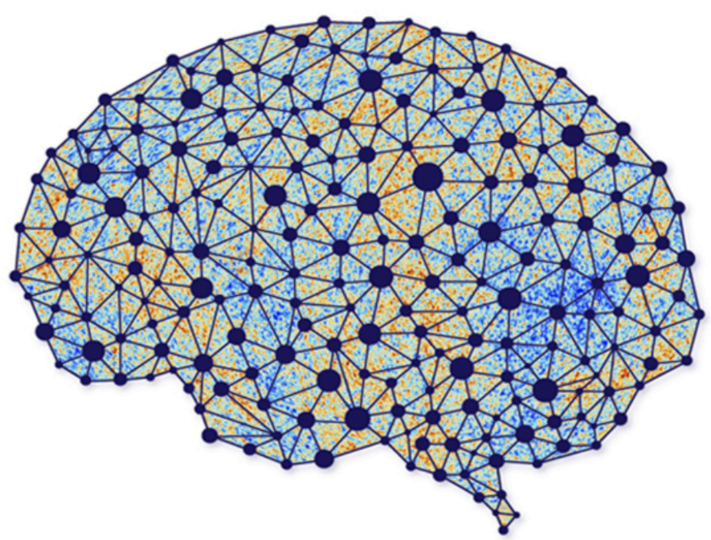
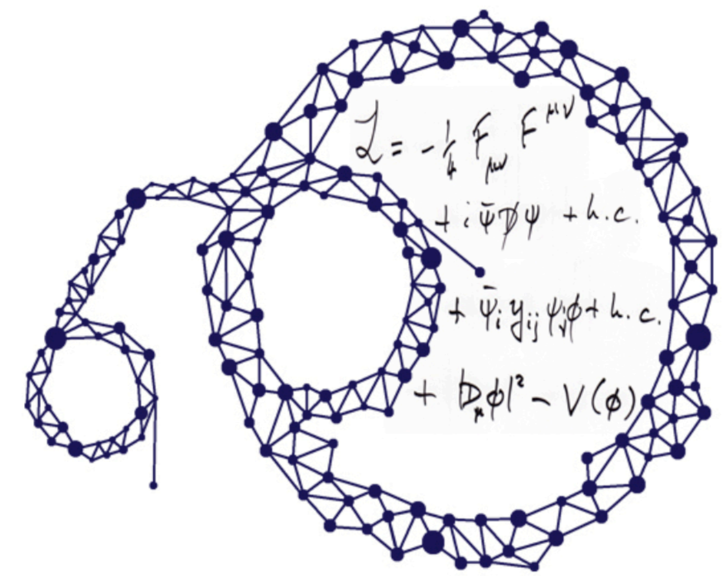
Physics 361 - Machine Learning in Physics

Lecture 11 – Learning PDFs

April 25th 2024



AI
∩
Universe



Moritz Münchmeyer

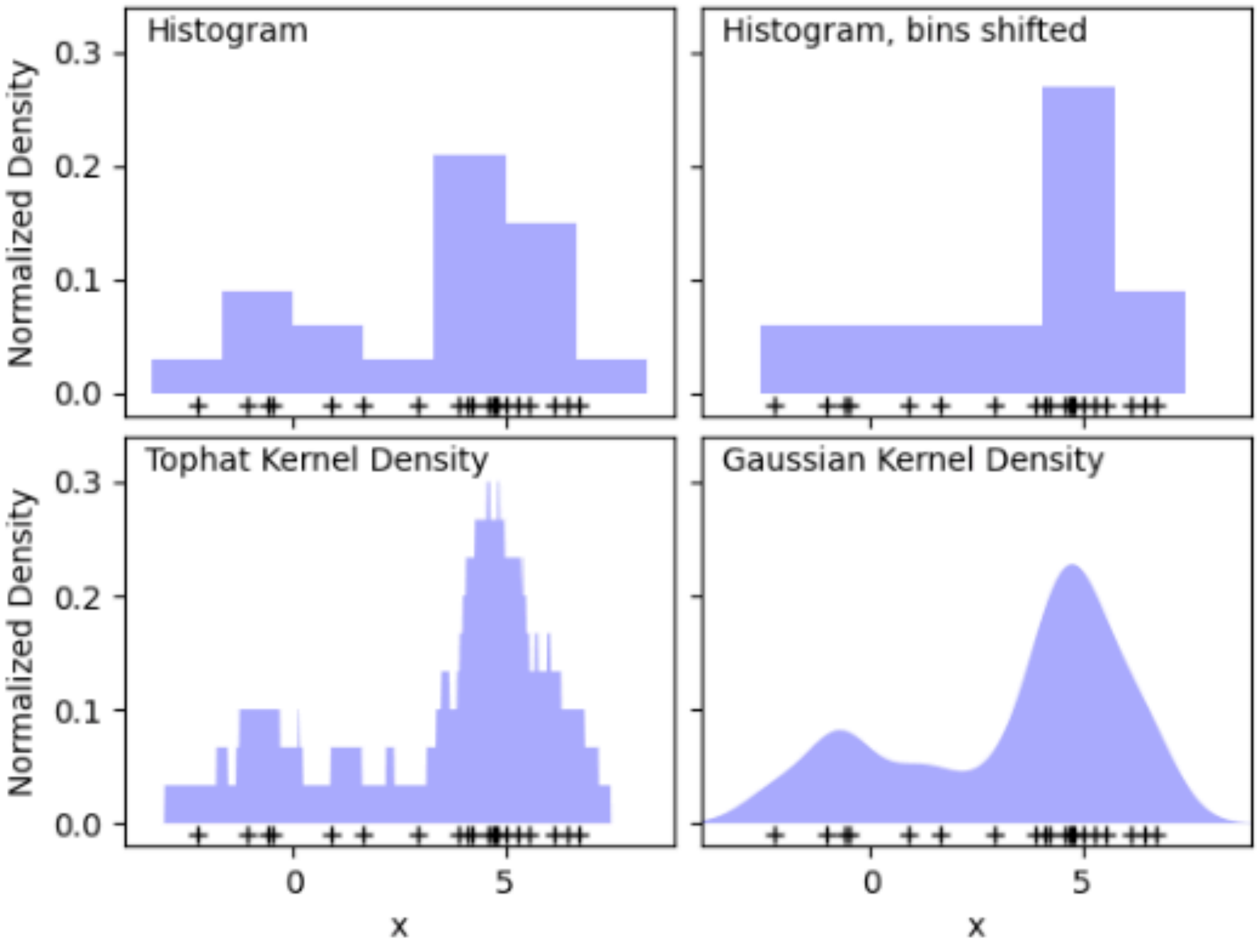
Introduction

- In this unit we will ultimately discuss how to get proper posteriors for measurements made with machine learning, e.g. how to "assign error bars".
- For this we need **probabilistic machine learning**. We need to know **how to learn the probability distribution of data**.
- Thus in the present lecture **we will discuss some main methods for machine learning probability distributions**.
- There are **non-parametric and parametric methods** to learn PDFs.
  - A non-parametric model smoothes the observed data in some way.
  - A parametric model fits a function that has some free parameters to the data, i.e. it makes stronger assumptions about the true PDF.
- For parametric methods, we will discuss those that give us **normalized PDFs**. This is not the case for e.g. diffusion models, which we will cover later.

Learning the parameters of a PDF

Non-parametric methods to learn PDFs

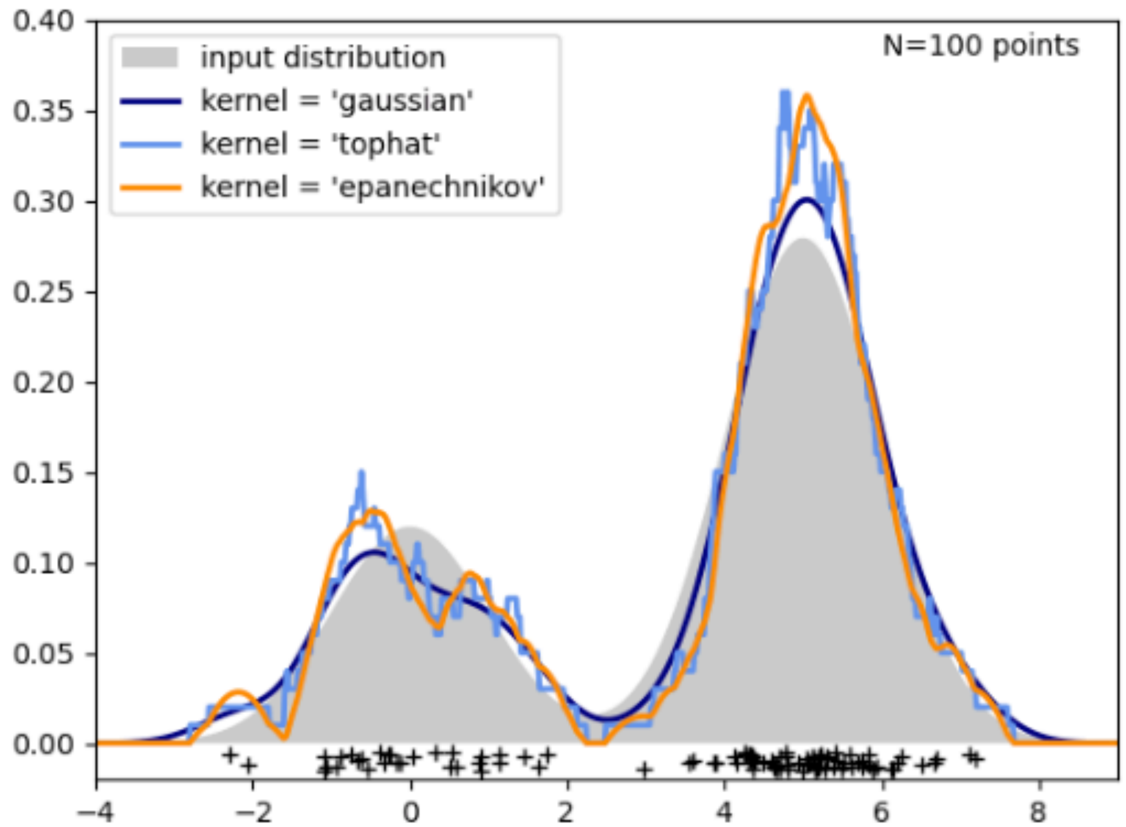
- The **simplest non-parametric method is just histogramming.**
- However:
  - Choice of binning can have a large effect
  - Does not work in high dimension



https://scikit-learn.org/stable/modules/density.html#

Kernel density estimators

- KDE work by smoothing the data with some Kernel, such as a Gaussian.
- In the following figure, 100 points are drawn from a bimodal distribution, and the kernel density estimates are shown for three choices of kernels:



