

# Normalizing flows

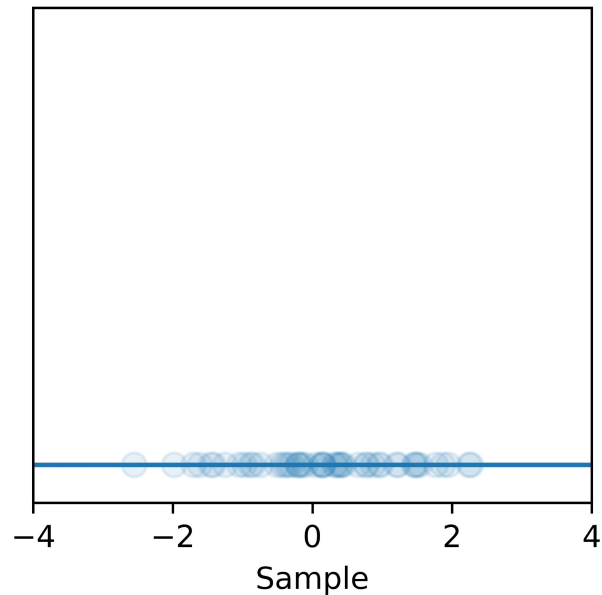
David Nabergoj

University of Ljubljana, Faculty of Computer and Information Science  
October 12, 2023, Ljubljana

# Finding well-fitting distributions

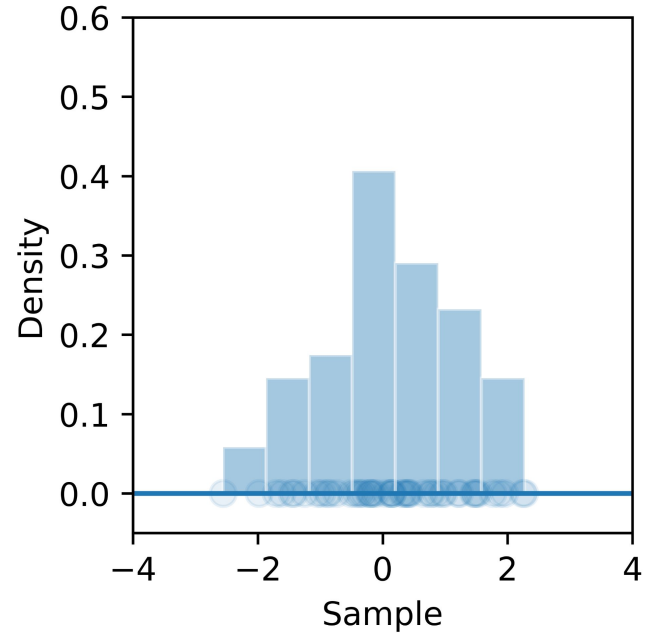
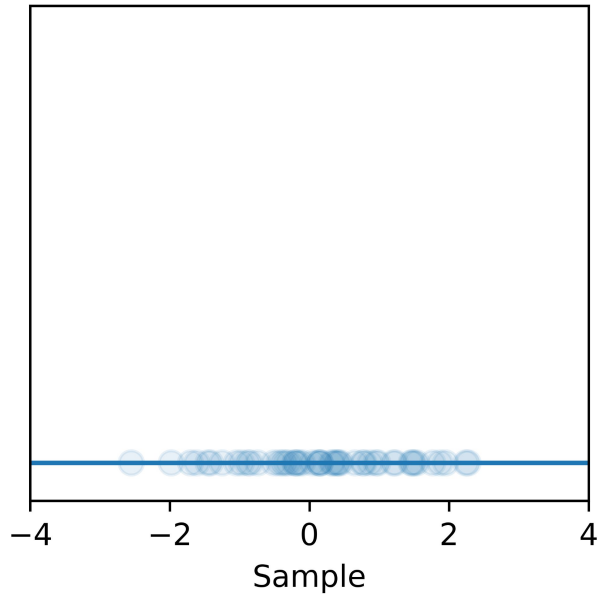
- Let's take some samples from a 1D data-generating process
- What distribution would be a good fit?

$D = [... -0.98 \ 0.95 \ -0.15 \ -0.10 \ 0.41 \ 0.14 \ 1.45 \ ...]$



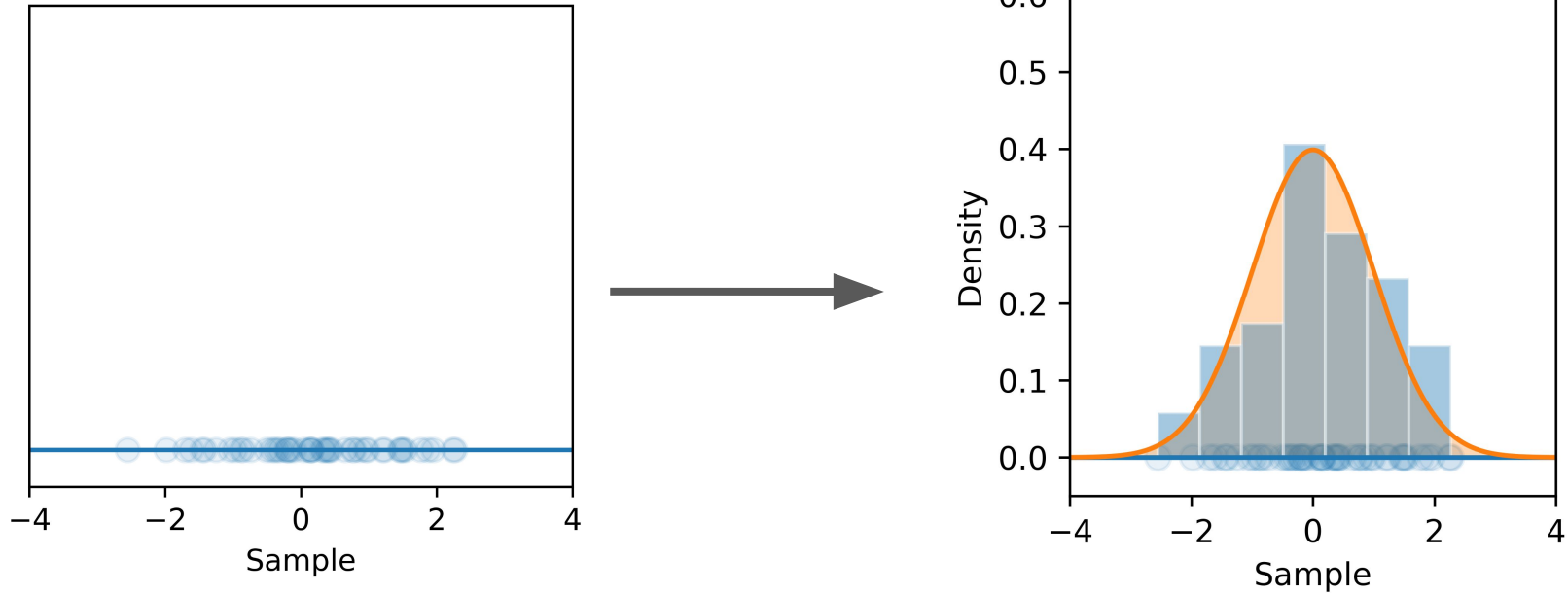
# Finding well-fitting distributions

- Let's look at a histogram



# Finding well-fitting distributions

- A **normal distribution** would be a decent choice



# Finding well-fitting distributions

- How do we find the best mean and variance for our normal?
- > We search by maximizing the probability of our data.

