Deep Probabilistic Models

Part I: Flow-Based Models

Robert Salomone

AMSI Winter School 2021

Introduction

- **Neural Networks** are a great tool for approximating a **general** function.
- The course so far has focused on **supervised** learning (i.e., **regression/classification**): given regressors $\boldsymbol{x}$, predict the response variable y (or maybe a vector $\boldsymbol{y}$).

This Week...

- This week is about **unsupervised** learning. We will explore models and methods for three key areas:
  - **Density Estimation**: Learning the underlying distribution of $\boldsymbol{X}$.
    - **Conditional Density Estimation**: learning the underlying distribution $\boldsymbol{X}|\boldsymbol{Y} = \boldsymbol{y}$.
  - **Generative Models**: Models that allow you to easily **simulate** from them, regardless of whether or not you have learned the explicit probability density function.
  - **Representation Learning**: learning some "nice" way of representing the data (by representing it with independent components and/or in a low-dimensional latent space).
- All the approaches we will cover will achieve one or more of the above, and we will see examples of how all of the above interact.
- **Neural nets will appear as black-box functions to achieve the above, they are the tool of choice as they are very flexible and trainable via gradient descent.**

Deep Probabilistic Models

Deep

- Involving **many layers** in some way.
  - Functions are Deep Neural Networks
  - Many Layers of **Latent Variables**
  - Both!
- The reason: **Flexibility** of our models.

Probabilistic

- It will involve **probability distributions**.

Why bother?

- Let me give you **two examples** of what can be pulled off with what we will look at this week...
- I stress however, that applications are **very broad** (e.g., some of the ideas you see have been used for example within new MCMC algorithms, in scientific modelling, and for generating synthetic data).
- However, without further ado, let me **show you why this week's content is interesting**...

This person and this cat do not exist...

- These are **simulations** from a fit model of the joint distribution of pictures of people, and pictures of cats, respectively. Talk about a **flexible** model!

So you think you can Bayes?

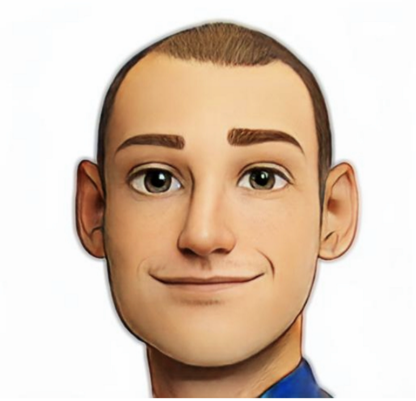
- That was kind of cool.
- But what is really neat is with models involving probability distributions you can "**be Bayesian**".
- By now, **this** is probably what you think "being Bayesian" is

$$p(\boldsymbol{\theta}|\boldsymbol{y}) \propto p(\boldsymbol{\theta})p(\boldsymbol{y}|\boldsymbol{\theta}).$$

- It is! However, let me show you how else we can "**be Bayesian**"...

The Power of Bayes Rule

$\mathbf{X} \sim p_{\mathbf{X}, y=\text{photo}}$



$\mathbf{X}|\mathbf{Z} \sim p_{\mathbf{X}|\mathbf{Z}, y=\text{cartoon}}$

$(Z_1, Z_2, \ldots, Z_m) \sim p_{\mathbf{Z}|\mathbf{X}, y=\text{photo}}$

(posterior over low-dimensional latent random vector)

- One of the goals of the course is for you to understand the above figure, and how a model called a **conditional variational autoencoder** can accomplish the above with (amortized) **approximate posterior inference**.

Roadmap

1. **Part I**: **Flow-Based Models** and their implementation in **Pyro**
1. **Part II**: **Generative Adversarial Networks** (GAN) and **Stochastic Backpropagation**
2. **Part III**: **Probabilistic Graphical Models**, **Variational Inference**, and **Pyro Basics**
    - The Variational Inference we will discuss is actually a **generalization** of what you may have seen with the same name so far at this Winter School, and we will use it to train (deep) latent variable models.
3. **Part IV**: **Amortized Inference** and **Variational Autoencoders**

Pedagogical Approach

- We will **not** focus on generating faces, cats, or conditional cartoons **(!)**
- With the exception of maybe one example, all data will be vector-valued. This **simplifies the implementations considerably** as well as how easy it is to grasp.
- Typically, ideas we will see are presented separately, but in fact they share many conceptual and mathematical aspects. I will **try to put it all together** and show you **how they relate**.
- Also get a taste of `Pyro`, which is a probabilistic programming language built on top of `Pytorch` that is designed for models **and** inference algorithms suitable to the Bayes rule that gave you a cartoon Rob.
- **My Goal**: Show you some **neat** ideas, presented in a **cohesive framework**, that a lot of people in the statistics community do not know about, and **prepare you to learn more** if you so desire.
- Finally, there will also be links throughout in pink, so anyone who is keen can refer back to these slides and go straight to papers referenced or website with more information.

A Note on Topics Covered

- We focus on things that are now the dominant approach owing to their **performance**, **flexibility**, and **scalability**.
    - **State of the art**!
    - I should mention that we will be excluding things such as **energy-based models** (e.g., Boltzmann Machines) and **undirected graphical models**, which are bound to come back in fashion sooner or later!

Part I: Flow-Based Models

Introduction: Flow-Based Models

- **Flow-Based models** are models that arise from transformations of some simple **base** distribution, usually a multivariate standard normal:

$$\boldsymbol{Z} \sim \mathcal{N}(\mathbf{0}, \mathrm{I})$$

- Sometimes the distribution for $\boldsymbol{Z}$ is called the **prior**, but we will avoid this abuse of terminology!
- Generally, for some function T from p-dimensional to a q-dimensional space, we have that

$$\boldsymbol{X} = T(\boldsymbol{Z})$$

  is **some** random object.
- In **principle**, if we had a very flexible class of functions T parametrized by some $\boldsymbol{\theta}$, **and** we could compute its likelihood function $p_{\boldsymbol{\theta}}(\boldsymbol{X})$, we could fit our samples $\boldsymbol{X}$ by **maximum likelihood**.
- Neural networks are general functions! So, ideally we would just plug $\boldsymbol{Z}$ into a neural net.
- We just need to find its associated probability distribution, so we need to have a look at how...

What is the probability distribution of $T(\boldsymbol{Z})$?

- If the random vector $\boldsymbol{Z}$ has probability measure μ, then $\boldsymbol{X} = T(\boldsymbol{Z})$ has measure $\nu = T_\sharp \mu$ defined to be the measure that satisfies

$$\nu(B) = \mu(T^{-1}(B))$$

  for all sets B in the space where $\boldsymbol{Z}$ takes values in.

- This is called the **pushforward** measure of μ under the map T.
- **Ahhhh! That's horrible!**
- The above is, in general not computable, and very abstract.
- To avoid the above headaches, we will restrict ourselves to T being a bijective (one-to-one and onto) map from $\mathbb{R}^d$ to $\mathbb{R}^d$ that is **invertible**, **differentiable**, and has **differentiable inverse**.
  - Then, we will **never need to talk about measures again**. :D
  - However, we need to **forget about neural networks for a bit**, we will figure out how to use them to make a tractable T later.