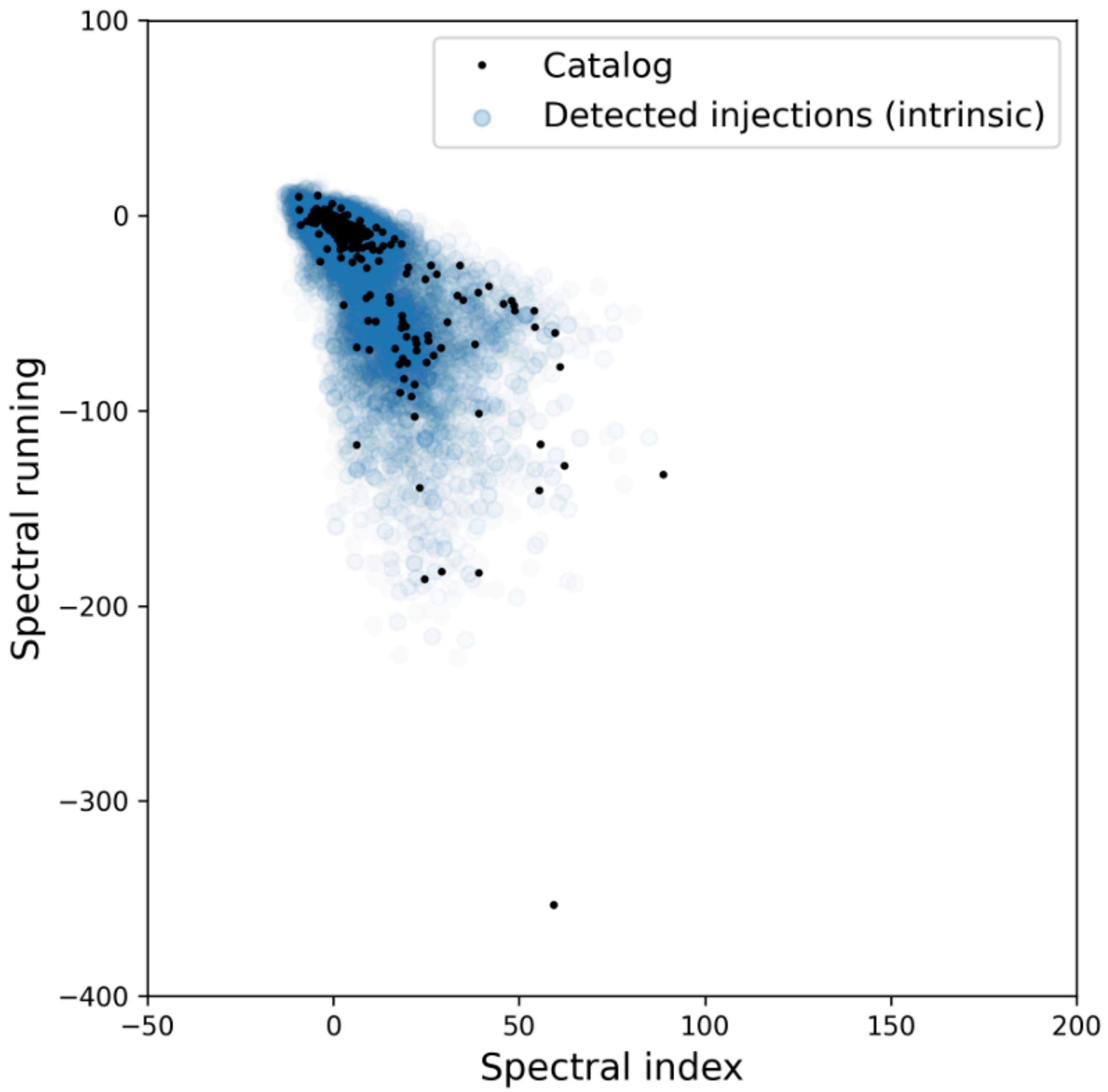
https://scikit-learn.org/stable/modules/density.html

Kernel density estimators

- **Kernel Density Estimators (KDEs)** belong to the class of non-parametric methods for estimating probability density functions (PDFs).
- Unlike parametric methods (such as Gaussian distributions) that assume a specific functional form of the PDF, KDEs make minimal assumptions. Instead, **they estimate the PDF directly from the data using kernels.**
- However:
  - **Struggles in high dimension**s due to the curse of dimensionality, where the data becomes sparse and the estimator requires exponentially more samples.
  - Limited to smoothing data points with a kernel function, which may not capture intricate patterns.

Example: KDE from my research

- **Population model in CHIME FRB catalogue https://arxiv.org/pdf/2106.04352**
- **Goal was to make synthetic data (blue) that is similar to the observed one (black) but explores a somewhat larger domain.**



https://scikit-learn.org/stable/modules/density.html

Parametric methods to learn PDFs

- We now discuss parametric methods to learn PDFs.
- In this case, we specify some functional form that we believe the PDF to be in, up to a number of free PFD parameters which we aim to learn.
- We have some data {x} and want to learn the parameters theta of the PDF that describes the data (assuming it is i.i.d distributed):

$$p(x_i|\theta)$$

- We can do this by having a training data set {x}. Once the PDF is learned, we can draw new samples that were not in the training data.

Learning parametric models with maximum likelihood

- Idea: Choose parameters that maximize the likelihood of observing the given data.

$$\theta^* = \arg\max_{\theta} \prod_{i=1}^{n} p(x_i|\theta)$$

- Or equivalently

$$\theta^* = \arg\max_{\theta} \sum_{i=1}^{n} \log p(x_i|\theta)$$

- Example for the Gaussian

$$p(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

- We get the analytic

$$\mu_{MLE} = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad \sigma^2_{MLE} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

Maximum Likelihood with gradient descent

- Recall: Choose parameters that maximize the likelihood of observing the given data.

$$\theta^* = \arg\max_{\theta} \sum_{i=1}^{n} \log p(x_i|\theta)$$

- When the PDF or likelihood is complex, **gradient-based methods** like stochastic gradient descent (SGD) are used to maximize the likelihood or log-likelihood.
- We'll use this for more **complex PDFs using "Normalizing flows" below.**
- **Gradient descent is used to fit the parameters of the PDF to make the training data maximally likely under the PDF.**

How to learn a conditional PDF

- It is easy to generalize maximum likelihood to the case of a conditional PDF:

Objective: Find parameters θ that maximize the conditional likelihood $P(Y|X,\theta)$

Optimization Problem:

$$\hat{\theta} = \arg\max_{\theta} \sum_{i=1}^{n} \log P(y_i|x_i,\theta)$$

Example: Learn posterior $P(\text{parameters} \mid \text{data})$ from simulated pairs $\{\text{parameters}, \text{data}\}_i$

$\Longrightarrow$ see next lecture

Other methods to learn PDFs

- **Method of Moments:** Match the theoretical moments of the distribution (mean, variance, skewness, etc.) to the empirical moments from the data.

$$\mathbb{E}[X] = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad \mathbb{E}[X^2] = \frac{1}{n}\sum_{i=1}^{n} x_i^2, \ldots$$

  - When the MLE involves complex optimization or derivatives that are difficult to compute, the MoM can provide a quick and straightforward alternative.
- **Expectation-Maximization (EM) Algorithm:** Used when data is incomplete or has latent variables. The algorithm iteratively estimates latent variables (E-step) and updates parameters (M-step) until convergence.
  - Example: **Mixture models such as Gaussian Mixture Models (GMMs).**
  - **See next section**
- **Bayesian Inference:** Treat the parameters themselves as random variables and update their distributions based on observed data.

Gaussian Mixture Models (GMM)

Gaussian Mixture Model (GMM)

- Generative model often used in the context of clustering.