$$\begin{aligned}\max_{\mu, \sigma} P(D|\mu, \sigma) &= \max_{\mu, \sigma} \prod_{i=1}^n p(x_i|\mu, \sigma) \\ &= \max_{\mu, \sigma} \log \prod_{i=1}^n p(x_i|\mu, \sigma) = \max_{\mu, \sigma} \sum_{i=1}^n \log p(x_i|\mu, \sigma)\end{aligned}$$

# Finding well-fitting distributions

- > Plug in the log of the normal density...

$$\begin{aligned} \max_{\mu, \sigma} \sum_{i=1}^n \log p(x_i | \mu, \sigma) &= \\ \max_{\mu, \sigma} \sum_{i=1}^n -\log \left( \sqrt{2\pi\sigma} \right) - \frac{1}{2} \left( \frac{x_i - \mu}{\sigma} \right)^2 &= \\ \min_{\mu, \sigma} \sum_{i=1}^n \log \left( \sqrt{2\pi\sigma} \right) + \frac{1}{2} \left( \frac{x_i - \mu}{\sigma} \right)^2 \end{aligned}$$

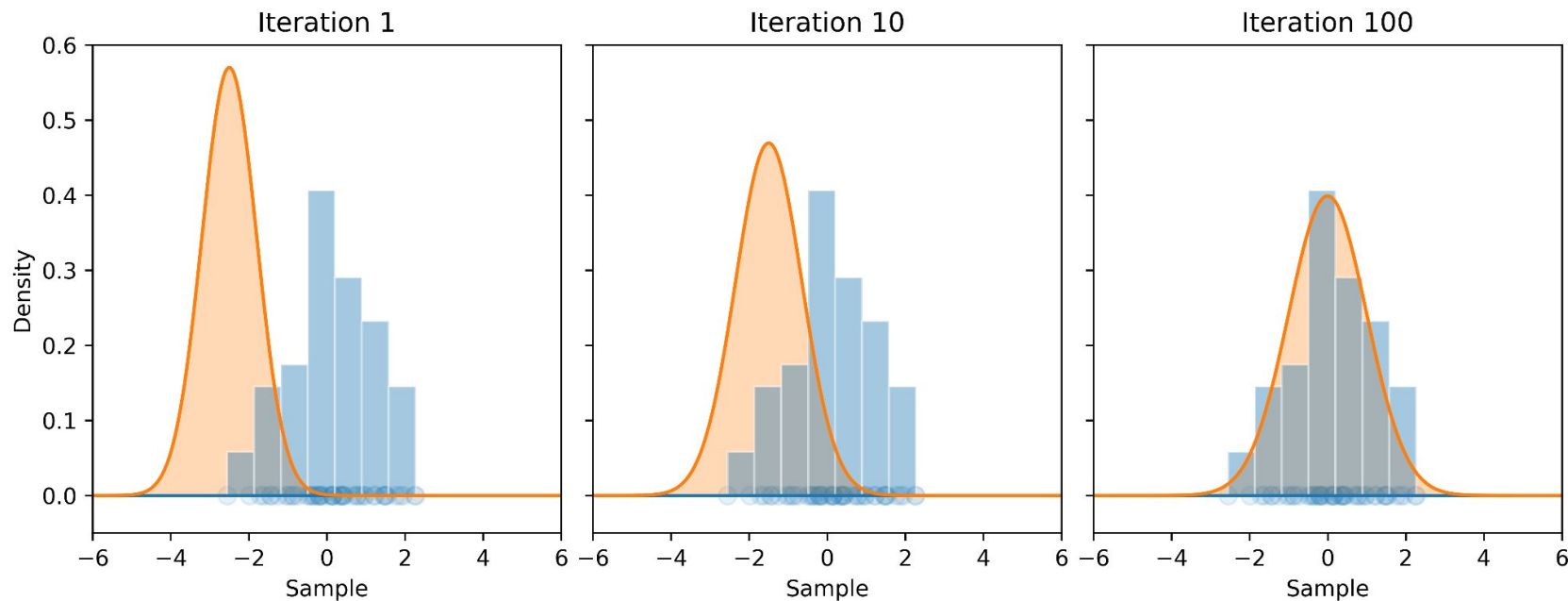
# Finding well-fitting distributions

- Now we can find the best parameters by gradient descent :)

$$\begin{aligned}(\mu, \sigma)_{t+1} &= (\mu, \sigma)_t - \eta \nabla_{(\mu, \sigma)} \left[ \sum_{i=1}^n \log \left( \sqrt{2\pi} \sigma_t \right) + \frac{1}{2} \left( \frac{x_i - \mu_t}{\sigma_t} \right)^2 \right] \\ &= (\mu, \sigma)_t - \eta \sum_{i=1}^n \nabla_{(\mu, \sigma)} \left[ \log \left( \sqrt{2\pi} \sigma_t \right) + \frac{1}{2} \left( \frac{x_i - \mu_t}{\sigma_t} \right)^2 \right]\end{aligned}$$

# Finding well-fitting distributions

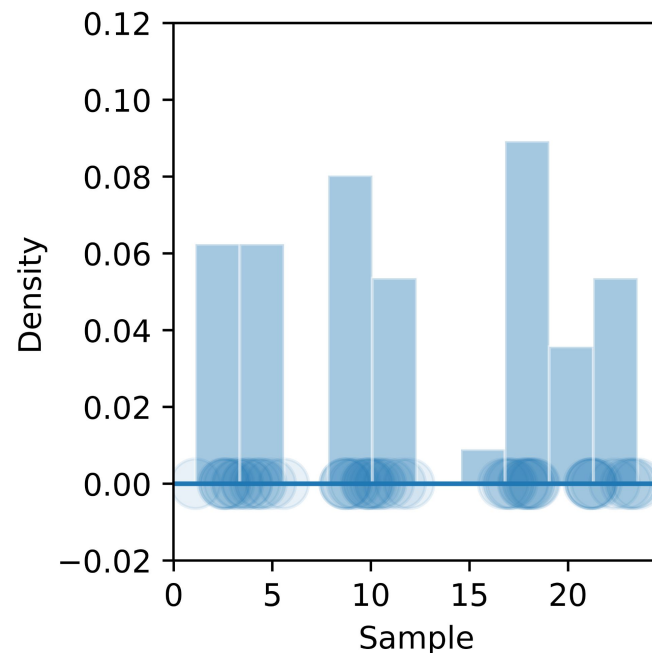
- By repeating the update steps, we arrive at a good fit :)





# Increasing the difficulty

- Finding the last distribution was easy. We could have even guessed the parameters.
- What about the next example?

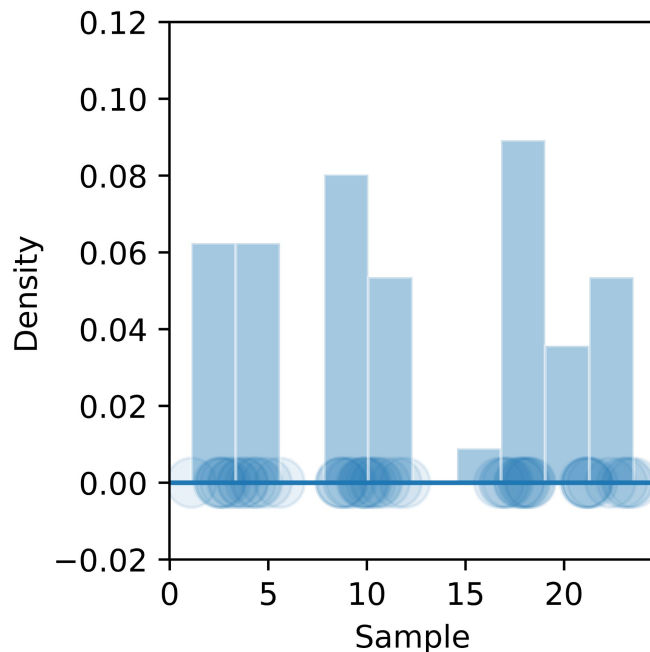


# Increasing the difficulty

- We could model each mode with a different Gaussian. Every sample comes from one of the modes.