Change of Variables Theorem

- Suppose that $\boldsymbol{Z}$ has pdf $p(\cdot)$ and $T : \mathbb{R}^d \to \mathbb{R}^d$ is a **diffeomorphism**. Then, the random vector $\boldsymbol{X} = T(\boldsymbol{Z})$ has the pdf

$$p(\boldsymbol{x}) = p\big(T^{-1}(\boldsymbol{x})\big)\left|\det \mathrm{J}_{T^{-1}}(\boldsymbol{x})\right|$$

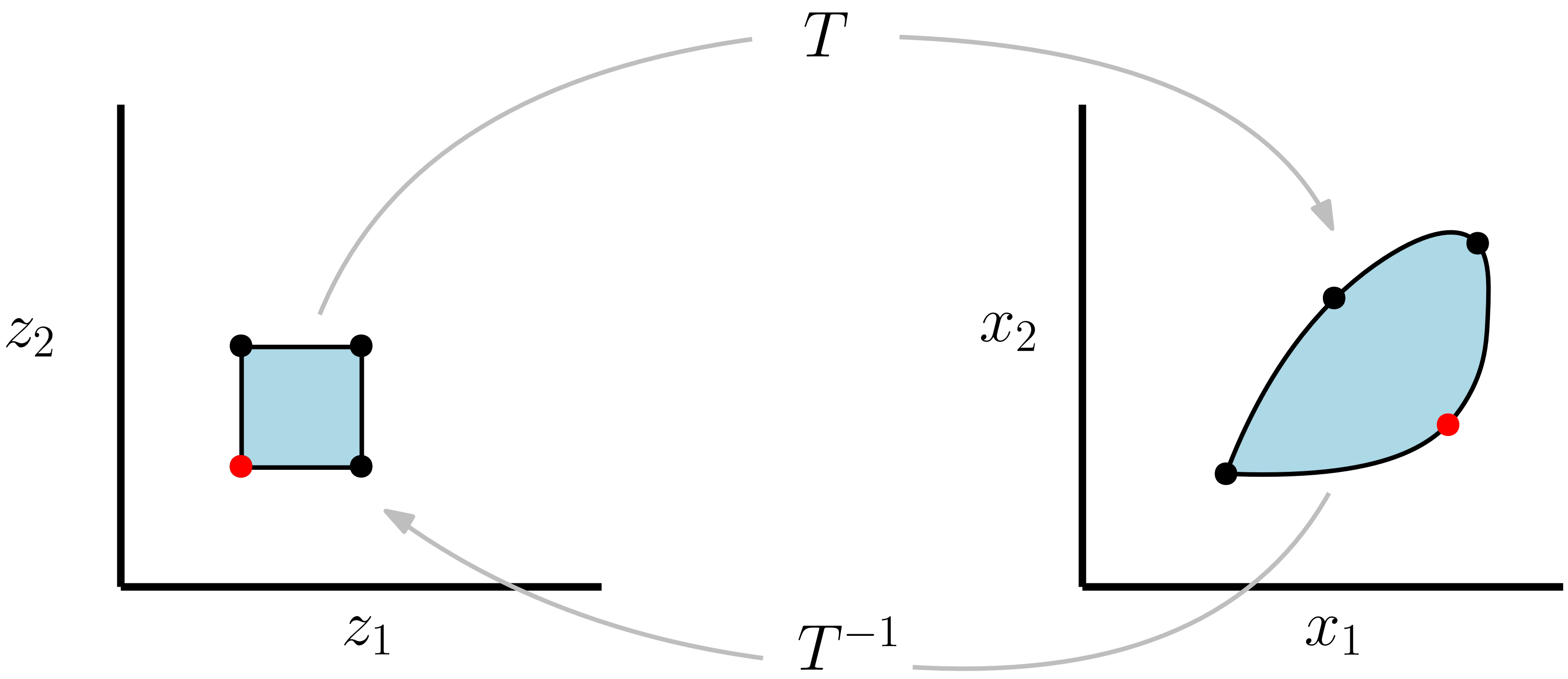
  - Above, $\mathrm{J}_{T^{-1}}(\boldsymbol{x})$ denotes the **Jacobian Matrix** of the inverse map T^{-1}, evaluated at the point $\boldsymbol{x}$.

$$\mathrm{J}_{T^{-1}}(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial z_1(\boldsymbol{x})}{\partial x_1} & \cdots & \frac{\partial z_1(\boldsymbol{x})}{\partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_d(\boldsymbol{x})}{\partial x_1} & \cdots & \frac{\partial z_d(\boldsymbol{x})}{\partial x_d} \end{bmatrix}$$

- As a consequence of the inverse function theorem, we also have that $\mathrm{J}_{T^{-1}}(\boldsymbol{x}) = \mathrm{J}_T(T^{-1}(\boldsymbol{x}))^{-1}$.
  - So, if we desire we can also write $|\mathrm{det}\mathrm{J}_{T^{-1}}(\boldsymbol{x})| = |\mathrm{det}\mathrm{J}_T(T^{-1}(\boldsymbol{x}))|^{-1}$.
- You may be wondering "what is that **Jacobian determinant** thing doing"? Let's look...

Jacobian Determinant

- The determinant of the Jacobian matrix accounts for the **infinitesimal change** in volume at each point. It is a **scaling** factor.



Basic 1D Example: *sinh-arcsinh* distribution

- Let $Z \sim \mathcal{N}(0,1)$, then take

$$X = T(Z\,;\,\epsilon,\delta) = \sinh(\delta^{-1}(\operatorname{arcsinh}(Z)+\epsilon)), \quad \epsilon \in \mathbb{R}, \delta \in \mathbb{R}_+.$$

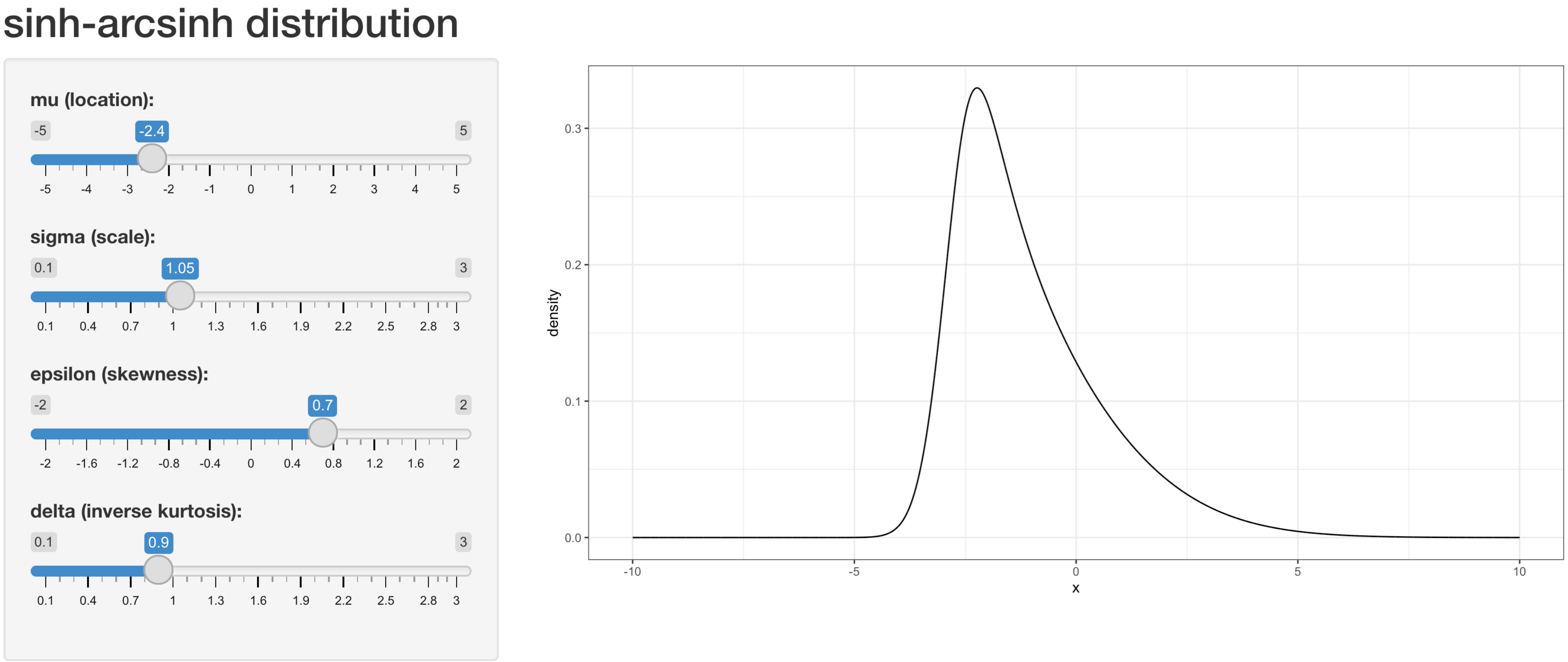
- By the Change of Variables Theorem, the above has pdf

$$p_X(x) = \phi\big(T^{-1}(x)\big) \cdot \left|\frac{\partial x}{\partial z}\right| = p_Z\big(\sinh(\delta \sinh^{-1}(x) - \epsilon)\big) \cdot \left|\frac{\delta \cosh(\delta \sinh^{-1}(x) - \epsilon)}{\sqrt{1+x^2}}\right|$$

- This is actually how you construct the **sinh-arcsinh**(δ,ϵ) distribution.
  - Jones, M. C., & Pewsey, A. (2009). Sinh-arcsinh distributions. *Biometrika*, 96(4), 761-780.