- Points are drawn from one of the K Gaussians, with its own μ_k & Σ_k:

$$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \sim \exp\left[-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})^T\right]$$

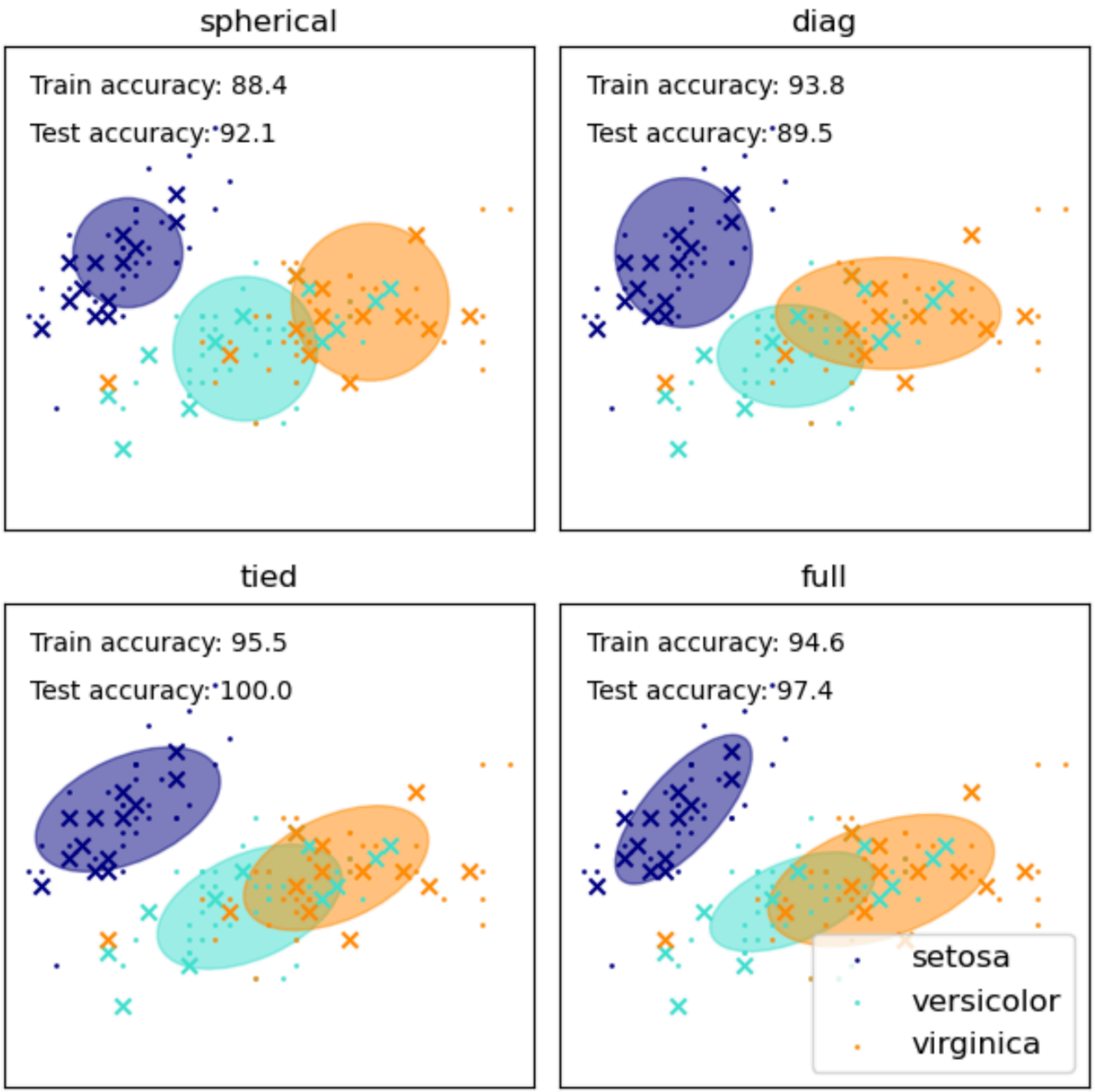
- π_k = Probability a pt is drawn from mixture k, the probability of generating a point $\mathbf{x}$ in a GMM is:

$$p(\boldsymbol{x}|\{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\}) = \sum_{k=1}^{K} \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\pi_k.$$

- Given a dataset $\mathbf{X} = \{\mathbf{x}_1, \ldots \mathbf{x}_N\}$, the likelihood of the dataset:

$$p(\boldsymbol{X}|\{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\}) = \prod_{i=1}^{N} p(\boldsymbol{x}_i|\{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \pi_k\})$$

- Denote the set of parameters $\{\mu_k, \Sigma_k, \pi_k\}$ by θ.



https://scikit-learn.org/stable/modules/mixture.html

Gaussian Mixture Model (GMM)

- Common cost function is Maximum likelihood estimation (MLE).

- Latent variables are chosen to maximize the likelihood of the observed data under our generative model → Expectation-Maximization (EM) equations.

- Latent variable $\mathbf{z} = (z_1, \ldots, z_K)$ for point $\mathbf{x}$ has the property that $z_k = 1$ if $\mathbf{x}$ is drawn from the k-th Gaussian, and $z_{j \neq k} = 0$.

- Probability of observing a datapoint $\mathbf{x}$ given $\mathbf{z}$:

$$p(\boldsymbol{x}|\boldsymbol{z}; \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}) = \prod_{k=1}^{K} \mathcal{N}(\boldsymbol{x}|\mu_k, \Sigma_k)^{z_k}$$

- Probability of observing a given value of latent variable:

$$p(\boldsymbol{z}|\{\pi_k\}) = \prod_{k=1}^{K} \pi_k^{z_k}$$

Gaussian Mixture Model (GMM)

- Joint probability of a clustering assignment $\mathbf{z}$ and a datapoint $\mathbf{x}$:

$$p(\boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\theta}) = p(\boldsymbol{x}|\boldsymbol{z}; \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\})p(\boldsymbol{z}|\{\pi_k\}).$$

- Conditional probability of the data point in the k-th cluster, $\gamma(z_k)$, given model parameters θ is

$$\gamma(z_k) \equiv p(z_k = 1|\boldsymbol{x}; \theta) = \frac{\pi_k \mathcal{N}(\boldsymbol{x}|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x|\mu_j, \Sigma_j)}.$$

known as the "responsibility" that mixture k takes for explaining $\mathbf{x}$.

Training GMM with the EM algorithm

- Recall:

A GMM models the probability of each data point x_i as:

$$p(x_i \mid \theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x_i \mid \mu_k, \Sigma_k)$$

- K is the number of Gaussian components.
- π_k are the mixing coefficients (weights), with $\sum_{k=1}^{K} \pi_k = 1$.
- $\mathcal{N}(x_i \mid \mu_k, \Sigma_k)$ is the Gaussian distribution with mean μ_k and covariance Σ_k.

- The Expectation-Maximization algorithm performs the following steps iteratively:

1. Initialization

- Randomly initialize:
  - Component means μ_k
  - Covariance matrices Σ_k
  - Mixing coefficients π_k

2. E-Step (Expectation Step)

In this step, we calculate the **responsibilities** — the probability that each data point belongs to each Gaussian component:

$$\gamma_{ik} = \frac{\pi_k \mathcal{N}(x_i \mid \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_i \mid \mu_j, \Sigma_j)}$$

Where:

- γ_{ik} is the responsibility of the k^{th} component for the i^{th} data point.

3. M-Step (Maximization Step)

In this step, we update the parameters μ_k, Σ_k, and π_k to maximize the expected log-likelihood:

1. **Update Means**

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^{N} \gamma_{ik} x_i$$

2. **Update Covariances**

$$\Sigma_k = \frac{1}{N_k} \sum_{i=1}^{N} \gamma_{ik} (x_i - \mu_k)(x_i - \mu_k)^T$$

3. **Update Mixing Coefficients**

$$\pi_k = \frac{N_k}{N}, \quad \text{where } N_k = \sum_{i=1}^{N} \gamma_{ik}$$

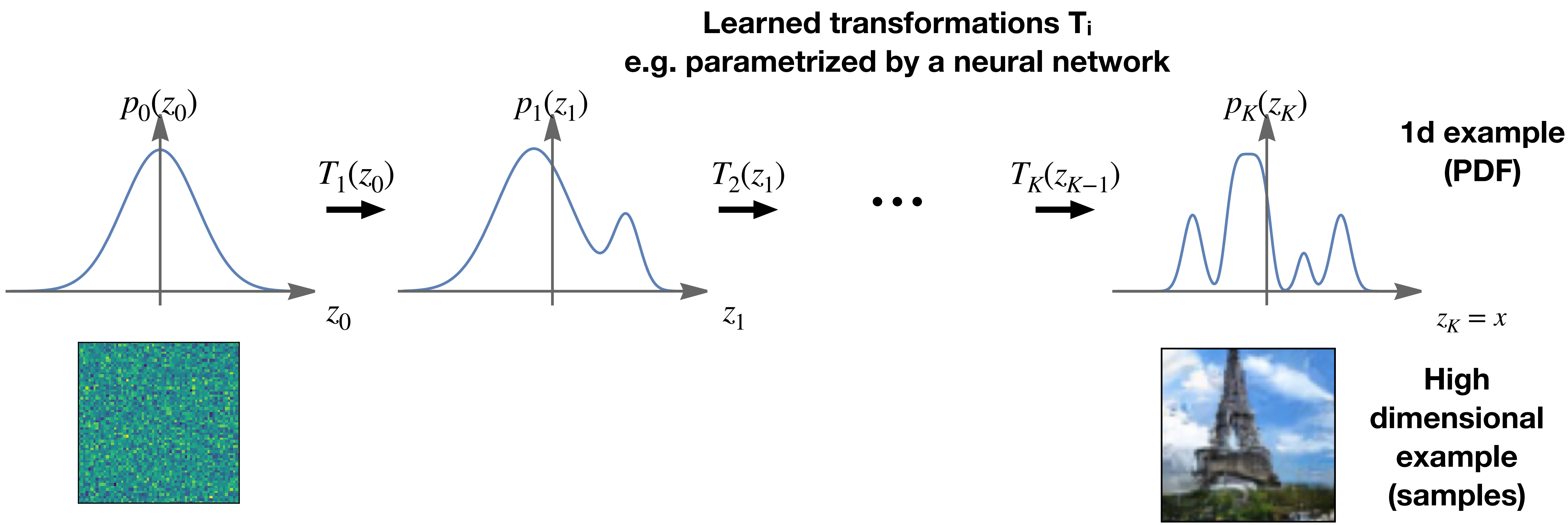
The EM algorithm is used because directly maximizing the likelihood of a GMM is difficult due to latent variables (cluster assignments). **EM provides an iterative approach that alternates between estimating latent variables (E-step) and optimizing parameters (M-step).**

Normalizing Flows

Introduction

Normalizing flows

- Normalizing flow: Series of learned transformations that **deform a simple base distribution into a complicated target distribution**.