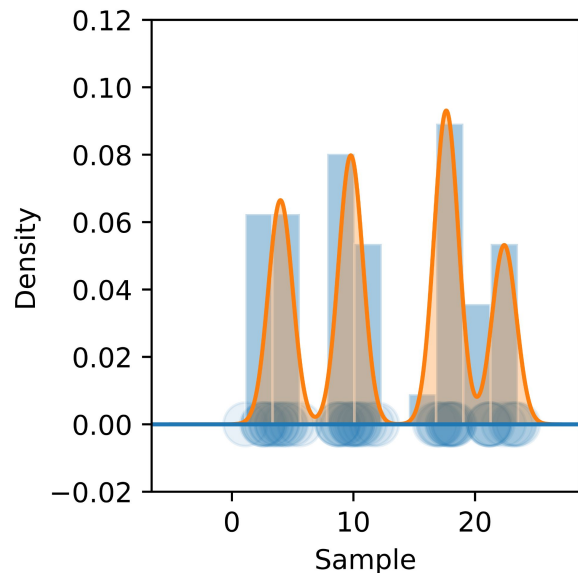
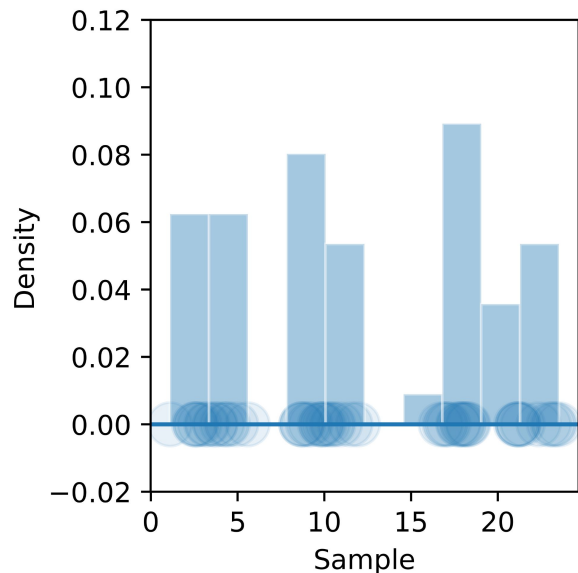
$$k \sim \text{Categorical}(w_1, w_2, w_3, w_4)$$

$$x|k \sim \mathcal{N}(\mu_k, \sigma_k)$$



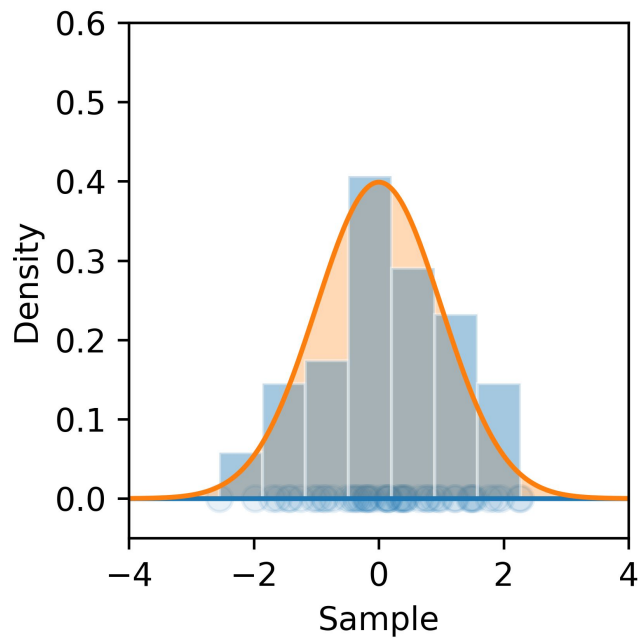
# Increasing the difficulty

- It works! But we needed to be clever. And optimization is not as stable.
- But we can sample and compute the density of new points :)



# Why do we even want to fit distributions?

- To **sample** from them
- To **compute the density** of new points
- Sampling = *generative modeling*
- Density computation: can see if a point is likely or not (*outlier detection*), can answer probabilistic questions as with a CDF



# What happens in the real world?

- Distributions can be **much more complicated** than 1D Gaussians
  - Images
  - Text
  - Audio
  - Point clouds
  - Bayesian posteriors
- We have **many more dimensions**
  - A 100x100x3 image has 30 thousand dimensions
  - A text or image embedding has  $O(1000)$  dimensions

# Shortcomings of specialized models

- Diffusion models make amazing images... but they **cannot evaluate density**.
- Highly specialized methods can detect tough outliers... but we **cannot generate new data** with them.
  
- What if we **need to do both**? Or what if we need to **find the exact distribution** that generated our data?

# Back to basics

- There is an elementary probabilistic theorem that states (simplified):

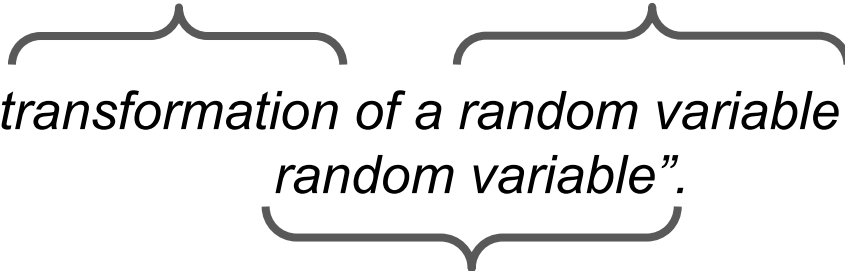
*“A transformation of a random variable is still a random variable”.*

- Remember: we can treat random variables as distributions.

# Back to basics

- Let's look at the theorem again.

Something complex      This can be simple



*“A transformation of a random variable is still a random variable”.*

Our complex data came from here

- Our data are just transformed samples of a simple distribution!



# Back to basics

- All generative models use such transformations.
- Why can't they compute the density?
- They need an **invertible transformation!**
- It lets us apply this formula:

$$\log p_X(x) = \log p_Z(f(x)) + \log |\det J_f(x)|$$

$f : \mathbb{R}^d \rightarrow \mathbb{R}^d$  is invertible

$J_f$  is the Jacobian matrix of  $f$

$Z$  is the base distribution with density  $p_Z$

$X$  is the target distribution with density  $p_X$

# Introducing normalizing flows

- A **normalizing flow** is a **distribution**.
- It is a **transformation** of a simple base distribution with an invertible map.
- The invertible map makes flows different from other generative models.

