the same as before but each variable is going to add a term to the likelihood

$$p_x(x) = \prod_{i=1}^{D} p_x(x_i|x_{<i})$$

So

$$\max_\theta \sum_i \sum_j \log p_z(f_\theta(x_{ij}|x_{<j})) + \log \left| \frac{\partial f_\theta(x_{ij}|x_{<j})}{\partial x_j} \right|$$

- Training can be done by gradient descent, given that we have all the x, and we can fit the different functions in parallel
- Sampling is going to be slow given that we have to compute each $z_{<i}$ for computing z_i
- This could be impractical for domains with high dimensionality
- Given that flows are invertible functions we can reformulate the problem as

$$z_i = f_\theta^{-1}(x_i|z_{<i}) \quad x_i = f_\theta(z_i|z_{<i})$$

- This makes the model slower to train, but with a fast sampling

Changing many variables

- Autoregressive flows only change one variable at a time, we could work with all variables
- The transformation f from x to z is rescaling the probability mass from one volume to another, we can write the relation among the two distributions as:

$$p(x)vol(dx) = p(z)vol(dz)$$

  In other terms, the transformation must distribute the probability mass from x to fill the probability mass of z
- We can rewrite $p(x)$ as:

$$p(x) = p(z)\frac{vol(dz)}{vol(dx)} = p(z)\left|det\frac{dz}{dx}\right|$$

- This reformulation allows to write the change of variable formula as:

$$p_\theta(x) = p(f_\theta(x)) \left| det \frac{\partial f_\theta(x)}{dx} \right|$$

- That can be optimized using maximum likelihood as:

$$\min_\theta \mathbb{E}_x[-\log(p_\theta(x)] = \mathbb{E}_x \left[-\log p_z(f_\theta(x)) - \log \left| det \frac{\partial f_\theta(x)}{dx} \right| \right]$$

- This is only scalable if the determinant of the jacobian is easy to compute (basically the jacobian needs to be a triangular matrix, so the determinant is the product of the diagonal elements) reducing expressiveness
- The composability property of Flows can reduce the problem

- Affine transformations can be used to compute flows
- The parameters are an invertible matrix A and a vector b (basically a linear one layer perceptron)

$$f(x) = A^{-1}(x - b)$$

- Sampling can be performed using a gaussian distribution $\mathcal{N}(0, 1)$

$$x = Az + b$$

- Computing the log likelihood is expensive for high dimensionality, it involves computing $det(A)$, that is not usually a triangular matrix

Elementwise flows

- We could build a flow using each variable independently with different choices for the transformation function

$$f_\theta(x_1, \ldots, x_n) = (f_\theta(x_1), \ldots, f_\theta(x_n))$$

- The jacobian is diagonal, so the determinant is easy to compute

$$\begin{aligned} \frac{\partial z}{\partial x} &= diag(f'_\theta(x_1), \ldots, f'_\theta(x_n)) \\ det \frac{\partial z}{\partial x} &= \prod_{i=1}^{d} f'_\theta(x_i) \end{aligned}$$

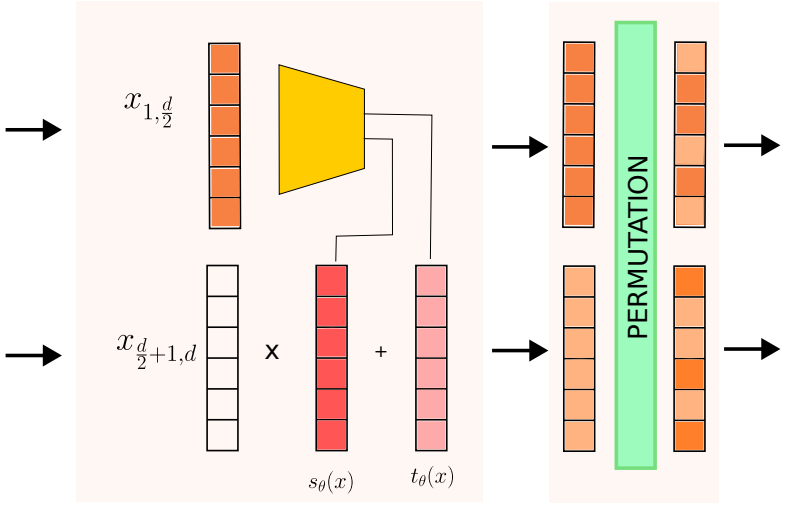
- We gain efficiency, but we lose expressive power

- We can combine these two ideas to obtain expressiveness and efficiency
- Split the variables in two groups $x_1, \ldots, x_{\frac{d}{2}}, x_{\frac{d}{2}+1}, \ldots, x_d$
- Separate the transformation in two

$$\begin{aligned} z_{1:\frac{d}{2}} &= x_{1:\frac{d}{2}} \\ z_{\frac{d}{2}+1:d} &= x_{\frac{d}{2}+1:d} \cdot s_\theta(x_{1:\frac{d}{2}}) + t_\theta(x_{1:\frac{d}{2}}) \end{aligned}$$

- This is an invertible transformation
- s_θ (scaling) and t_θ (translation) can be arbitrary neural networks
- We can use multiple successive layers and perform a reordering of variables between each transformation (also invertible) to mix the variables and obtain better expressiveness

NICE/RealNVP

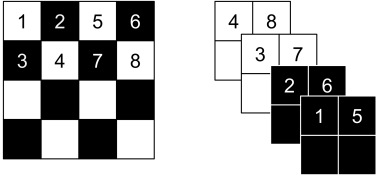


RealNVP

- RealNVP: flow based architecture for generating images
- Uses a checkerboard pattern to partition the variables (the pattern matters)
- Implements a multiscale architecture squeezing the input and changing spatial resolution for number of channels

$$(d, d, c) \rightarrow (\frac{d}{2}, \frac{d}{2}, 4 \times c)$$

- Each layer alternates spatial and channel wise checkerboard patterns



This Python Notebook shows examples of Fow models

- Normalizing Flows Notebook (click here to open the notebook in colab)

If you download the notebook you will be able to use it locally (run jupyter notebook to open the notebooks)

BSC AI Factory

AI Factories

Co-funded by the European Union

Plan de Recuperación, Transformación y Resiliencia

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101234399 and Spain, Portugal, Romania and Türkiye. The JU receives support from the European Union's Horizon Europe Programme.

BSC AI Factory Partners

Affiliated entities