- Difference with most other ML methods: We learn a probability distribution, rather than an arbitrary input->output mapping.

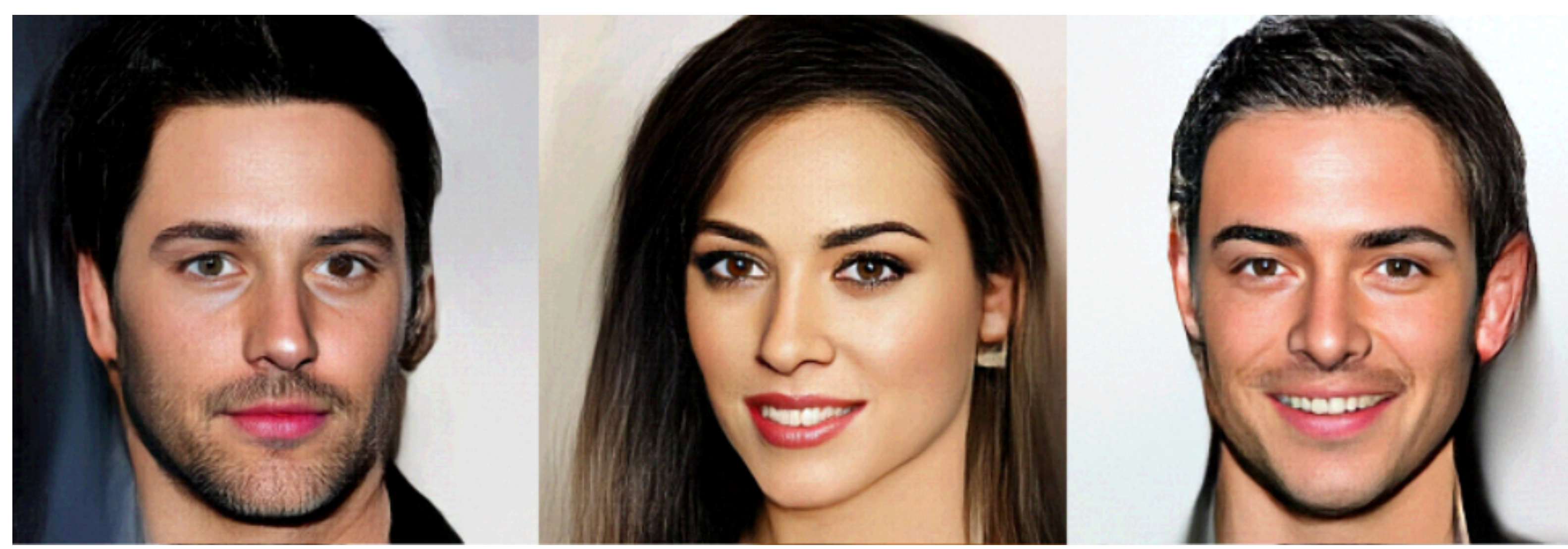
- Review: https://arxiv.org/abs/1912.02762. Widely used in physics e.g. in QFT, likelihood-free inference and cosmology

Normalizing flows are generative models

- Like GANs and diffusion models, normalizing flows are generative models.
- They can be used to generate images too. However they are not currently as good at that as these other models.
- But they can do something other models cannot: give a normalized probability density for the sample. The are real PDFs.

Real-NVP flow

Glow flow



Normalizing flows

- Transformation T (the "flow")

$$\mathbf{x} = T(\mathbf{u})$$

Change of variables of PDF

$$p_x(\mathbf{x}) = p_{\mathrm{u}}(\mathbf{u}) \; |\det J_T(\mathbf{u})|^{-1}$$

- Chain many "simple" transformations together to make a complicated distribution:

$$T = T_K \circ \cdots \circ T_1$$

$$\mathbf{z}_k = T_k(\mathbf{z}_{k-1})$$

- After training **two basic operations can be performed**:
  - **Exact density evaluation** (backward mode)

    Sample x → **p(x)**

  - **Sampling from the distribution** (forward mode)

    Base distribution sample u → **Target sample x**

Basics and definitions

- data distribution: $\vec{x}$: D-dimensional real vector

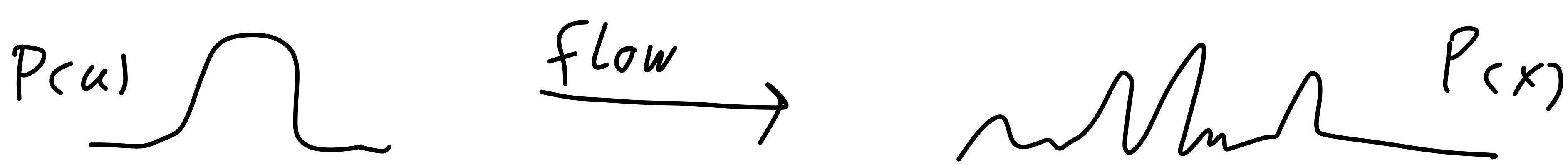
  $\vec{x} \sim P(\vec{x})$

  E.g.: $\vec{x} = \begin{pmatrix} \text{height} \\ \text{age} \\ \text{weight} \\ \text{strength} \end{pmatrix}$ of people

- Main idea of normalizing flows:
  Express $\vec{x}$ as a transformation T of a real vector $\vec{u}$ sampled from a simple base distribution.

$$\vec{x} = T(\vec{u}) \quad \text{where} \quad \vec{u} \sim P(\vec{u})$$

E.g.: $\vec{u}$ is a Gaussian random variable



Properties of the transformation

- To specify $p(x)$ we need to specify
  - The transformation $T(x;\theta)$ with „learned" parameters θ
  - The base distribution $p_u(u;\phi)$ with „learned" parameters ϕ } often kept static
- Defining properties of transformation T:
  - T must be invertible
  - Both T and T^{-1} must be differentiable

  Thus T is a diffeomorphism.
- $\Longrightarrow$ $u = T^{-1}(x)$ must also be D-dimensional.

Properties of the transformation

- The flow transformation is a change of variables.

$$p_x(x) = p_u(u) \left| \det J_T(u) \right|^{-1}$$

where $u = T^{-1}(x)$

with Jacobian

$$J_T(u) = \begin{bmatrix} \frac{\partial T_1}{\partial u_1} & \cdots & \frac{\partial T_1}{\partial u_D} \\ \vdots & \ddots & \\ \frac{\partial T_D}{\partial u_1} & & \frac{\partial T_D}{\partial u_D} \end{bmatrix}$$

$\left| \det J_T(u) \right|$ quantifies the local change of volume that „molds" $p(u)$ into $p(x)$.

Composition of transformations

- Invertible and differentiable transformations are composable. In practice we stack many simple base transformations:

$$T = T_K \circ T_{K-1} \circ \dots \circ T_1$$

$$\det J_{T_2 \circ T_1}(u) = \det J_{T_2}(T_1(u)) \det J_{T_1}(u) \quad \text{etc.}$$

- Example: Flow from Gaussian to cross shape with 4 transf.

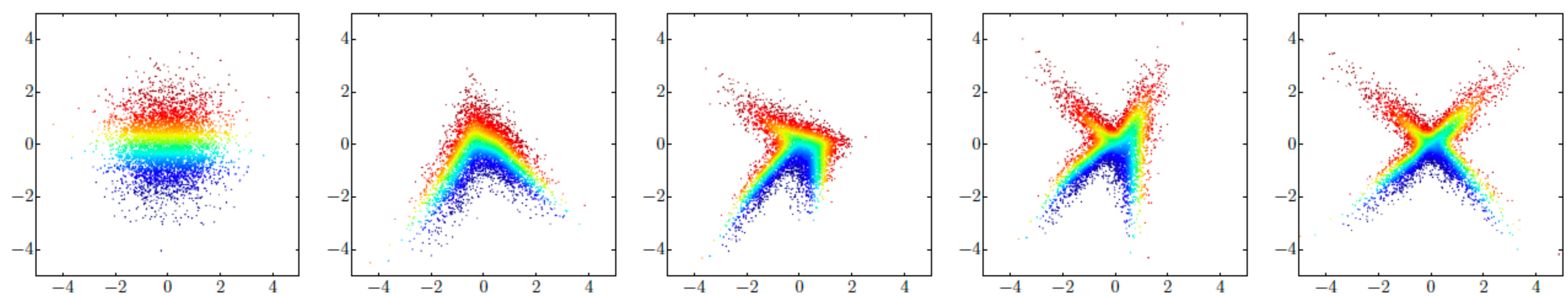


Figure 1: Example of a 4-step flow transforming samples from a standard-normal base density to a cross-shaped target density.

Forward and backward use of the flow

- The flow (after training) has two fundamental operations:
- **Sampling**: Draw samples from $p(u)$ and transform

$$x = T(u)$$

to get samples from x. Other generative models (GANs, VAE) can also sample.