$$\log p_X(x) = \log p_Z(f(x)) + \log |\det J_f(x)|$$

$f : \mathbb{R}^d \rightarrow \mathbb{R}^d$  is invertible

$J_f$  is the Jacobian matrix of  $f$

$Z$  is the base distribution with density  $p_Z$

$X$  is the target distribution with density  $p_X$

# Introducing normalizing flows

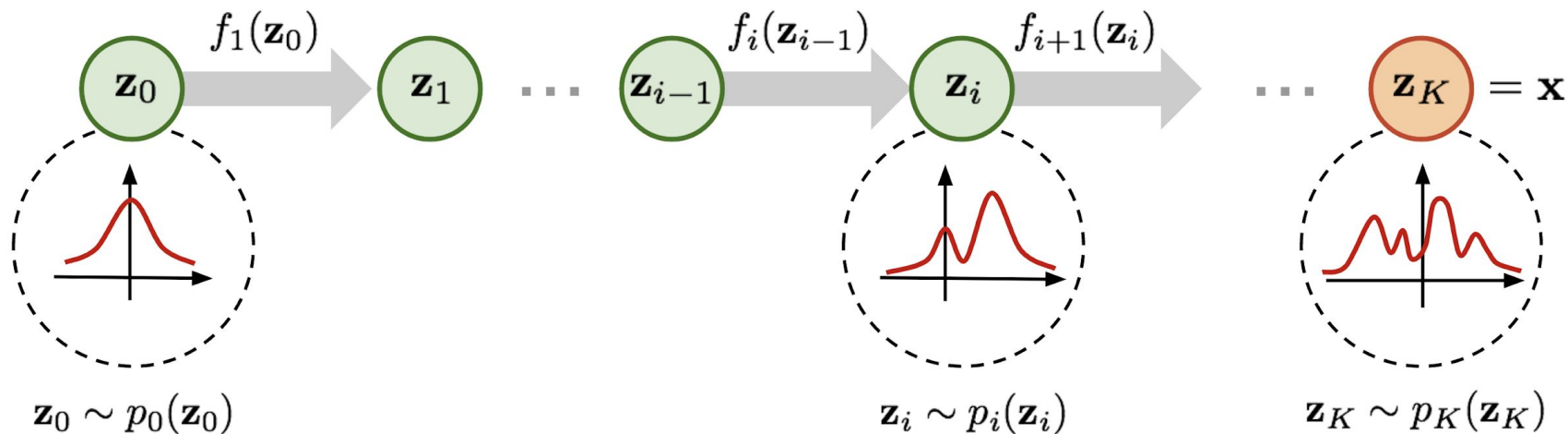
- To compute the log probability density:

$$\log p_X(x) = \log p_Z(f(x)) + \log |\det J_f(x)|$$

- To generate data: sample from the base distribution, transform the sample with the inverse map.

$$z \sim Z, x = f^{-1}(z) \Rightarrow x \sim X$$

# What does sampling look like?



# The small question

- What **base distribution** do we use?
- One that lets us compute the density easily
- One that we can sample from easily
- A **standard normal** fits the bill :)

... by the way, it needs to have the *same dimensionality* as our data.

# The BIG question

- How to define the **invertible map**?
- Need something **expressive**
- Need to compute the **Jacobian** easily
- Need to **invert** the map easily

# The BIG answer

- How to define the invertible map?
- Need something expressive ... use **neural networks**
- Need to compute the Jacobian easily ... **restrict** the architecture
- Need to invert the map easily ... **restrict** the architecture

# Composing invertible maps

- Idea: a **composition** of invertible maps is invertible.
- We will have a sequence of invertible layers.

$$f = f_1 \circ f_2 \circ \cdots \circ f_L$$

- We define one invertible layer, repeat it, and we're done :)



# What invertible functions do we know

- **Shift**: adding a number to an input is invertible.
- **Shift and scale**: scaling and then shifting is too.
- **Permutation**: shuffling elements of a vector is invertible, just have to remember the order.

$$f(x) = x + b$$

$$f(x) = ax + b \quad (a > 0)$$

$$f(x) = \text{permutation}$$

These are easy to invert and have an easy to compute Jacobian.

# Where do we plug in neural networks?

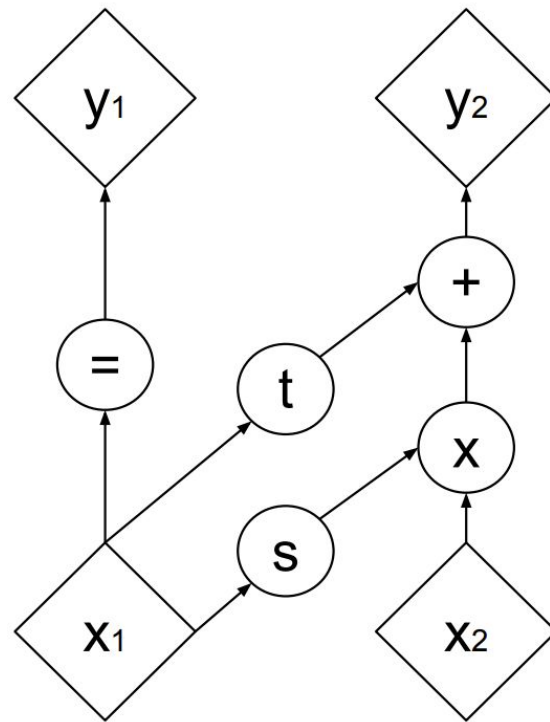
- We can make a flow by stacking shifts, scales, and permutations.
- BUT it won't be very expressive, as they are all **linear**.
- We want **nonlinear** invertible layers.
- If we are clever, we can make shift and scale “*nonlinear!*” 🤖  
How?

# Making nonlinear invertible maps

- Let  $x$  be our layer input
- Let  $y$  be our layer output

The genius idea:

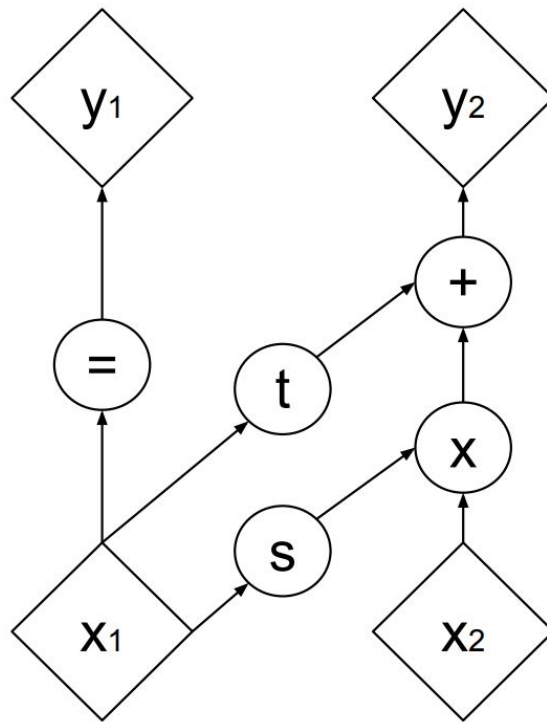
- Split  $x$  into two disjoint parts ( $x_1, x_2$ )
- Split  $y$  the same way
- Keep  $y_1 = x_1$
- Transform  $y_2 = s * x_2 + t$



# Making nonlinear invertible maps

The genius idea (cont'd):

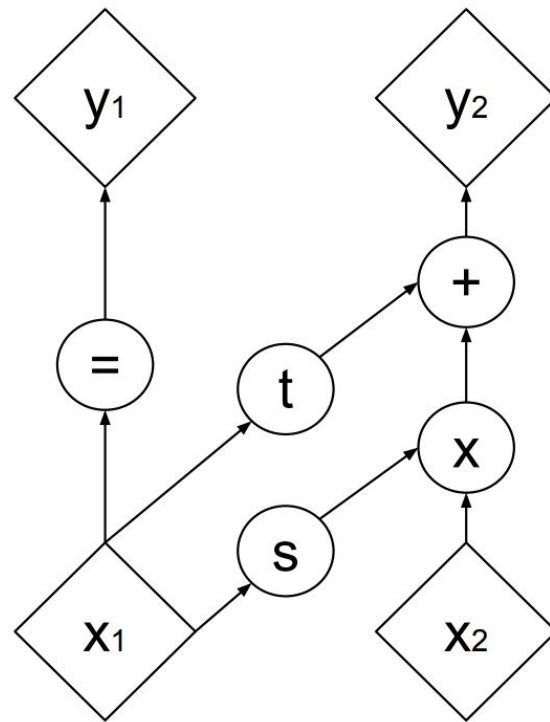
- Split  $x$  into two disjoint parts ( $x_1, x_2$ )
- Split  $y$  the same way
- Keep  $y_1 = x_1$
- Transform  $y_2 = s * x_2 + t$
- Use a neural network to predict  $(s, t)$  from  $x_1$



# Making nonlinear invertible maps

How to invert this?