Location Scale Sinh-Arcsinh

- I made an **RShiny** app to play around with visualizing the density for the location-scale **sinh-arcsinh** family: Click This Link!



Example in d-dimensions: Multivariate Normal

- Let L be a lower triangular matrix, and $\boldsymbol{Z} \sim \mathcal{N}(\mathbf{0}, \mathrm{I})$. Perform the transformation

$$\boldsymbol{X} = T_{\mathrm{L},\boldsymbol{\mu}}(\boldsymbol{Z}) = \boldsymbol{\mu} + \mathrm{L}\boldsymbol{Z}.$$

- Then, it is easy to show that $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ where $\Sigma = \mathrm{L}\mathrm{L}^{\top}$.

Composing Transformations for Additional Flexibility

- More generally, we can compose several transformations together into a "flow" of transformations to increase the flexibility of our final distribution.

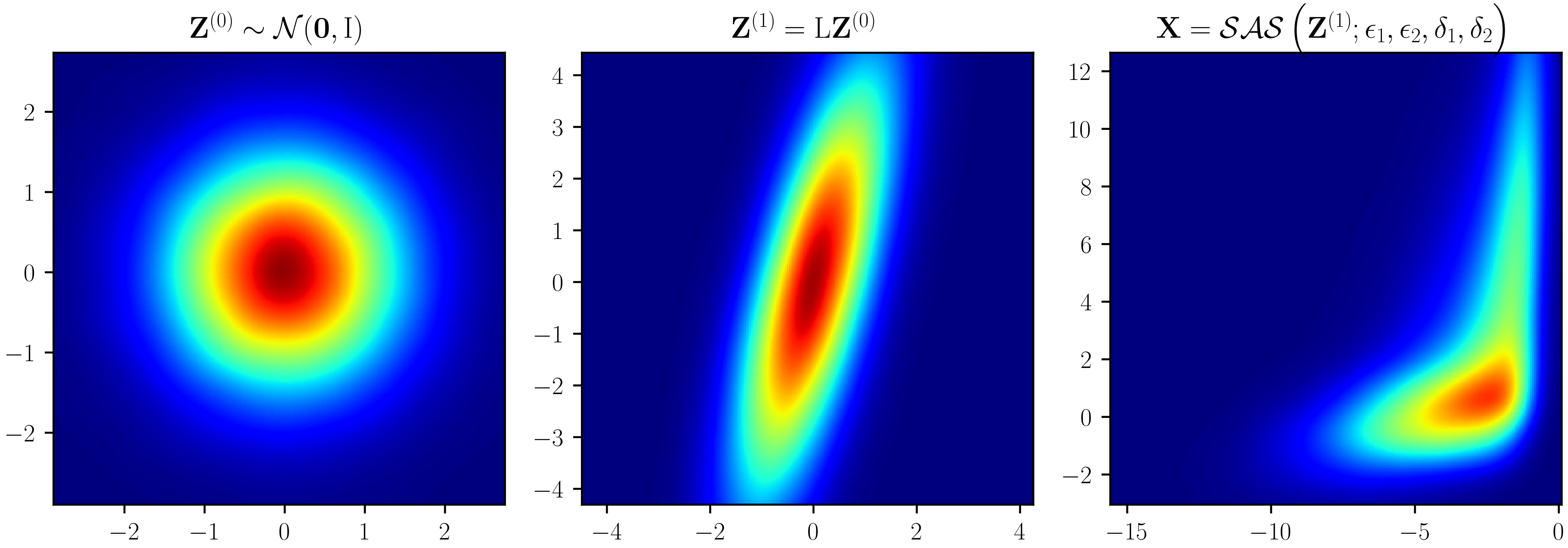
- Let's do a simple example, letting $\mathcal{SAS}$ denote the sinh-arcsinh transform defined earlier:

$$\boldsymbol{Z}^{(0)} \sim \mathcal{N}(\mathbf{0}, \mathrm{I})$$

$$\boldsymbol{Z}^{(1)} = T_1(\boldsymbol{Z}^{(0)}) = \mathrm{L}\boldsymbol{Z}^{(0)}$$

$$\boldsymbol{X} = T_2(\boldsymbol{Z}^{(1)}) = \mathcal{SAS}(\boldsymbol{Z}^{(2)};\, \boldsymbol{\epsilon}, \boldsymbol{\delta})$$

A "Flow" of Two Transformations



Identities for the Two Transform Case

- Note that **inverse of a composition** is simply the **individual inverses iterated backwards**

$$T^{-1}(\boldsymbol{x}) = (T_2 \circ T_1)^{-1}(\boldsymbol{x}) = T_1^{-1} \circ T_2^{-1}(\boldsymbol{x})$$

- Similarly, by the (multivariate) chain rule, and using that the determinant of a product of matrices is the product of determinants, we can obtain

$$\det \mathrm{J}_{T_1^{-1} \circ T_2^{-1}}(\boldsymbol{x}) = \det \mathrm{J}_{T_1^{-1}}(T_2^{-1}(\boldsymbol{x})) \cdot \det \mathrm{J}_{T_2^{-1}}(\boldsymbol{x})$$

- If we have $m > 2$ individual transformations, we can apply the procedure **recursively**...

A Tale of Two Directions...

Simulation (Forward)

Input: transforms $\{T_k\}_{k=1}^K$ and base distribution $p_{\boldsymbol{z}}$

Draw $\boldsymbol{Z}_0 \sim p_{\boldsymbol{z}}$

for $k = 1, \ldots, K$:

$$\boldsymbol{Z}_k \leftarrow T_k(\boldsymbol{Z}_{k-1})$$

Return $\boldsymbol{X} = \boldsymbol{Z}_K$ as a sample from the normalizing flow.

Log-Likelihood Evaluation (Backward)

Input: Sample $\boldsymbol{x}$, inverse-transforms $\{T_k^{-1}\}_{k=1}^K$, and base density $p_{\boldsymbol{z}}(\cdot)$

1. Initialise: $\boldsymbol{z}_K \leftarrow \boldsymbol{x}$, $\quad 0 \leftarrow \text{logdet\_term}$
2. for $l = K, \ldots, 1$:

   $\text{logdet\_term} \leftarrow \text{logdet\_term} + \log \left|\det \mathrm{J}_{T_l^{-1}}(\boldsymbol{z}_l)\right|$

   $\boldsymbol{z}_{l-1} \leftarrow T^{-1}(\boldsymbol{z}_l)$

1. Return $p_{\boldsymbol{z}}(\boldsymbol{z}_0) + \text{logdet\_term}$

Time to get serious...

- ***Question***: Can we use neural networks to make a very flexible multivariate T somehow?
  - **Challenge #1**: We need to make sure T is *invertible* and *surjective*.
  - **Challenge #2**: In general, determinant computation is $\mathcal{O}(d^3)$ complexity. We need something that will scale to very high dimensions.
- In light of the above, it isn't as easy as just throwing $\boldsymbol{Z}$ into **any** neural net (we will learn how to get away with that next lecture, but we can't do maximum likelihood then).
- **Answer**: Yes we can, as long as we are **smart** about the way we use neural nets.
  - The neural nets aren't going to be the transform itself, but **determine the parameters of the transforms**.