- **Density evaluation**: From a sample x, we can calculate $p(x)$ as

$$p_x(x) = p_u(u) \, |\det J(u)|^{-1}$$

Computational tradeoffs

- Sampling and density evaluation have different computational requirements.
- Density evaluation requires calculating T^{-1} and $\det J_{T^{-1}}$.
- For large D, these calculations can easily be forbiddingly expensive.

$\Longrightarrow$ Need flow architectures that are both flexible (expressive) and fast.

- T must be easily invertible. Most functions are not.
- Some flows are universal approximators, some aren't, and sometimes it is unknown.

Training the flow

- Goal: Fit a flow $p_x(x;\theta)$ to target distribution $p_x^*(x)$

  parameters of transformation $T(x;\theta)$
  and sometimes prior $p(u;\theta)$

- Usually we have a collection of samples from $p_x^*(x)$, the training data.

- There are different measures of similarities between p_x and $p^*(x)$.

- The most popular choice: Kullback-Leibler (KL) divergence

Training the flow: KL divergence

- The loss is given by

(see lecture 4)

$$\mathcal{L}(\theta) = D_{KL}\left[p_x^* \,||\, p_x(x;\theta) \right]$$

$$= -\mathbb{E}_{p_x^*}\left[\log p_x(x;\theta) \right] + \text{const}$$

make samples from training data likely under flow pdf

- using $p_x(x) = p_u(T^{-1}(x)) \,|\det J_{T^{-1}}(x)|$

$$\Rightarrow \quad \mathcal{L}(\theta) = -\frac{1}{N}\sum_{n=1}^{N} \log p_u(T^{-1}(x_n,\theta)) + \log|\det J_{T^{-1}}(x_n;\theta)|$$

- Training: SGD for $\nabla_\theta \mathcal{L}$

Designing normalizing flows

Designing efficient flows

- We stack many simple building blocks:

$$T = T_K \circ \circ \circ T_1$$

$$z_K = T_K(z_{K-1})$$

$$\log |J_T(z)| = \sum_{K=1}^{K} \log |J_{T_K}(z_{K-1})|$$

more depth → $O(K)$ growth in cost :)

- Note: making T_K invertible in theory and actually inverting it are very different requirements

$\Rightarrow$ want easily invertible functions

Designing efficient flows

- We want a tractable Jacobian determinant.

  For a general map D inputs $\rightarrow$ D outputs calculating $\det J$ has cost $\mathcal{O}(D^3)$, often intractable for large D.

  Most flows are using forms for which $\det J$ is $\mathcal{O}(D)$.

- We now discuss building blocks T_K that have this feature.

$$\vec{z}' = f(\vec{z})$$

Planar flows

- Building block:

elementwise non-linearity

1505.05770

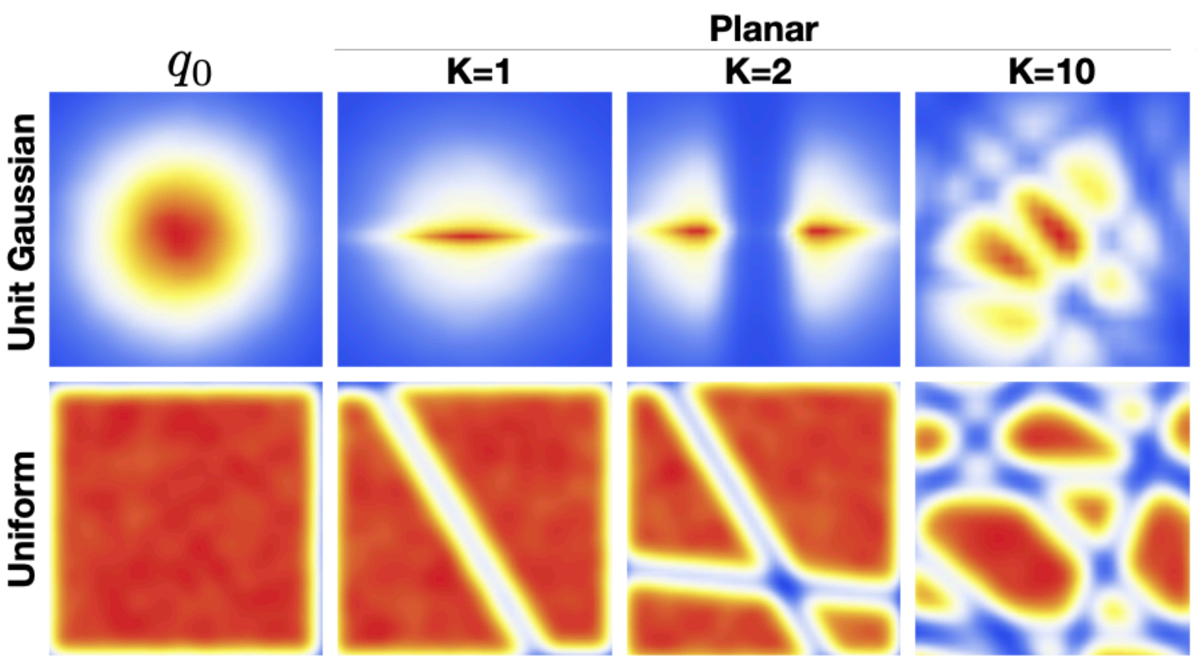
$$f(\vec{z}) = \vec{z} + \vec{u}\, h(\vec{w}^T \vec{z} + b)$$

vectors dim D

- can calculate det J in $\mathcal{O}(D)$ time.
- stack many of these transformations

- Example

z : 2dim.



Not suitable for large dimensions, since transformations are too local (i.e. affect a small volume)

Autoregressive flows

- Autoregressive property: $\vec{z} = (z_1, z_2, z_3 \dots z_i \dots z_D)$

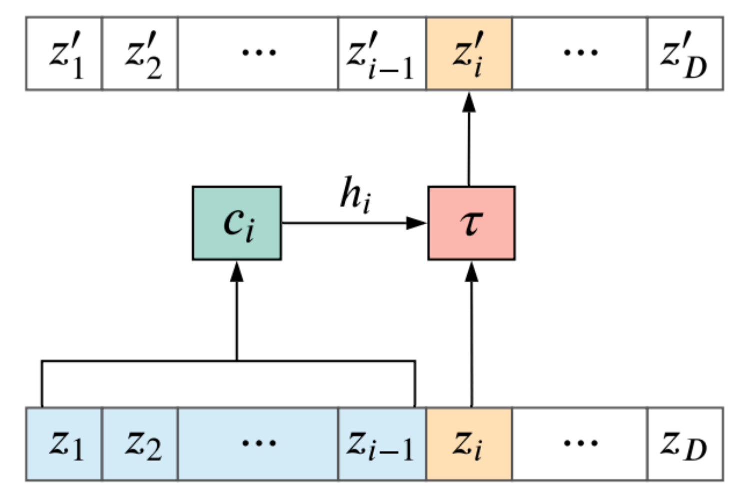
  variable z_i depends on $z_{1:i}$ only

  $$z'_i = \tau(z_i, \vec{h}_i)$$

  „transformer": must be invertible. i.e. monotonic in z_i

  $$\vec{h}_i = c_i(\vec{z}_{<i})$$

  „conditioner": modulates the transformer



- This leads to a triangular Jacobian:

$$J_T(z) = \begin{bmatrix} \diagdown & 0 \\ \sim & \end{bmatrix} \Rightarrow \det J = \prod_i J_{ii} \qquad \mathcal{O}(d)$$

Affine Autoregressive flows

- Particalarly simple Linear transformer

$$\tau(z_i) = \alpha_i z_i + \beta_i$$

shift and scale z_i

"affine transformation"

invertible for $\alpha \neq 0$

or: $\exp^{\alpha_i} z_i + \beta_i$

- The conditioner here is:

α, β depend on $\{z_{<i}\}$ by some neural network parametrisation

$$\{z_{<i}\} \longrightarrow \boxed{NN_\Theta} \longrightarrow \begin{matrix} \alpha_i \\ \beta_i \end{matrix}$$

NN: e.g. MLP or CNN

So the conditioner has the Learned parameters Θ.

Different autoregressive flows

- Again, autoregressive flows are

$$z_i' = \tau(z_i, \vec{h}_i)$$

„transformer": must be invertible.
i.e. monotonic in z_i

$$\vec{h}_i = c_i(\vec{z}_{<i})$$

„conditioner": modulates the transformer. not a bijection.

- Many different transformers and conditioners have been proposed. Popular:
  - Masked autoregressive flows (MAF)
  - Inverse autoregressive flows (IAF)

They have different cost and expressivity.