- Receive  $y$  as input
- Split it into  $y_1, y_2$
- Keep  $x_1 = y_1$
- Take  $x_1$  and use it to predict  $(s, t)$
- $x_2 = (y_2 - t) / s$
- Concatenate  $x_1$  and  $x_2$  into  $x$  :)

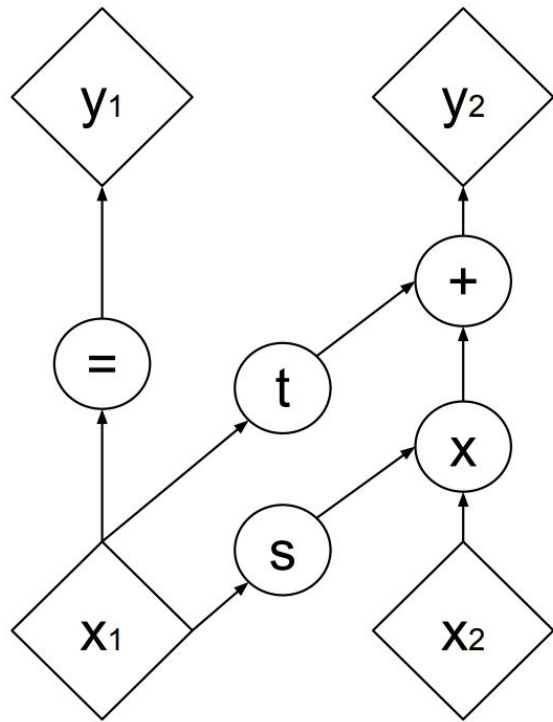


# Coupling flows

These maps are called coupling layers.

- They split a vector into two parts.
- They keep one part the same.
- They predict the parameters for the **transformer** with a **conditioner** neural network.
- They transform the other part with these parameters.

Stacking coupling layers makes a coupling flow!



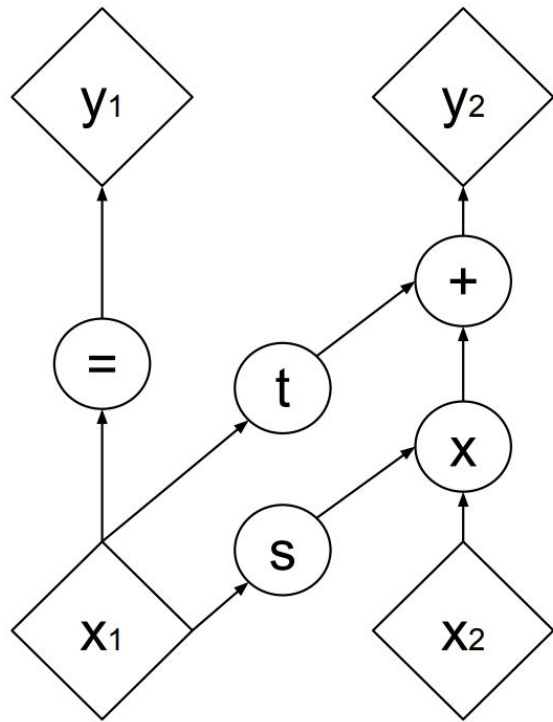
# Coupling flows

Here:

- transformer = *affine map*,
- conditioner = *feed-forward neural network*.

In general:

- transformer = any invertible map,
- conditioner = any function.



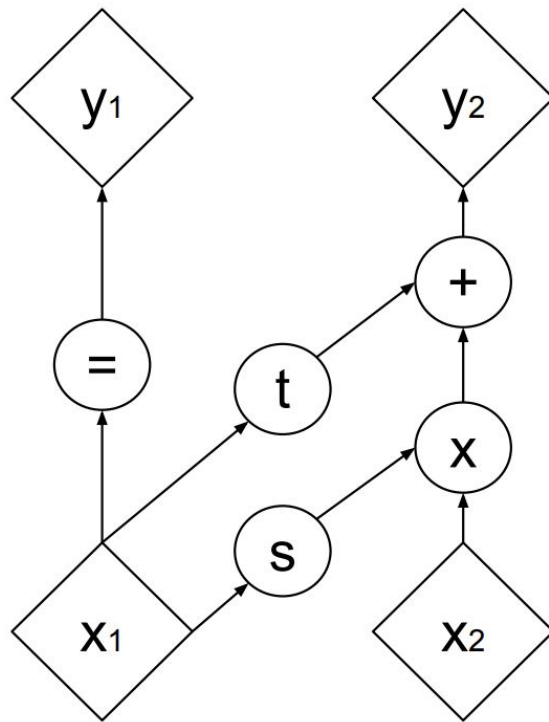
# Can we compute the log Jacobian determinant?

Yes.

Turns out the Jacobian is triangular.

$$\frac{\partial y}{\partial x^T} = \begin{bmatrix} \mathbb{I}_d & 0 \\ \frac{\partial y_{d+1:D}}{\partial x_{1:d}^T} & \text{diag}(\exp[s(x_{1:d})]) \end{bmatrix}$$

Log det = sum of the log of diagonal elements.





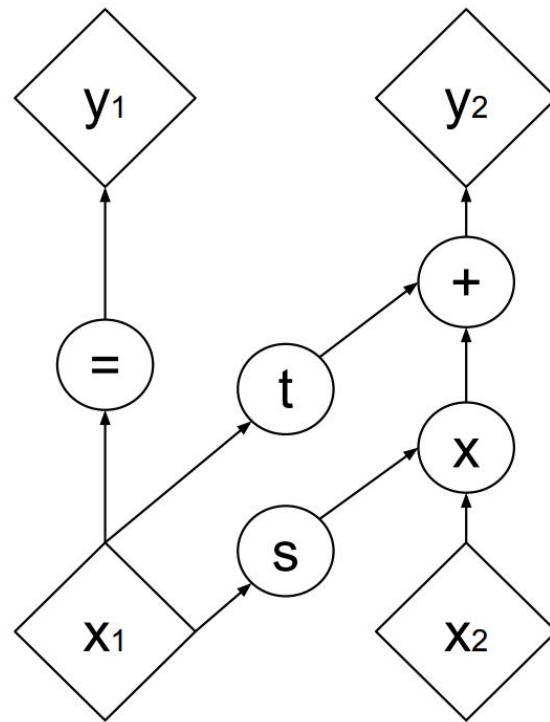
# Small detail

The input  $x_1$  will never be transformed.

But:

- permutations are invertible,
- we can place them between coupling blocks.