Real Non-Volume Preserving (Real NVP) Transformations (Dinh et al., 2017)

Real Non-Volume Preserving (Real NVP) Flows

- Partition $\boldsymbol{Z} = \begin{pmatrix} \boldsymbol{Z}_A \\ \boldsymbol{Z}_B \end{pmatrix}$, where $\boldsymbol{Z} \in \mathbb{R}^d$, $\boldsymbol{Z}_A \in \mathbb{R}^{n_A}$ and $\boldsymbol{Z}_B \in \mathbb{R}^{n_B}$. A typical approach is to split it down the middle, i.e., take $n_A = \lfloor d/2 \rfloor$ and $n_b = d - n_A$
- Let $\boldsymbol{\mu} : \mathbb{R}^{n_A} \to \mathbb{R}^{n_B}$ and $\boldsymbol{s} : \mathbb{R}^{n_A} \to \mathbb{R}^{n_B}$ be arbitrary but flexible (neural net!) functions parametrized by some vector ξ that we suppress in the notation. Denote the function $\boldsymbol{\sigma} = \exp \circ \boldsymbol{s}$ elementwise.
- Write $\odot$ to denote **elementwise multiplication**, and $\oslash$ to denote **elementwise division**.
- Then, define the (single transform) **Real NVP transformed random vector** as

$$\boldsymbol{X} = T(\boldsymbol{Z}) = \begin{pmatrix} \boldsymbol{Z}_A \\ \boldsymbol{\mu}(\boldsymbol{Z}_A) + \boldsymbol{\sigma}(\boldsymbol{Z}_A) \odot \boldsymbol{Z}_B \end{pmatrix}$$

- It is straightforward to see that **the inverse is**

$$\boldsymbol{Z} = T^{-1}(\boldsymbol{X}) = \begin{pmatrix} \boldsymbol{X}_A \\ (\boldsymbol{X}_B - \boldsymbol{\mu}(\boldsymbol{X}_A)) \oslash \sigma(\boldsymbol{X}_A) \end{pmatrix}.$$

- This operation is also known as an **affine coupling transform**.

Real NVP Jacobian

- It is straightforward to derive that the Jacobian is equal to

$$\mathrm{J}_T = \begin{bmatrix} \mathrm{I} & 0 \\ \frac{\partial \boldsymbol{x}_B}{\partial \boldsymbol{z}_A} & \mathrm{diag}\big(\exp(\boldsymbol{s}(\boldsymbol{z}_A))\big) \end{bmatrix}$$

- And, because the **determinant of lower triangular matrices is simply the product of the diagonal**:

$$\det \mathrm{J}_T(\boldsymbol{z}) = \prod_{k=1}^{n_A} 1 \prod_{k=1}^{n_B} \sigma_k(\boldsymbol{z}_A) = \prod_{k=1}^{n_B} \sigma_k(\boldsymbol{z}_A) \implies \det \mathrm{J}_{T^{-1}}(\boldsymbol{x}) = \left(\prod_{k=1}^{n_B} \sigma_k(\boldsymbol{x}_A)\right)^{-1}$$

- So, using that the individual σ_k are all positive, we obtain the **beautifully simple result** that

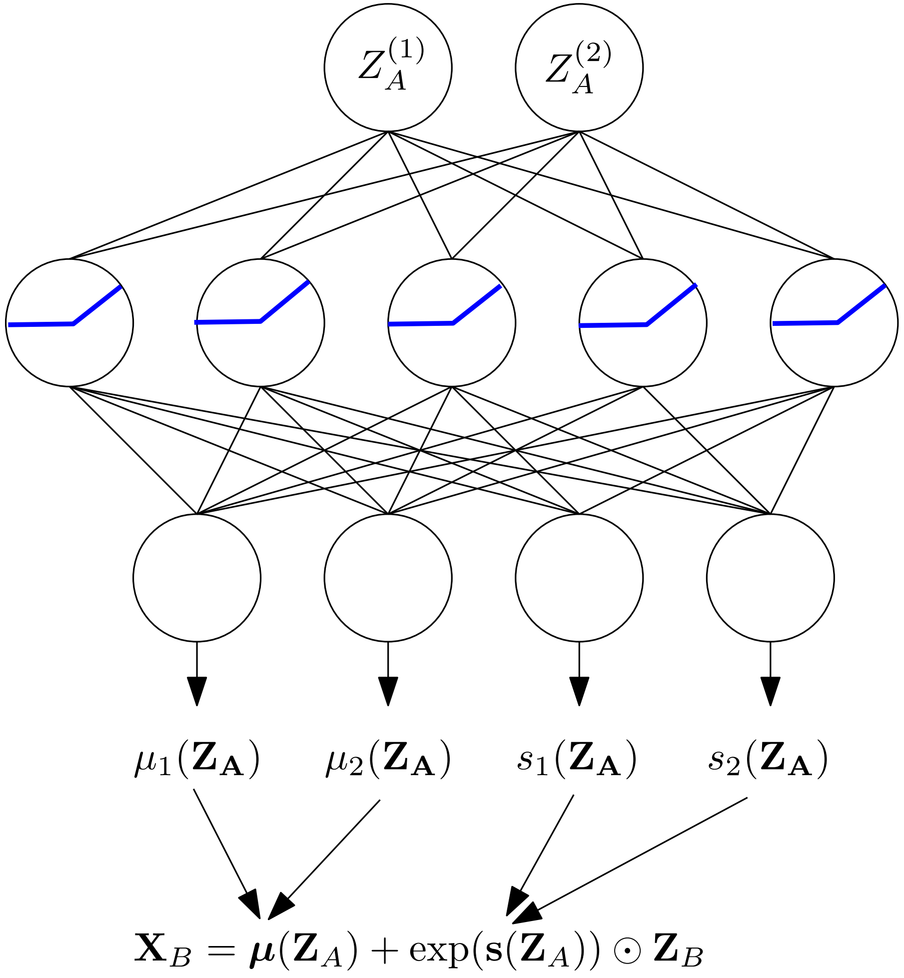
$$\log \left|\det \mathrm{J}_{T^{-1}}(\boldsymbol{x})\right| = -\sum_{k=1}^{n_B} s_k(\boldsymbol{x}_A).$$

- **It's easy, and computed in linear time!** We have $\mathcal{O}(d)$ computation of determinant and the functions μ and σ can be as complicated as we like!

- There is one problem however, only half of the variables get transformed! (we deal with that soon).

Real NVP Transform

- In practice, all parameters are decided by a **single neural network** that takes in $\boldsymbol{Z}_A$ and outputs $\boldsymbol{\mu}$ and $\boldsymbol{s}$.
- This is illustrated in the figure on the right.



Permutation Transforms

- Recall that a Real NVP transform does not transform about half of the variables.
- Introducing a transform that reverses the order of the variables:

$$\mathrm{P}_{\mathrm{reverse}} : (x_1, x_2, \dots, x_{p-1}, x_p) \mapsto (x_p, x_{p-1}, \dots, x_2, x_1)$$

- Reordering the variables according to a permutation is equivalent to multiplication by a **permutation matrix**.
- A **permutation matrix** is a matrix that has exactly one entry equal to one in each row and column, and zero elements elsewhere.
- Permutation operations are volume-preserving: $\det \mathrm{P} = 1$.
- **Intersperse reverse permutations with Real NVP and we have solved the issue!**

Real NVP Flow with K Real NVP layers...