Illustration of the MAF flow

Source: 2310.03741

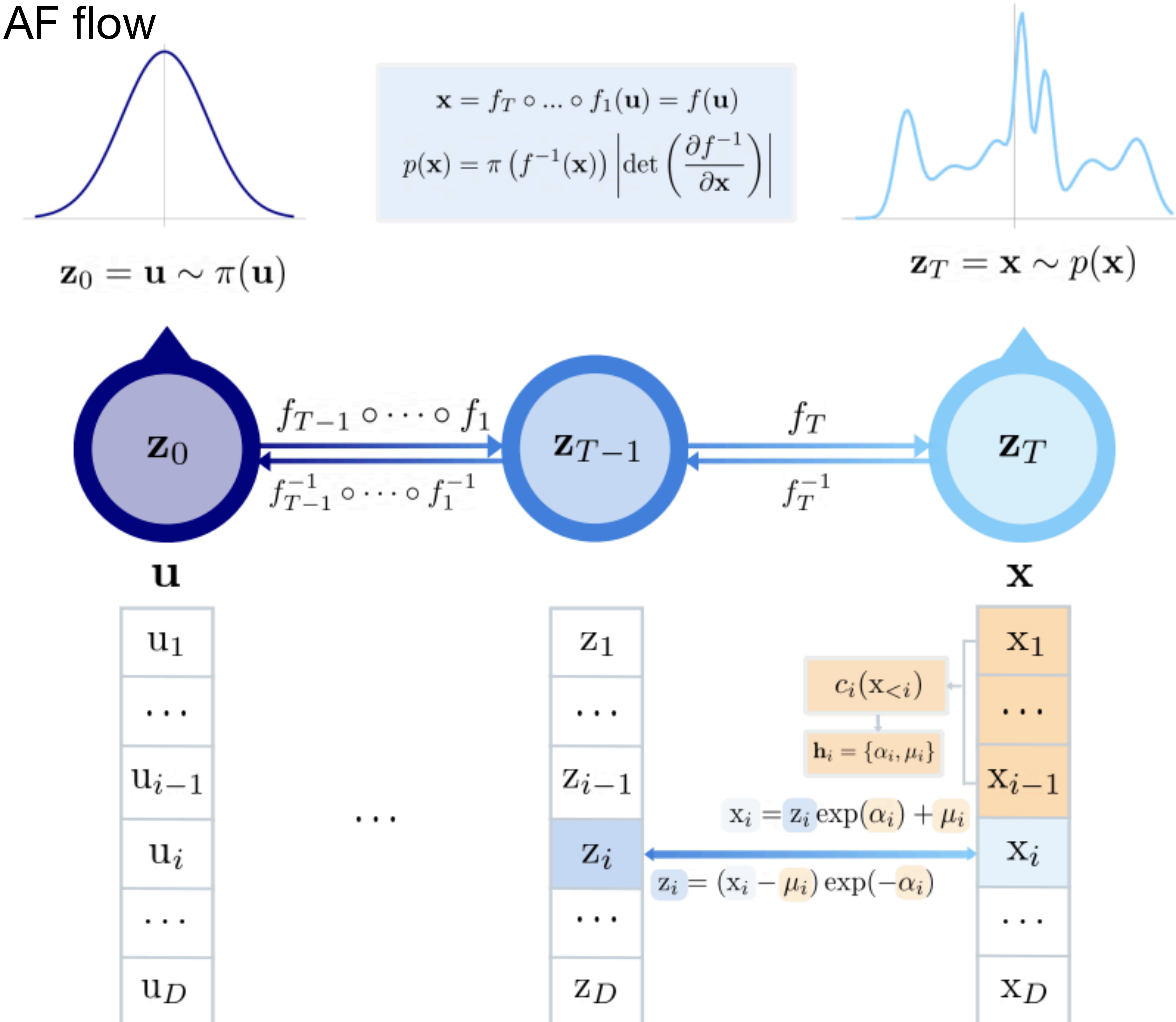


Figure 2. Diagram of how normalizing flows work, with the specific example of Masked Autoregressive Flows. The samples from the vector $\mathbf{z}_0 = \mathbf{u}$, sampled from the simple distribution $\pi(\mathbf{u})$, are deformed through the sequence of transformations $f = f_T \circ \cdots \circ f_1$ into those of $\mathbf{z}_T = \mathbf{x}$, which follow a more complex distribution $p(\mathbf{x})$. In the lower panel, we illustrate the conditioner that "masks out" the connections between z_i and $\mathbf{h}_{<i}$, as well as the affine functions applied to the vector components.

Real-NVP

2d spatial array

- Baseline flow for many „image" applications
  1605.08803

- Special case of affine autoregressive flow:
  Split variables into two halfs:

$$z'_{1:K} = z_{1:K}$$

$$z'_{K+1:D} = z_{K+1:d}\, \alpha(z_{1:K}) + \beta(z_{1:K})$$

$\Rightarrow$ Jacobian $\left(\begin{array}{ccc|ccc} 1 & & & & & \\ & 1 & & & 0 & \\ & & 1 & & & \\ \hline \cdot & \cdots & \cdot & \ddots & & 0 \\ \vdots & \ddots & \vdots & & \ddots & \\ \cdot & \cdots & \cdot & 0 & & \ddots \end{array}\right)$

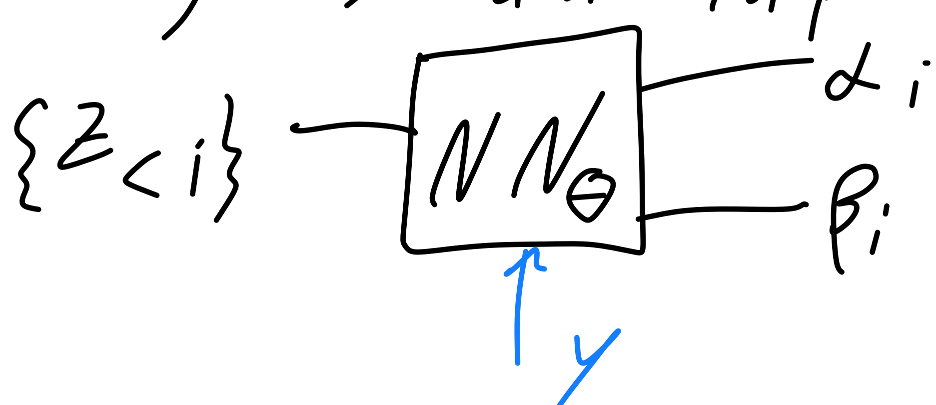
Allows parrallel computation of z' since all inputs are available.

- stack many such layers with different orderings of the variables.

Use example: https://arxiv.org/abs/2101.08176 Normalizing Flows for Lattice Field Theory

Conditional flows

- We want to learn not just PDFs $p(x)$ but also conditional PDFs
$$p(x \mid y)$$
- This is achieved by making the flow transformations T dependent on the condition y.
- For example, in a autoregressive flow, the neural network which parametrizes z now also gets an input of y:



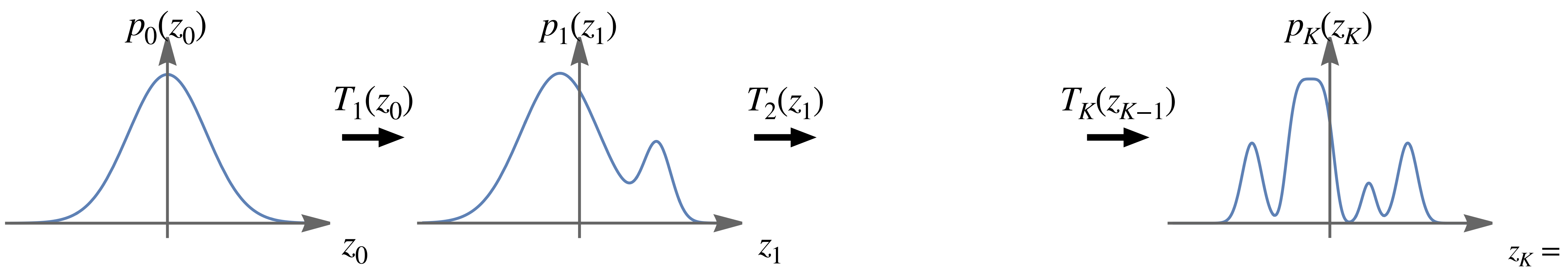
- Training works the same as before (KL-div.).

Aside: Normalizing flows to model the matter distribution in cosmology

(Research example from my group)

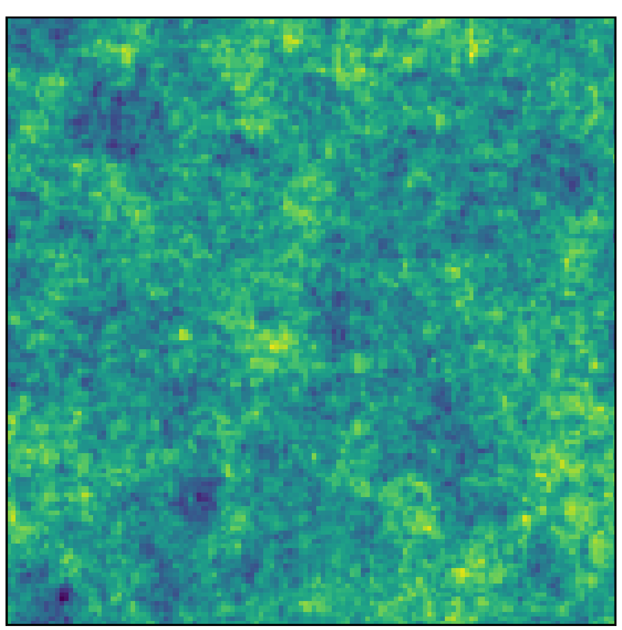
NFs vs structure formation

Gaussian initial conditions PDF morphs into complicated late-time matter distribution.