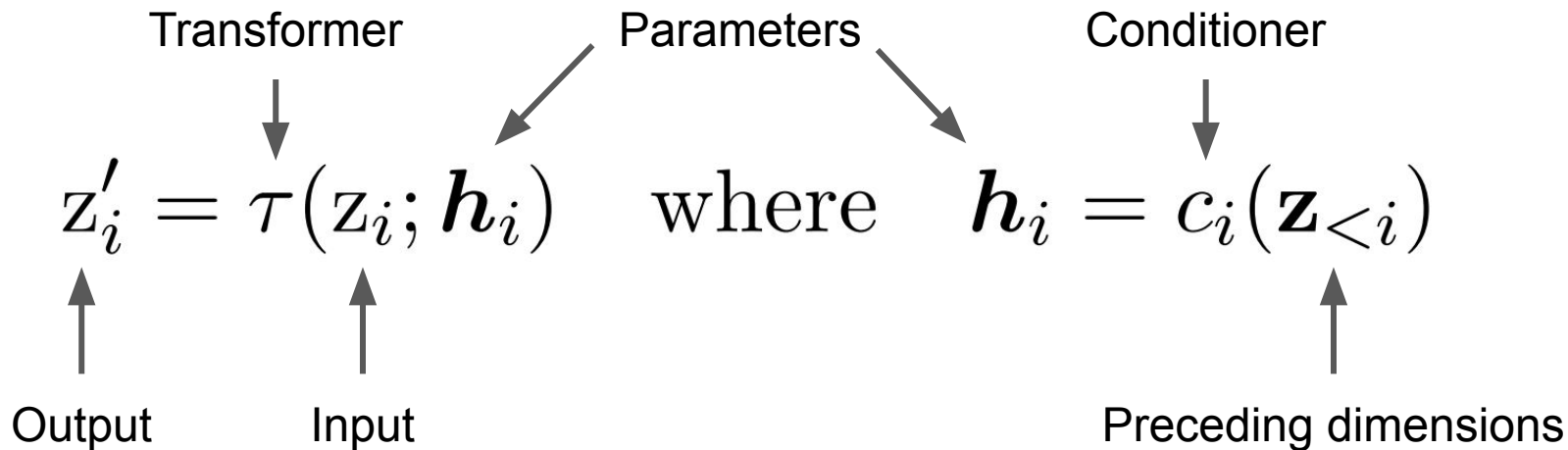
This will shuffle dimensions and ensure each is transformed.



# Autoregressive flows

We can generalize coupling layers to autoregressive layers.

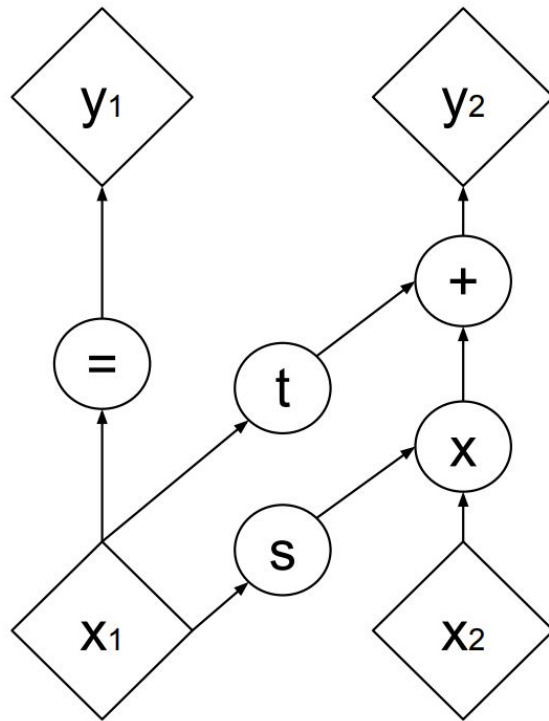
- Autoregressive layers transform inputs so that each output dimension is only affected by its preceding input dimensions.



# Autoregressive flows

Coupling layers are also autoregressive layers:

- Each output dimension of  $y_1$  is only affected by itself (valid).
- Each output dimension of  $y_2$  is only affected by  $x_1$  which precede them (valid).



# How to implement autoregressive flows

Theoretically, we could do it with coupling layers.

- Make many coupling layers
- Each layer's conditioner gets one more dimension input than the last

But this is very slow.

# How to implement autoregressive flows

Better strategy - a masked autoencoder:

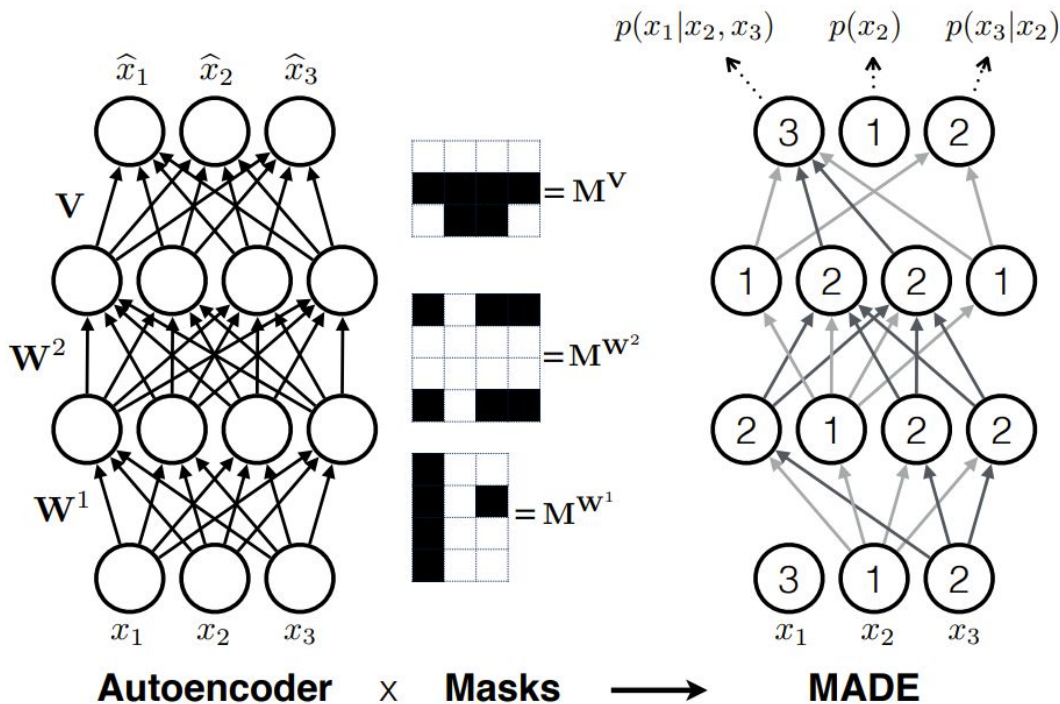
- takes the *entire vector*  $x$  as input,
- predicts as outputs the parameters for each dimension,

The detail:

- Certain weights become zero,
- this ensures each dimension is only affected by preceding ones,

# How to implement autoregressive flows

Using a masked autoencoder as a conditioner network results in a masked autoregressive layer.



## By the way, training autoregressive flows is easy

- We want to maximize the probability of samples.
- Parameters of the distribution = parameters of the conditioner network.
- We proceed by gradient descent (or Adam, Adagrad, ...).

$$\max_{\theta} P(D|\theta) = \min_{\theta} - \sum_{i=1}^n \log p_Z(f_{\theta}(x_i)) + \log |\det J_{f_{\theta}}(x_i)|$$

# So what are these architectures called?

- **NICE**: shift transformer, coupling conditioner
- **Real NVP**: affine transformer, coupling conditioner
- **MAF**: affine transformer, masked autoregressive conditioner
- **IAF**: affine transformer, masked autoregressive conditioner (reverse direction)
- **LRS-NSF**: linear rational spline transformer, either conditioner
- **RQ-NSF**: rational quadratic spline transformer, either conditioner
- **SINF**: rational quadratic spline transformer, coupling conditioner (also includes some orthogonal transforms in between)
- **UMNN-MAF**: monotonic neural network transformer, either conditioner
- ... some others as well



# Can we see some examples?

You can use Glow to make images. Example: project two images into the space of the base distribution, then draw a line in this base space. Points on the line interpolate between images in original space.