- For $\boldsymbol{Z} \sim \mathcal{N}(\mathbf{0}, \mathrm{I}_d)$, we have

$$\boldsymbol{X} = T(\boldsymbol{Z}) := T_K \circ \mathrm{P}_{\mathrm{reverse}} \circ \cdots \circ T_2 \circ \mathrm{P}_{\mathrm{reverse}} \circ T_1(\boldsymbol{Z})$$

  with T is parametrized by some $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_K\}$ where each $\boldsymbol{\theta}_k$ is the parameters of the k-th transform.

- As as a fit T^{-1} to some data will transform $p_{X_{\mathrm{data}}}$ approximately to a normal distribution, the flow T^{-1} is often referred to as a **normalizing flow**.

More flexible (volume-preserving) ways that involve variable interaction...

- One can in principle use some other transformation that makes the two layers "interact".
- One example is to use a **Householder Reflection** (or composition thereof)

$$\boldsymbol{Z}^{(t+1)} = \left(\mathrm{I} - 2\frac{\boldsymbol{v}_t \boldsymbol{v}_t^\top}{||\boldsymbol{v}_t||^2}\right) \boldsymbol{Z}^{(t)}$$

which requires only $\mathcal{O}(d)$ parameters.

- Applying a sparse matrix with Given's Rotations is another possible way (see e.g., Section 4 of this paper for an operation only requiring $\mathcal{O}(\log d)$ parameters to make all variables interact).
- Both approaches above are **volume-preserving** so we don't need to worry about Jacobian terms.

Masked Autoregressive Flows

Masked Autoregressive Flows

- Here, we consider

$$X_1 = \mu_1 + \sigma_1 Z_1$$

$$X_2 = \mu_2(X_1) + \sigma_2(X_1) Z_2$$

$$X_3 = \mu_3(X_1, X_2) + \sigma_3(X_1, X_2) \cdot Z_3$$

$$\vdots$$

$$X_d' = \mu_d(\boldsymbol{X}_{1:d-1}) + \sigma_d(\boldsymbol{X}_{1:d-1}) \cdot Z_d$$

- Issue: Generation is Slow (Sequential)
- However, inversion is **very fast**!

$$Z_k = \frac{X_k - \mu_k(\boldsymbol{X}_{1:k-1})}{\sigma_k(\boldsymbol{X}_{k-1})}, \quad k = 1, \ldots, d$$

MAF

Generation (*Sequential*)

$$X_1 = \mu_1 + \sigma_1 Z_1$$

$$X_2 = \mu_1(X_1) + \sigma_1(X_1) Z_2$$

$$X_3 = \mu_2(X_1, X_2) + \sigma_2(X_1, X_2) \cdot Z_3$$

$$\vdots$$

$$X_d = \mu_{d-1}(\boldsymbol{X}_{1:d-1}) + \sigma_{d-1}(\boldsymbol{X}_{1:d-1}) \cdot Z_d.$$

Inversion (*Parallel*)

$$\boldsymbol{Z} = \begin{pmatrix} (X_1 - \mu_1)/\sigma_1 \\ \big(X_2 - \mu_1(X_1)\big)/\sigma_1(X_1) \\ \vdots \\ \big(X_d - \mu_{d-1}(\boldsymbol{X}_{1:d-1})\big)/\sigma_{d-1}(\boldsymbol{X}_{1:d-1}) \end{pmatrix}$$

- Recall, we require inversion to evaluate the (log)-likelihood function (and subsequently get its gradient):

$$\log p_{\boldsymbol{X}}(\boldsymbol{x}) = \log p_{\boldsymbol{Z}}\big(T^{-1}(\boldsymbol{x})\big) + \log \left|\det \mathrm{J}_{T^{-1}}(\boldsymbol{x})\right|$$

MAF Jacobian

- MAFs have a **lower triangular** Jacobian, so again the determinant is just the product of the Jacobian's **diagonal** entries (similar to the case with the multivariate normal from earlier).
- Determinant computation is $\mathcal{O}(n)$ as with Real NVP.
- As before, we reverse the order with a permutation map between each MAF in a flow.

Autoregressive Neural Networks

- Similar to how we did it all one one neural network for Real NVP, it is possible to get *all* the parameters out of a neural network, but it requires a very particular structure that *masks* weights (i.e., sets some weights to be zero) so the **autoregressive condition is satisfied**.

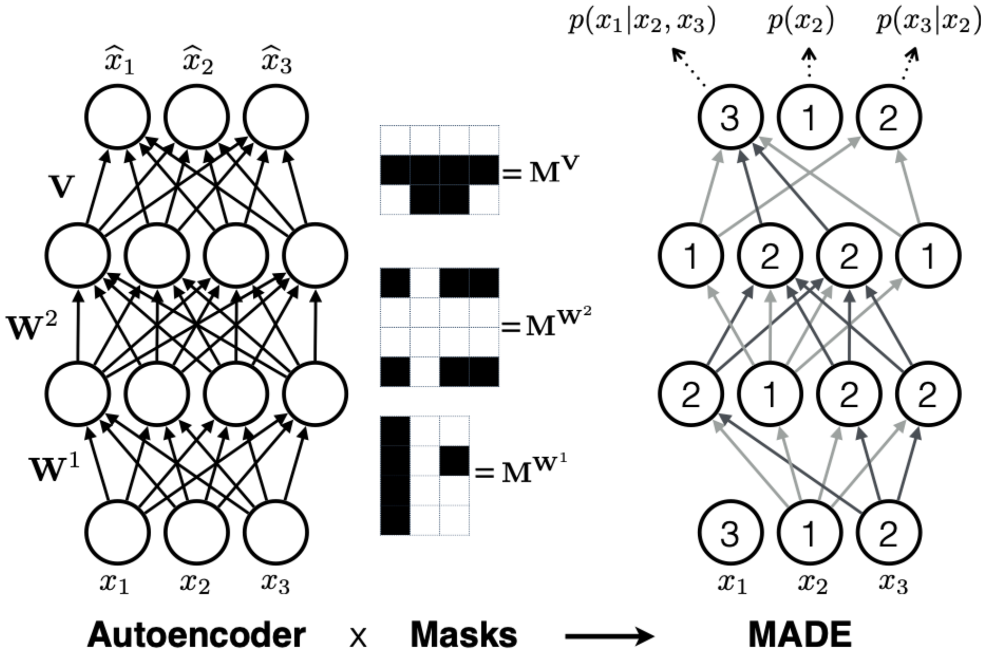


Figure from Germain et al., (2015), MADE: Masked Autoencoder for Distribution Estimation, ICML 2015. (Precursor to MAF)

Inverse Autoregressive Flows

Inverse Autoregressive Flows

- Replacing the X terms with Z terms within the μ and σ functions in Masked Autoregressive flows is equivalent to instead using its inverse as a flow (up to reparametrization).
- In this case, it is parallelizable in the **generation** direction, the Jacobian determinant term is again computable in $\mathcal{O}(n)$ time.

Generation (*Parallel*)

$$\boldsymbol{X} = \begin{pmatrix} \mu_1 + \sigma_1 Z_1 \\ \mu_2(Z_1) + \sigma_2(Z_1) \cdot Z_2 \\ \mu_3(Z_1, Z_2) + \sigma_3(Z_1, Z_2) \cdot Z_3 \\ \vdots \\ \mu_d(\boldsymbol{Z}_{1:d-1}) + \sigma_d(\boldsymbol{Z}_{1:d-1}) \cdot Z_d \end{pmatrix}$$

Inversion (*Sequential*)

$$Z_1 = (X_1 - \mu_1)/\sigma_1$$

$$Z_2 = \big(X_2 - \mu_2(Z_1)\big)/\sigma_2(Z_1)$$

$$Z_3 = \big(X_3 - \mu_3(Z_1, Z_2)\big)/\sigma_3(Z_1, Z_2)$$

$$\vdots$$

$$Z_d = \big(X_d - \mu_d(\boldsymbol{Z}_{1:d})\big)/\sigma_d(\boldsymbol{Z}_{1:d-1})$$

Affine Inverse Autoregressive Flow

- Consider this familiar friend from earlier:

$$\boldsymbol{X} = \nu + \mathrm{L}\boldsymbol{Z}$$

- The above is just a *linear* autoregressive flow. Actually, we just take

$$\mu_j(\boldsymbol{Z}_{1:j-1}) = \nu_j + \sum_{k=1}^{j-1} L_{ik} Z_k, \quad \text{and} \quad \sigma_j(\boldsymbol{Z}_{1:k-1}) = L_{jj}$$

- It is also easy to show (!) that affine coupling layers are a **special case** of autoregressive flows.