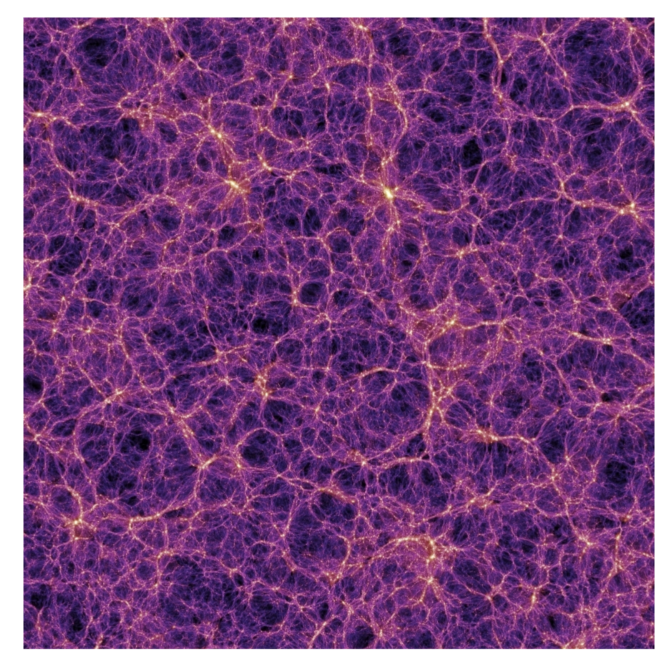


Gaussian primordial matter perturbations

Non-gaussian matter/galaxy distribution today



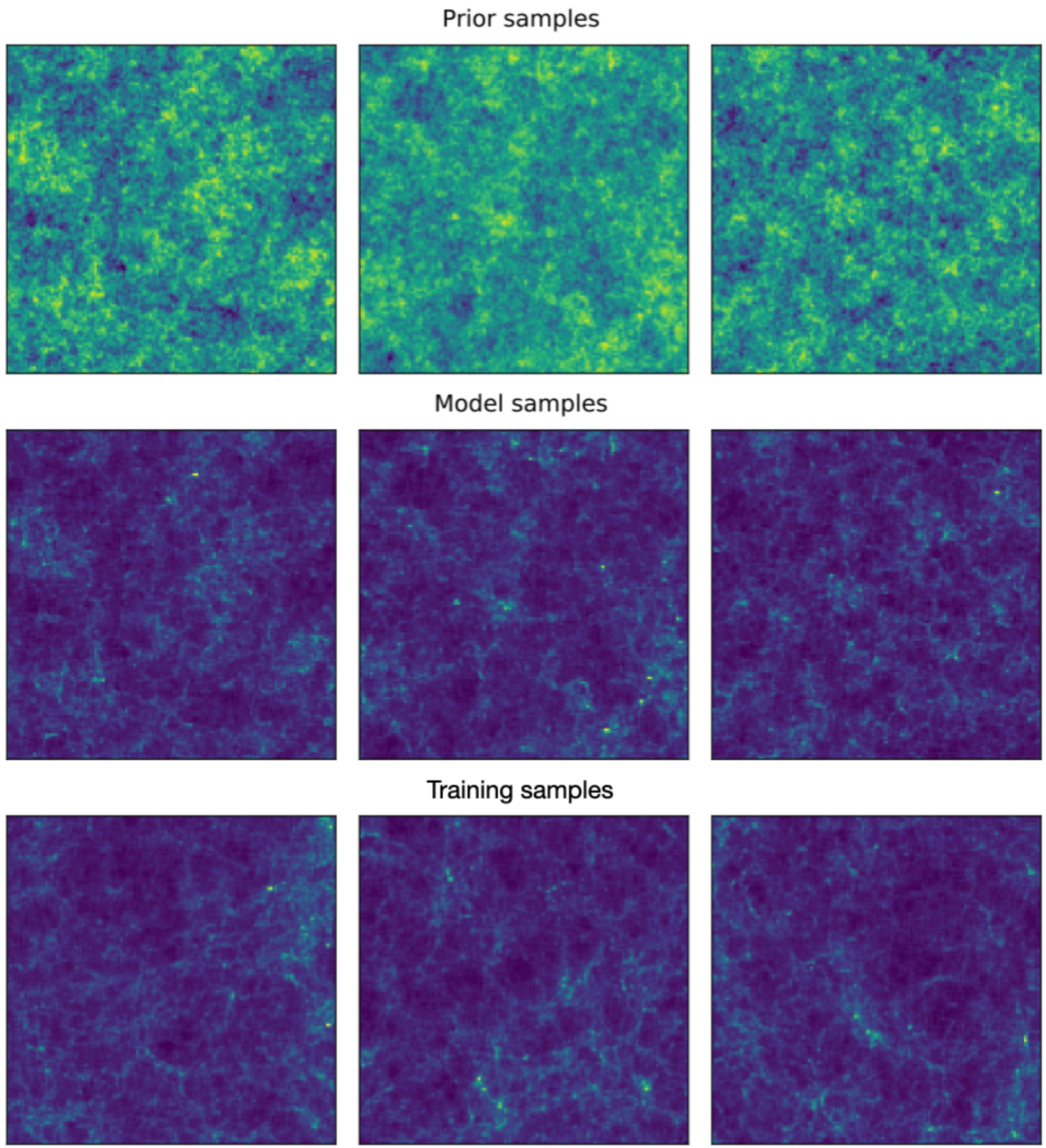
Cosmological time evolution



Rouhiainen, MM: [arXiv:2105.12024]() Normalizing flows for random fields in cosmology

Flowing from a correlated Gaussian to todays matter distribution

Flow:
RealNVP



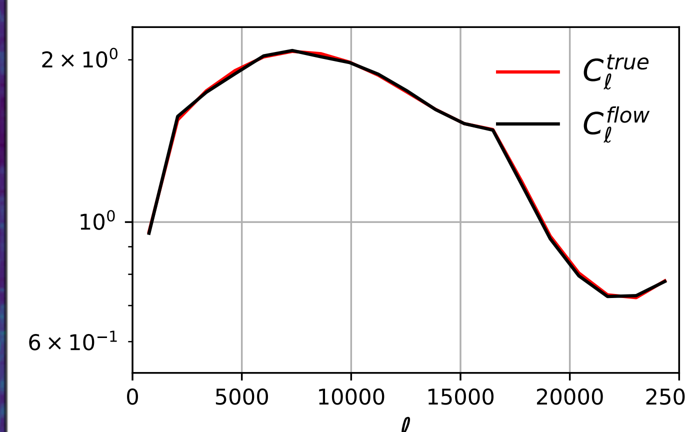
Density peaks match, as in physical structure formation.

The flow learned to deform a Gaussian PDF into a highly non-Gaussian PDF

$125\ h^{-1}$ Mpc

About 128^3 particles per field

Power spectrum matches



De-noising with a Generative Prior

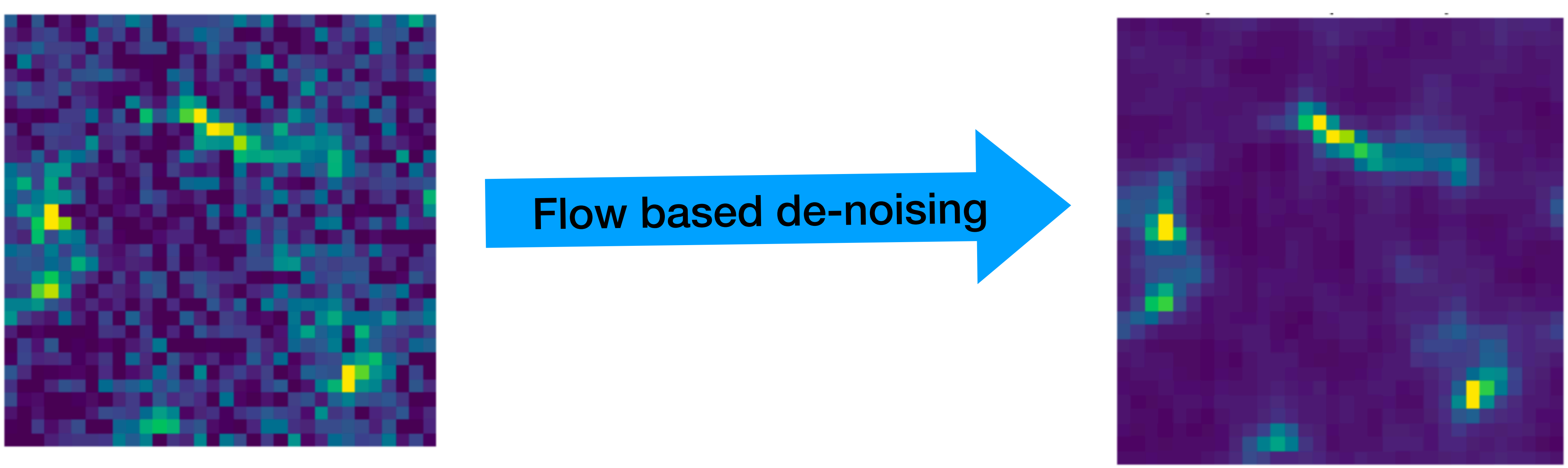
In data analysis in cosmology we often make use of **Gaussian priors (Wiener Filter)**. This is no longer justified for very high resolution observations. Using the trained normalizing flow we can **include non-Gaussian priors:**

$$\ln p(y\,|\,d) = -\frac{1}{2}(y-d)^{\mathrm{T}} N^{-1}(y-d) - \ln p_{\mathrm{flow}}(y)$$