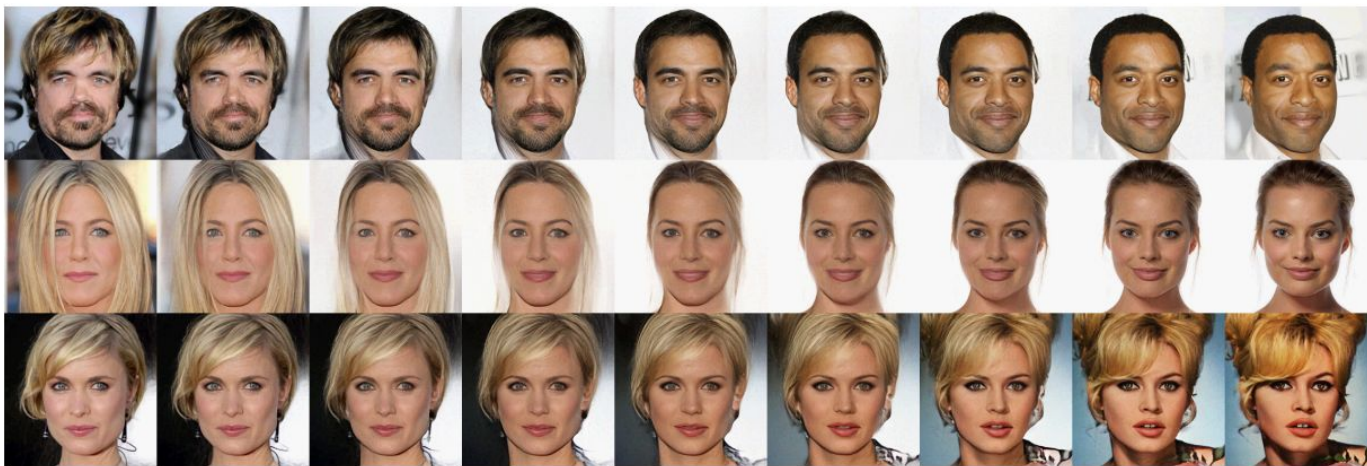
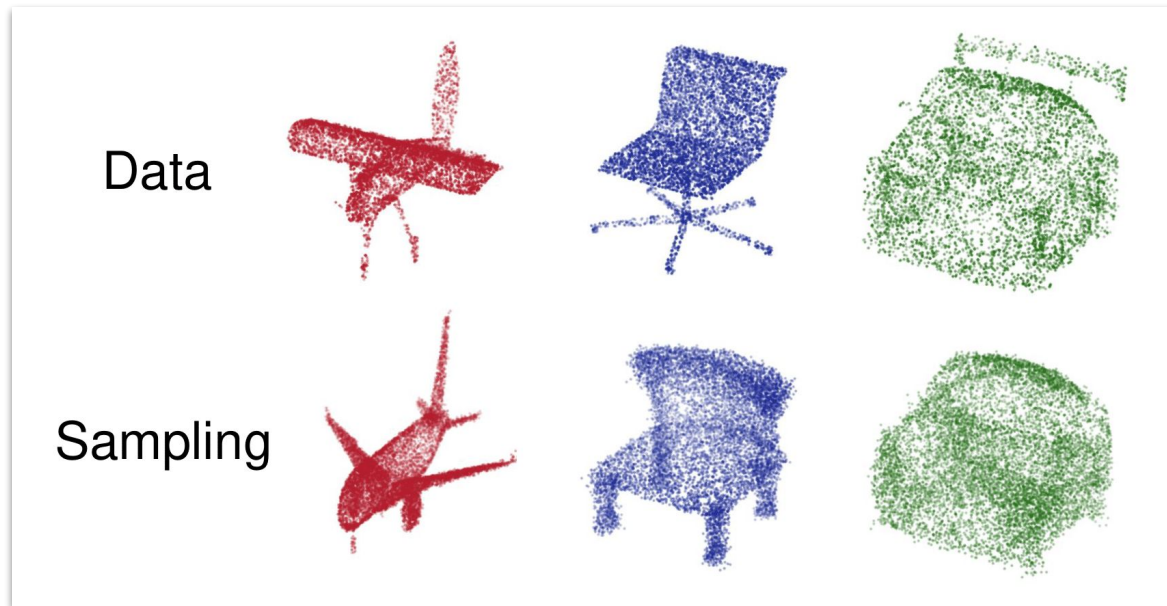


Figure 5: Linear interpolation in latent space between real images

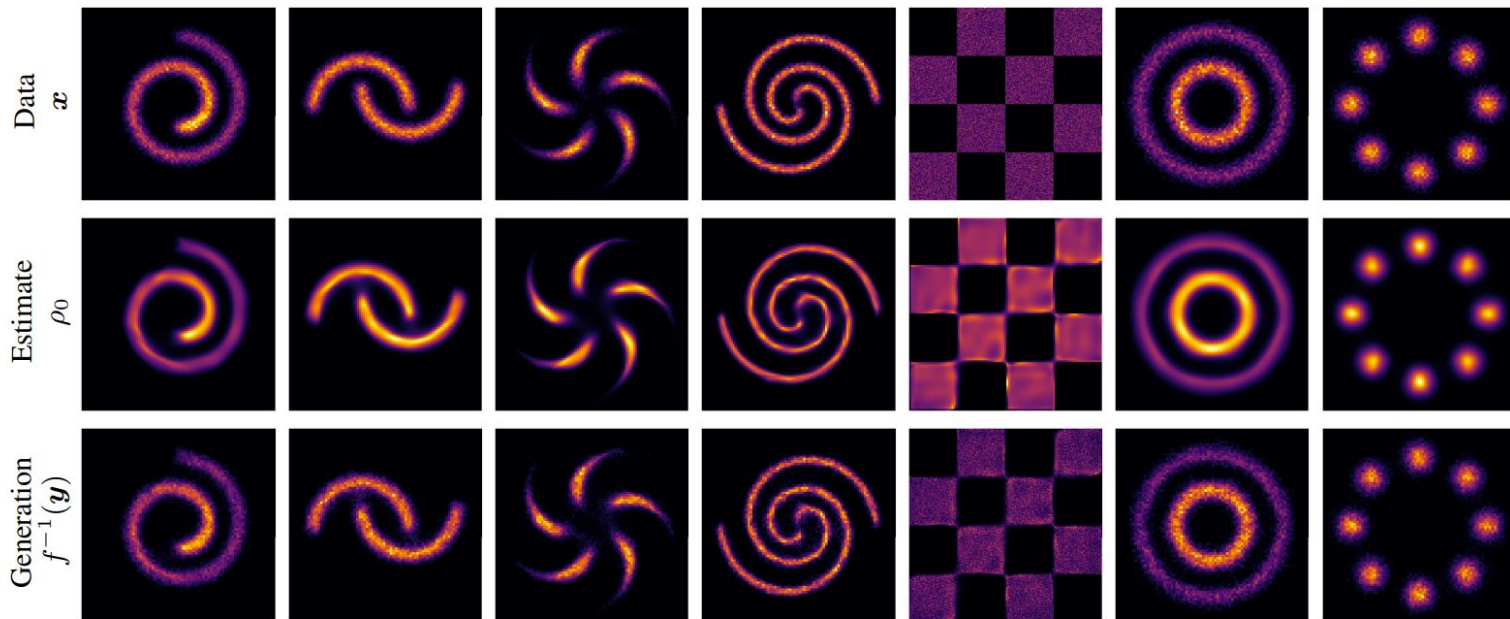
# Can we see some examples?