The "Hole-in-One" Autoregressive Transformation

- So, we can turn a standard multivariate normal into **any** multivariate normal with a linear autoregressive map.
- Turn outs, you can turn a standard multivariate normal into **any** distribution. We just want...

$$X_1 = T_1(Z_1) \sim p(x_1)$$

$$X_2 = T_2(Z_2\,;\, Z_1) \sim p(x_2|x_1)$$

$$X_3 = T_3(Z_3\,;\, Z_1, Z_2) \sim p(x_3|x_1, x_2)$$

$$\vdots$$

$$X_d = T_d(Z_d\,;\, \boldsymbol{Z}_{1:d-1}) \sim p(x_d|\boldsymbol{x}_{1:d})$$

- This is called the **Knothe-Rosenblatt** rearrangement.
- It motivates that maybe we can do "better" in one iteration.

More generally...

- More generally, we can have a **transformer** function $\tau(\cdot\,;\,\boldsymbol{c})$ where $\boldsymbol{c}$ is a parameter vector that is dependent on some elements of $\boldsymbol{Z}$, and have

$$X_k = \tau_k(Z_k; \boldsymbol{c}_k(\boldsymbol{Z}_{1:k-1})), \quad k = 1, \ldots, d.$$

- The transformer **must be invertible**.

- Choosing an **affine** transformer

$$\tau_k(Z_k; c_k(\boldsymbol{Z}_{1:k-1})) = \mu_k(\boldsymbol{Z}_{1:k-1}) + \sigma_k(\boldsymbol{Z}_{1:k-1})Z_k$$

where (μ_k, σ_k) is the (two-dimensional) output of $\boldsymbol{c}_i(\boldsymbol{Z}_{i:i-1})$ that recovers what we have seen thus far (Real NVP and MAF/IAF).

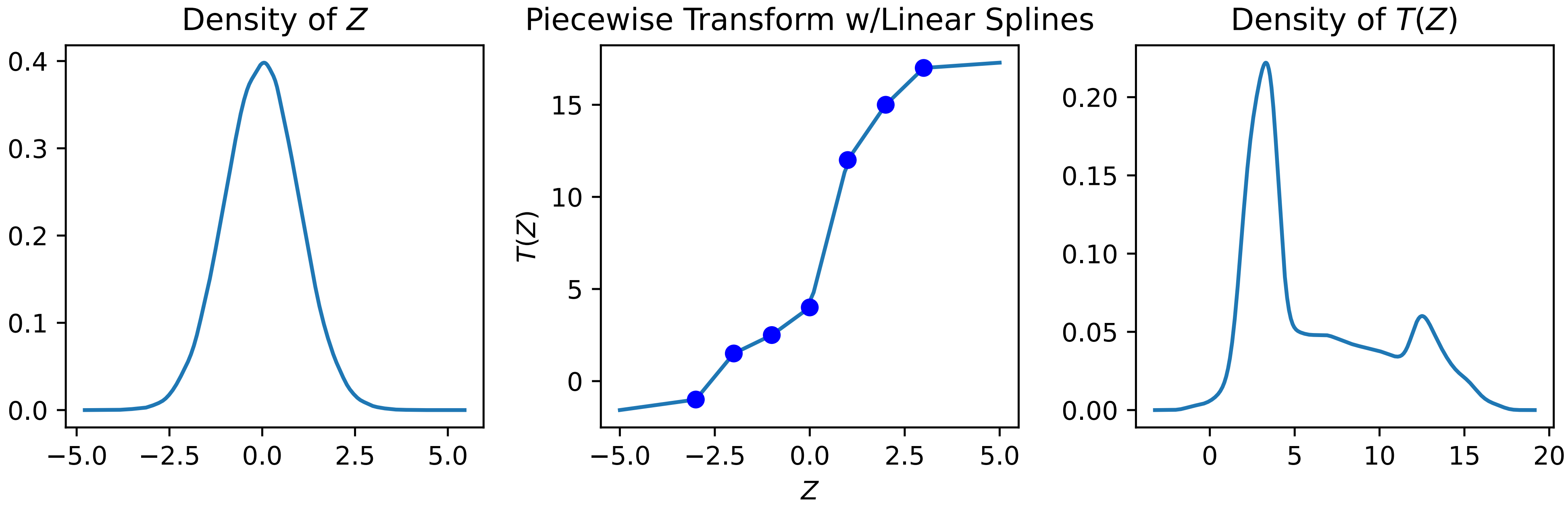
- Conditioner **only needs to satisfy the autoregressive constraint**, with that aside it can do whatever it wants.

- **Key point: The conditioner outputs the parameters for the transformer functions (which are univariate transforms)**.

- Subject to the above, computing the Jacobian determinant term is **still** $\mathcal{O}(n)$.

Splines

Piecewise Linear Splines

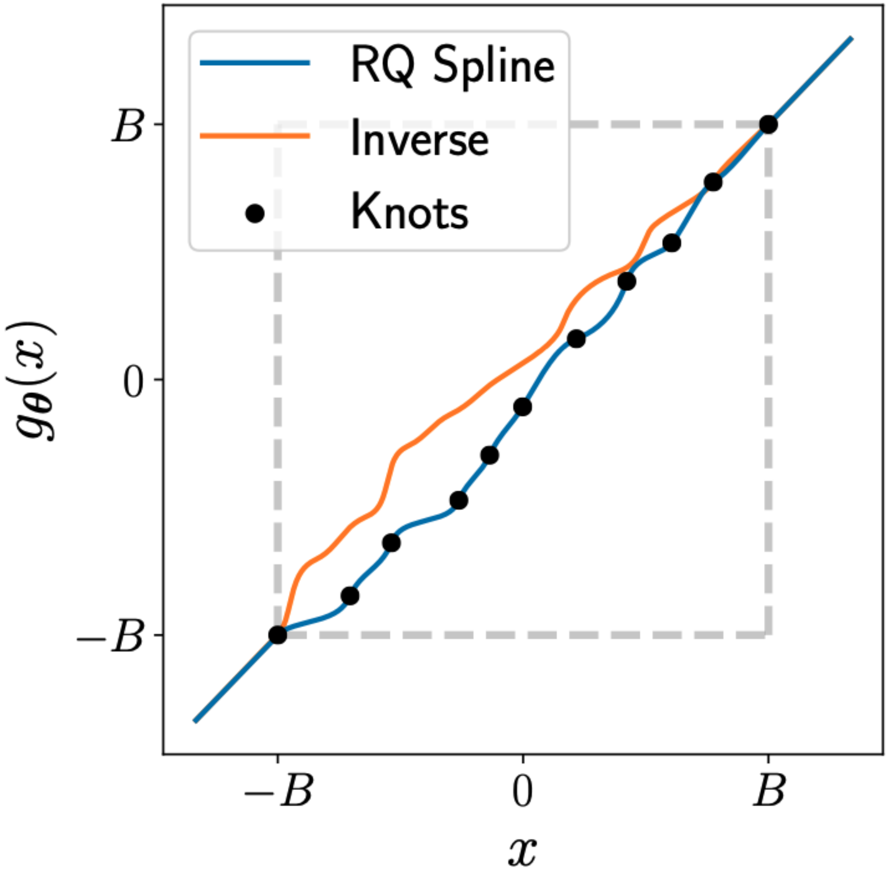


- Remember that the parameters of the spline are given by other variables.

$$X_k = \mathrm{PiecewiseSplines}(Z_k\,; g(\boldsymbol{Z}_{1:k-1})), \quad k = 1, \ldots, d.$$

Beyond Linear Splines

- Figure from "Neural Spline Flows" **(Durkan et al., 2019)**
- State-of-the-art Spline transforms like the Rational Quadratic Splines also have the location of the knots as a parameter.
- For K bins, Linear Rational Splines **(Dolatabadi et al, 2020)** have $4K-1$ parameters for each dimension.
    - The above includes width, height, a special parameter λ, and $K-1$ derivatives at all points except start and end.
- **Don't panic, these are implemented for us already!**



Viewing Maximum Likelihood as Divergence Minimization

- Before we get into seeing some implementations, it is good to have a discussion about MLE.
- We will view it here through the lens of **divergence minimization**, as this will provide a good way of comparing some other things throughout the course.