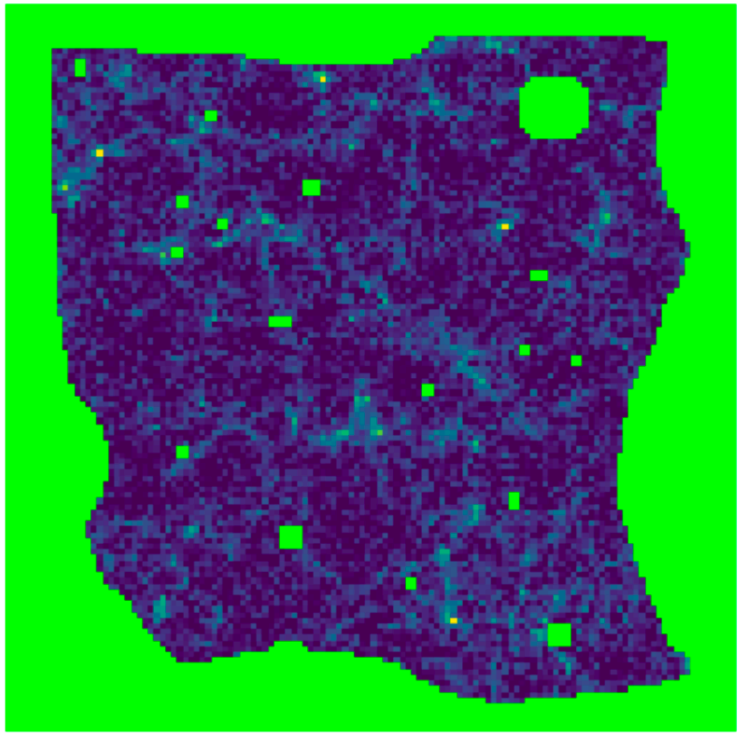
True matter field

Noisy observation

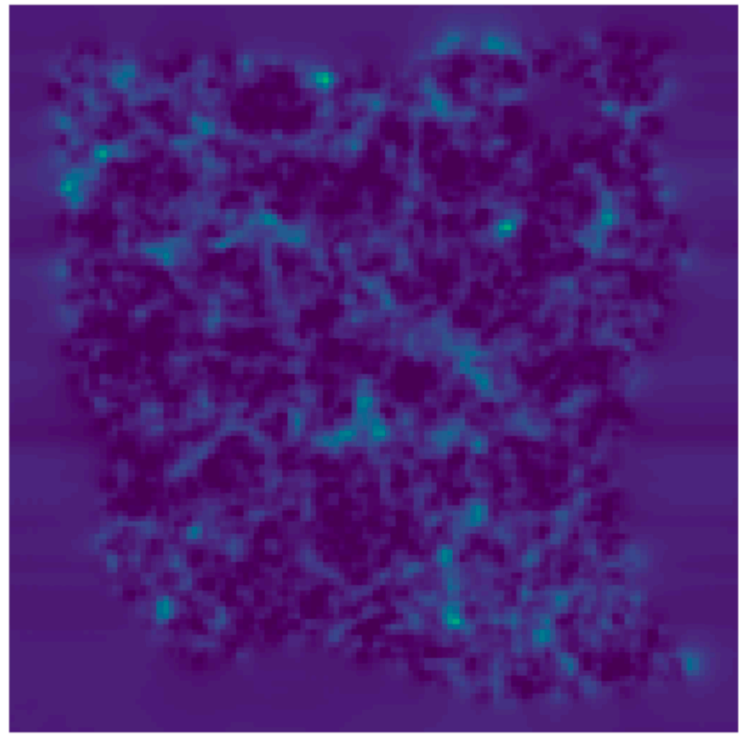
We use a flow trained on simulations of the matter distribution. Then we use this knowledge of the matter PDF to **de-noise an observation of the matter field by maximizing the posterior**.



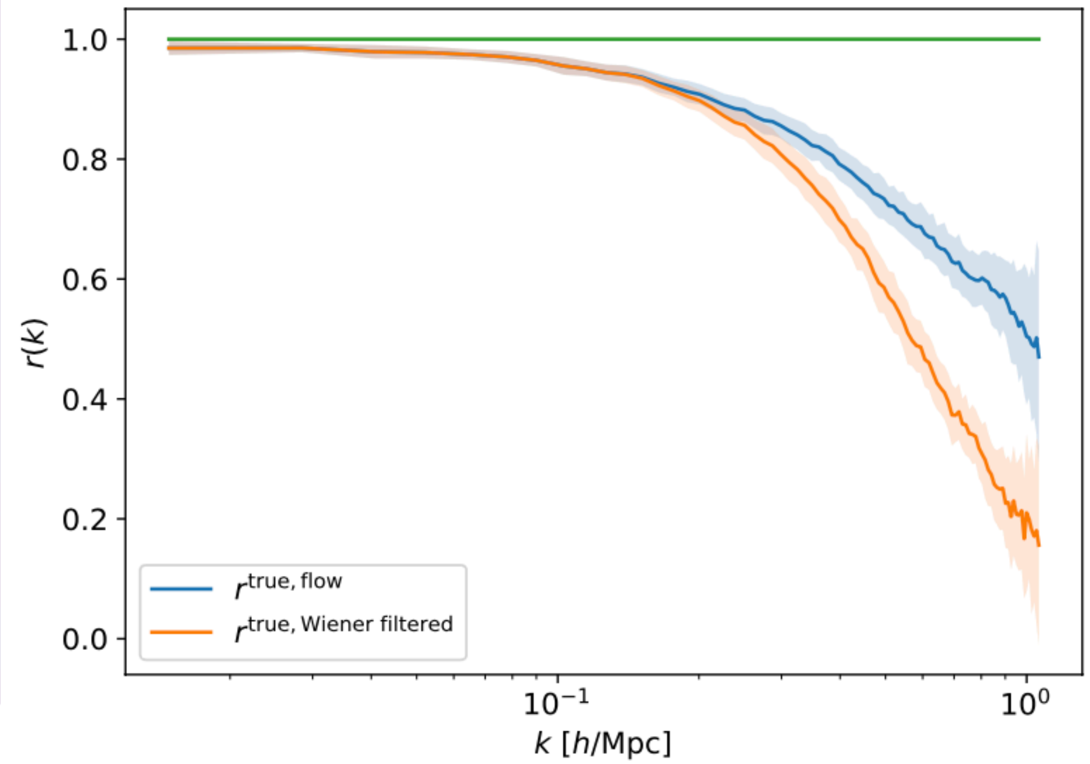
De-noising the observed matter field

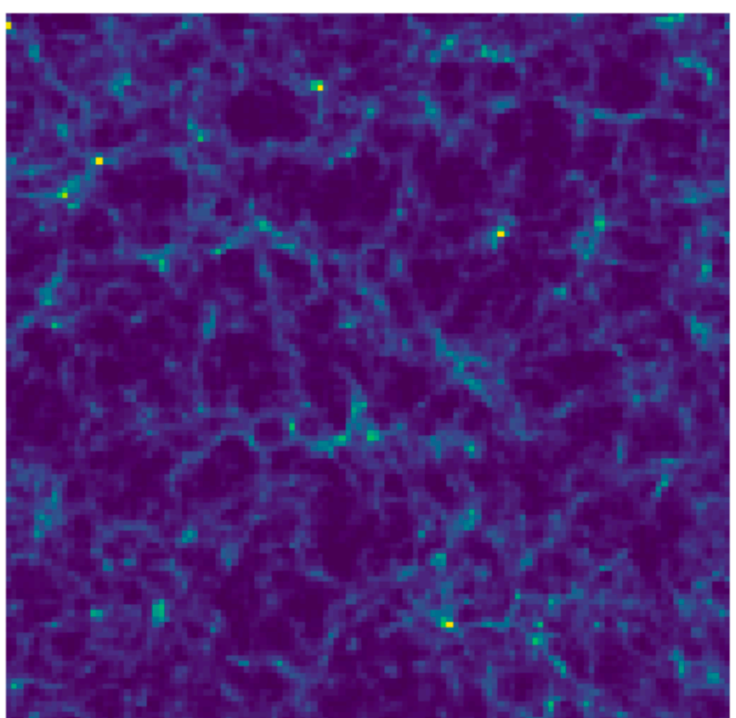


Observed (noisy, masked)

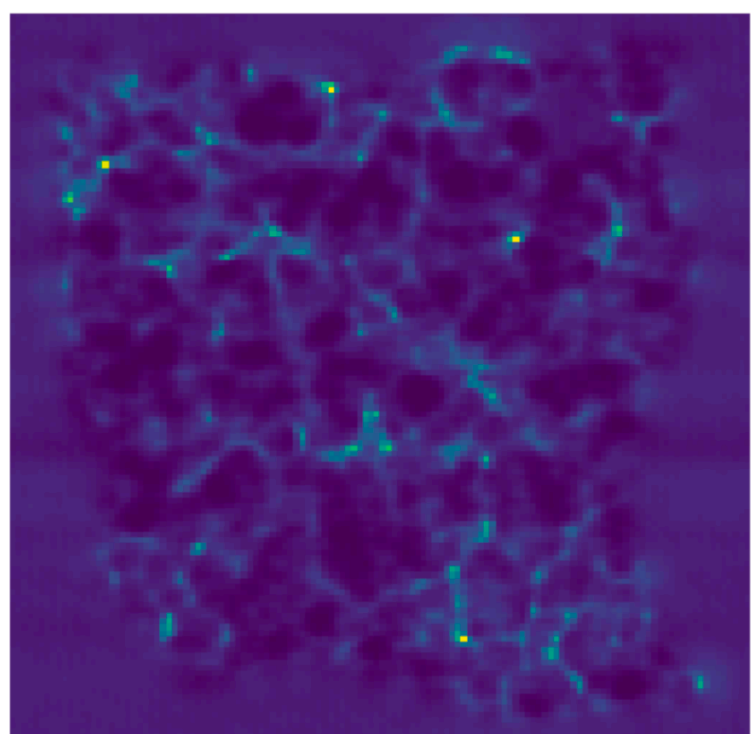


Wiener filtered





Truth



Flow **MAP**

As expected, the NF lowers the reconstruction noise on non-linear scales compared to the Wiener filter.

Generative de-noising is useful in many other domains.

Rouhiainen, MM: arXiv:2211.15161 De-noising non-Gaussian fields in cosmology with normalizing flows

Course logistics

- **Reading for this lecture:**
    - I did not use a specific primary reference for this lecture. However some of the main textbooks on the website cover these topics.