- The **Kullback-Leibler Divergence**

$$\mathrm{KL}(p||q_{\boldsymbol{\theta}}) = \mathbb{E}_p[\log p(\boldsymbol{X}) - \log q(\boldsymbol{X}\,;\boldsymbol{\theta})] = \mathbb{E}_p \log p(\boldsymbol{X}) - \mathbb{E}_p \log q(\boldsymbol{X};\boldsymbol{\theta})$$

- Note that the first term on the RHS above does not depend on $\boldsymbol{\theta}$, so

$$\begin{aligned} \arg\min\{\mathbb{E}_p \log p(\boldsymbol{X}) - \mathbb{E}_p \log q(\boldsymbol{X};\boldsymbol{\theta})\} &= \arg\min\{-\mathbb{E}_p \log q(\boldsymbol{X};\boldsymbol{\theta})\} \\ &= \arg\max\{\mathbb{E}_p \log q(\boldsymbol{X};\boldsymbol{\theta})\}. \end{aligned}$$

- Substituting the empirical distribution of the data p_{data} for p, we obtain

$$\arg\max\{\mathbb{E}_p \log q(\boldsymbol{X};\boldsymbol{\theta})\} \approx \arg\max\{\mathbb{E}_{p_{\mathrm{data}}} \log q(\boldsymbol{X};\boldsymbol{\theta})\} = \arg\max\left\{\frac{1}{n}\sum_{k=1}^{n} \log q(\boldsymbol{X}_k;\boldsymbol{\theta})\right\}.$$

Minimizing $\mathrm{KL}(p||q)$: How good is it?

- Writing

$$\mathrm{KL}(p||q) = \mathbb{E}_p\left[\log \frac{p(\boldsymbol{X})}{q(\boldsymbol{X};\boldsymbol{\theta})}\right]$$

we see that the above yields an enormous penalty if q is close to zero when p is not.

- This may (or **may not**!) not be desirable depending on one's ultimate goal.
- The above is often called "**Inclusive KL**" or "**Forward KL**", whereas $\mathrm{KL}(q||p)$ is often called "**Exclusive KL**" or "**Reverse KL**".
    - The above discussion/names assume you are minimizing KL by choosing q and that p is your "target".
- In any case, it tells us that we should expect maximising the likelihood to yield distributions that are "**conservative**" in terms of trying to cover the data generating processes areas of high density.

Flow-Based Models in Python (Pyro)

Probabilistic AI with PyTorch

Pyro

- Universal Probabilistic Programming (i.e., Modelling) Language
- Extremely Flexible Variational Inference Engine
- MCMC Engine (including HMC and convergence diagnostics)
- **Normalizing Flows Library**
- Special Neural Network Architectures (e.g., Autoregressive)
- Bayesian Experimental Design Algorithms
- Much more…

PyTorch

Tensors, GPU support, automatic differentiation, distributions, and stochastic backpropagation

Flows: Implementation in Pyro

- The probabilistic programming library **`Pyro`** was developed by Uber AI labs. It is now being developed by a dedicated team at the Broad Institute (and anyone else who wishes to contribute, it is open source!).
- It has, amongst many other things, **a very nice Normalizing Flow library**.
- This is very beneficial, as implementation can be quite involved, especially when it comes to the Spline stuff.

```
import torch
import pyro
import pyro.distributions as dist
```

- Generally all the normalizing flows functions you will need to use are in `pyro.distributions.transforms`
  - `affineautoregressive` is IAF (`spline_autoregressive` is the extension with splines)
  - `affine_coupling` is Real NVP (`spline_coupling` is the extension with splines)
- The spline transforms available are **Linear Rational Splines** (briefly mentioned previously).

pyro.distributions.transforms

- To create a normalizing flow, we must use an object in the `pyro.distributions.transforms` module.
- For our purposes, make sure you use the **helper functions** (e.g., `spline_autoregressive`, not `SplineAutoregressive`).
- These make life very easy and we do not need to even create the underlying neural networks for the flow, **we only need to tell it the architecture**!

```
import torch
import pyro.distributions as dist

d = 4 # dimension of model to be fit
distZ = dist.Normal(torch.zeros(d), torch.ones(d)) # base distribution

# transform specifying using autoregressive flows with an autoregressive
# network having 25 hidden nodes for each of two hidden layers, using splines
# with 8 bins each.
T = dist.transforms.spline_autoregressive(input_dim=d, hidden_dims=[25,25],
                                          count_bins=8)

flow = [T] # create an iterable (list) of transformations
distX = dist.TransformedDistribution(distZ, flow) # create flow-based distribution
```

`TransformedDistribution`

- Each object of class `TransformedDistribution` has the following
  - `log_prob()`: which is its log-pdf (log-likelihood).
  - `clear_cache()`: method which clears the transform's forward-inverse cache
    - Intuitively, `Pyro` stores information when it computes things in one direction to make it easier in the other, this is the cache..
  - `sample()` : this one is self self explanatory
- Recall, we want to train the model via maximum likelihood. Using `Pyro` and `Pytorch`, **it's easy**.

Training Loop

```
def train_flow_model(dataset, distX, params, steps = 2501, lr = 1e-2):
    dataset = torch.tensor(dataset, dtype=torch.float) # ensures correct type
    optimizer = torch.optim.Adam(params, lr=lr) # initializes optimizer of choice

    for step in range(steps): # iterate over training loop
        optimizer.zero_grad() # clear gradients prev. accumulated in parameter tensors
        loss = -distX.log_prob(dataset).sum() # negative log-likelihood
        loss.backward() # accumulate gradients in parameter tensors
        optimizer.step() # take a step in parameter space w/ accumulated gradients
        distX.clear_cache() # always remember to do this!
```

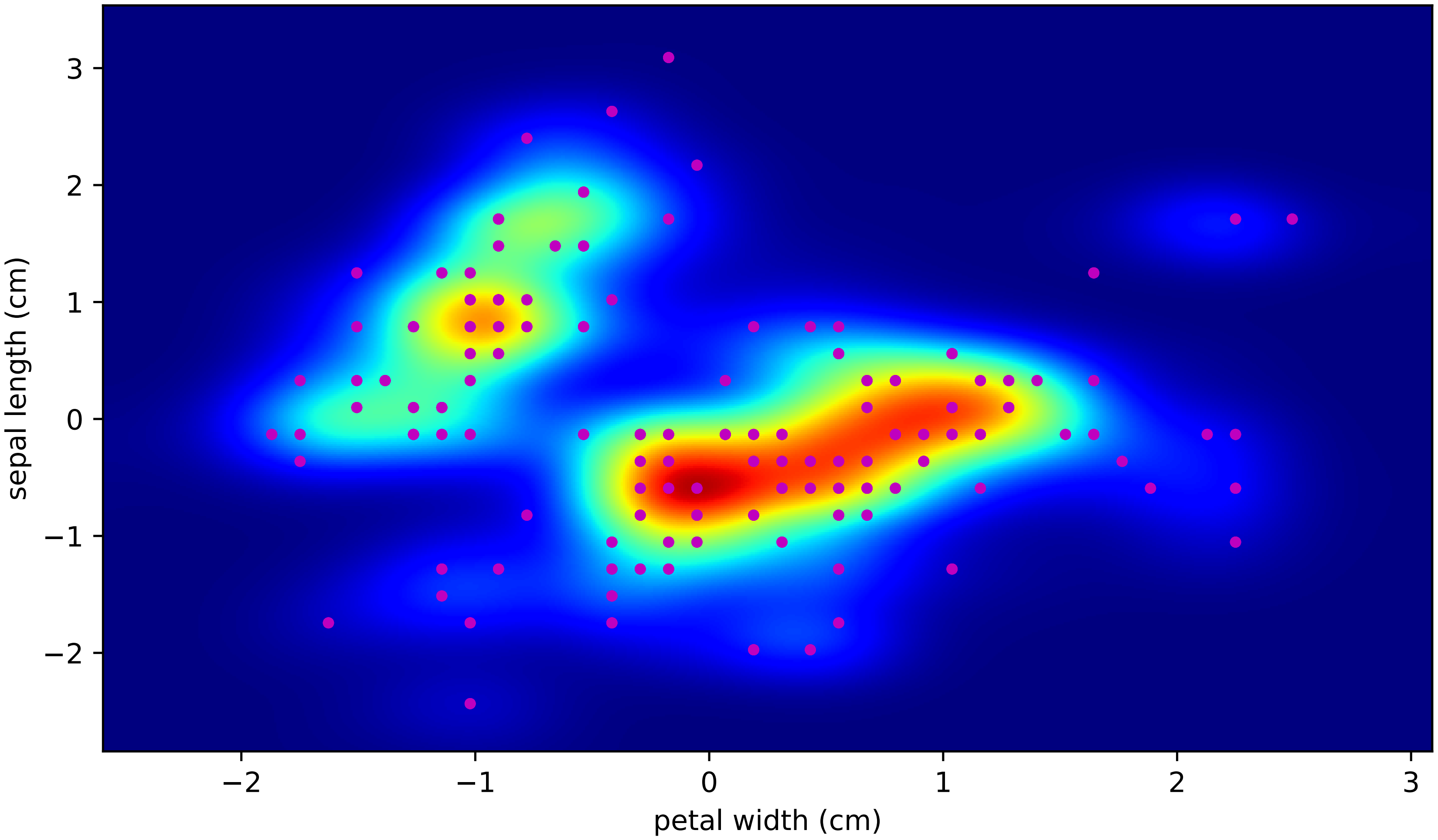
With the above function, we can then call the training procedure. Note that the `params` object is `T.parameters()` which is an **iterable** object.

```
train_flow_model(dataset, T.parameters())
```

Evaluation

- Recall that the `distX.log_prob()` gives us the likelihood for a collection of samples. We are free to evaluate the log-likelihood on some **test set** if we desire.
- We will fit a normalizing flow model to the **Iris** data set which I'm sure you are all familiar with.
- For our 2D case, we can look at the model that is fit...

Result: Normalizing Flow



Evaluation

- The kernel density estimate in the picture I showed you was based on simulating many samples forward. You can also **see how well your flow works in the reverse direction by putting in your original (or test) data and seeing how "normal" the samples look**.
- This is our first expecterience with **independent component estimation** / **disentanglement**. We have learned a way to represent the data (i.e., the data generating distribution) in a way (space) where everything is independent.
- **You will get to do this in Tutorial 1.**

Composition of Transforms: Real NVP with Permutations

```python
import pyro.distributions as dist
from pyro.distributions.transforms import permute, affine_coupling

p = 2 # Data Dimension
distZ = dist.Normal(torch.zeros(p), torch.ones(p)) # Initial Distribution
num_transforms = 5 # how many RealNVP and then Rotations transforms we use

T=[] # empty list to put transfroms in
for i in range(num_transforms):
    if i == 0:  # first one doesn't need to start with a rotation
        T.append(affine_coupling(input_dim = p, hidden_dims=[20,20]))
    else:
        # transformation that reverses the order and then Real NVP
        T.append(permute(input_dim=p, permutation=torch.tensor(range(p-1,-1,-1))))
        T.append(affine_coupling(input_dim = p, hidden_dims=[20,20]))

distX = dist.TransformedDistribution(distZ, T)

# add each transform's parameters into one list (to pass to the optimizer)
params = []
for tr in T:
    if hasattr(tr, 'parameters'): params += list(tr.parameters())
```

Conditional Flows

Conditional Flows

- We can go from approximating $p_{\boldsymbol{X}}$ to modelling a **conditional** distribution $p_{\boldsymbol{X}|\boldsymbol{Y}}$ using **conditional normalizing flows**.

- Really all that happens is that the transformations (neural networks) which normally take $\boldsymbol{Z}$ values will also take in an additional variable as the **context**. For example, Conditional Real NVP would have the transform

$$\boldsymbol{X}_B = \mu(\boldsymbol{Z}_A, \boldsymbol{y}) + \sigma(\boldsymbol{Z}_A, \boldsymbol{y})\boldsymbol{Z}_B$$

- The same principle applies to autoregressive flows and splines. For example, the former may use...

$$X_k = \mu(\boldsymbol{Z}_{1:k-1}, \boldsymbol{y}) + \sigma(\boldsymbol{Z}_{1:k-1}, \boldsymbol{y})Z_k$$

Conditional Flows in Pyro

```
# Base Distribution
distZ = dist.Normal(torch.zeros(2), torch.ones(2))

# Distribution of X1
transX1 = dist.transforms.spline(input_dim = 1)
distX1 = dist.TransformedDistribution(distZ, [transX1])

# Distribution of X1|X2
transX2 = dist.transforms.conditional_spline(input_dim = 1, context_dim=1)
distX2gvnX1 = dist.ConditionalTransformedDistribution(distZ, [transX2])
```

- The above can be useful for **conditional density estimation**.
- You can also factorize your joint distribution how you see fit and train a number of flows jointly.

$$p(\boldsymbol{x}_1)p(\boldsymbol{x}_2|\boldsymbol{x}_1)p(\boldsymbol{x}_3|\boldsymbol{x}_2, \boldsymbol{x}_1)$$

Brief Aside: The Discrete Case

- It is also possible to construct normalizing flows for **discrete** variables.
- Method for categorical variables introduced by *Tran et al (2019)*
  - The `argmax` function is not differentiable, so an approximation is made, this is called the **straight-through** estimator, or the **Gumbel-softmax Trick**.
- Flows for Ordinal Data (*Hoogeboom et al, 2019*)
- In general, discrete flows remain tricky and are a little outside the scope of this course.
- However, in principle, you could fit discrete variables using a discrete flow (or simple method), and then continuous variables using a conditional normalizing flow. After all, it is still fitting a joint as

$$p(\boldsymbol{x}_{\mathrm{d}}, \boldsymbol{x}_{\mathrm{c}}) = p(\boldsymbol{x}_{\mathrm{d}})p(\boldsymbol{x}_{\mathrm{c}}|\boldsymbol{x}_{\mathrm{d}}).$$

where $\boldsymbol{x}_d$ and $\boldsymbol{x}_c$ are discrete and continuous variables, respectively.

Recommended Survey Articles

- In this lecture, I have **restricted myself to discussing the most popular approaches that have both tractable forward and backward operations** .
    - There are many more approaches, some even based on continuous time (e.g., Neural ODEs).
- **Survey Articles**
    1. Papamakarios et al., 2021, Normalizing Flows for Probabilistic Modeling and Inference, Journal of Machine Learning Research (57):1−64.
    2. Kobyzev et al., 2020, Normalizing flows: An introduction and review of current methods. IEEE Transactions on Pattern Analysis and Machine Intelligence. 1-16.
- **Websites**
    - The Github page Awesome Normalizing Flows is an excellent compendium of papers, videos, and software implementations.