You can use SoftFlow to generate point clouds. Train it on point clouds of planes, chairs, armchairs; then sample new kinds of these objects.



# Can we see some examples?

You can model a variety of complex distributions with all normalizing flows.

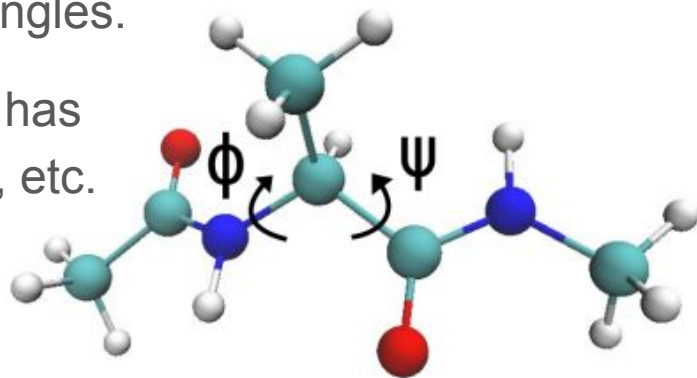


# Can we see some examples?

Normalizing flows are used in sampling from complex Bayesian posteriors (e.g. molecular dynamics, cosmology, quantum chromodynamics).

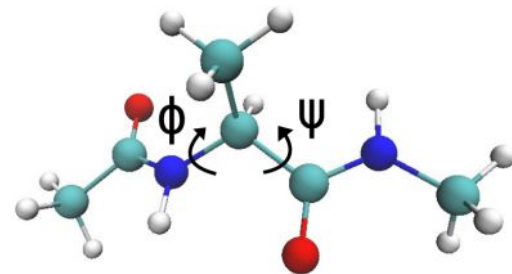
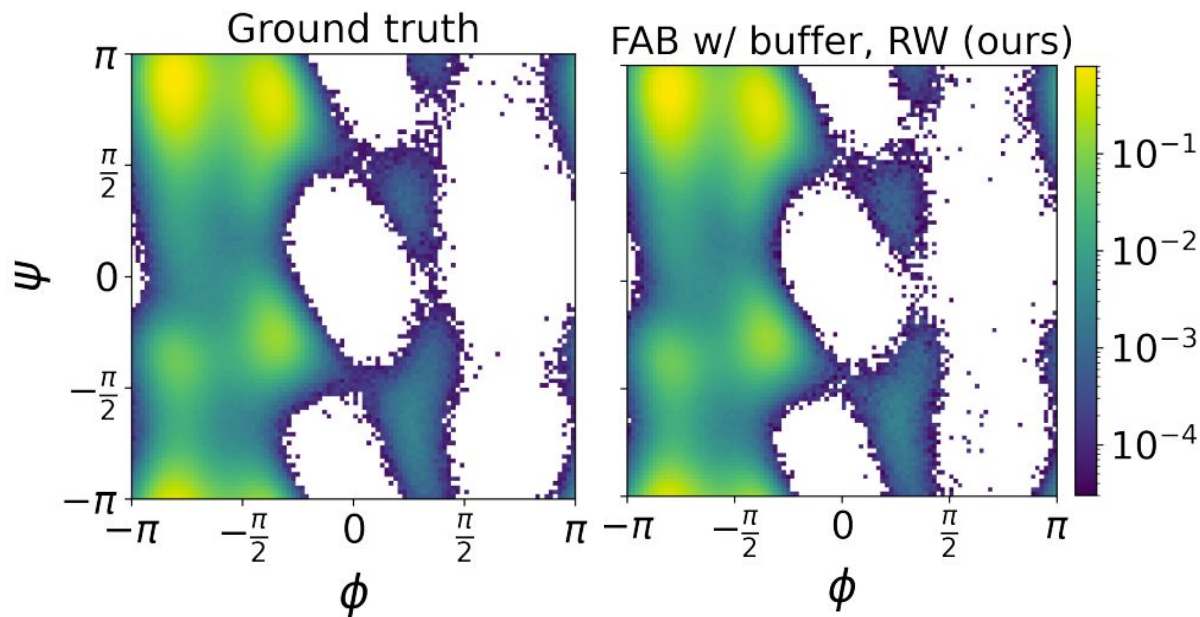
Example: we take a molecule of Alanine Dipeptide and dissolve it in a solvent. We want to model the distribution of dihedral angles (phi and psi). We use a simulation, so we never actually have reference angles.

Reliably obtaining the distribution of these angles has big implications for protein folding, ligand docking, etc.



# Can we see some examples?

By combining a Markov Chain Monte Carlo method with a normalizing flow, we obtain the final distribution of these angles.



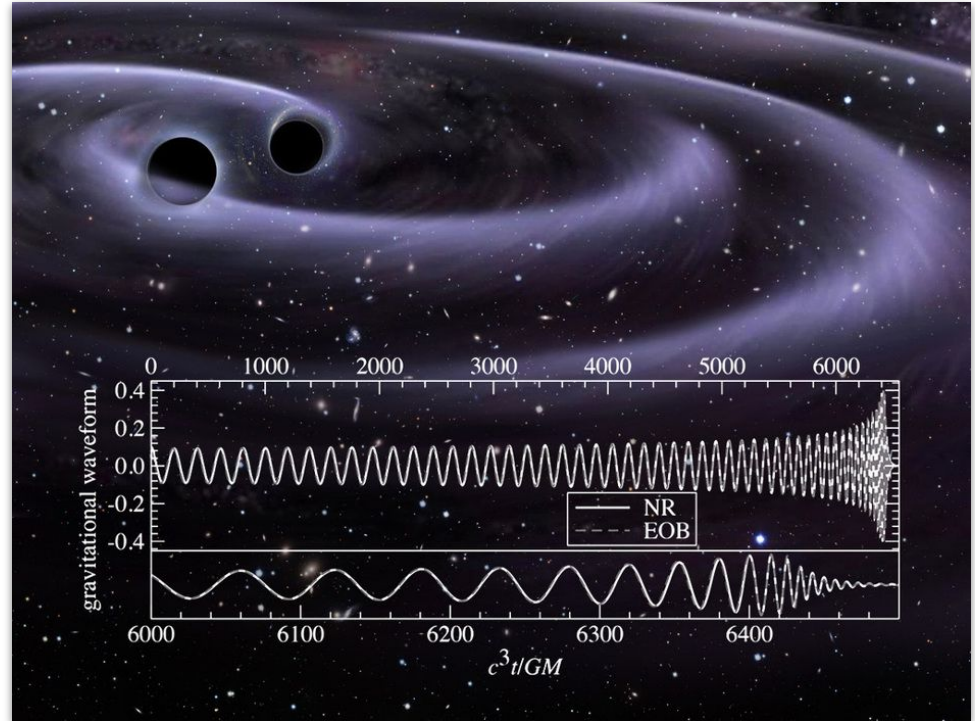


# Can we see some examples?

An example from cosmology is modeling orbital binary systems using gravitational wave measurements.

When two massive objects (e.g. black holes or neutron stars) rotate in a binary system and begin merging, they send out ripples in space-time, called gravitational waves.

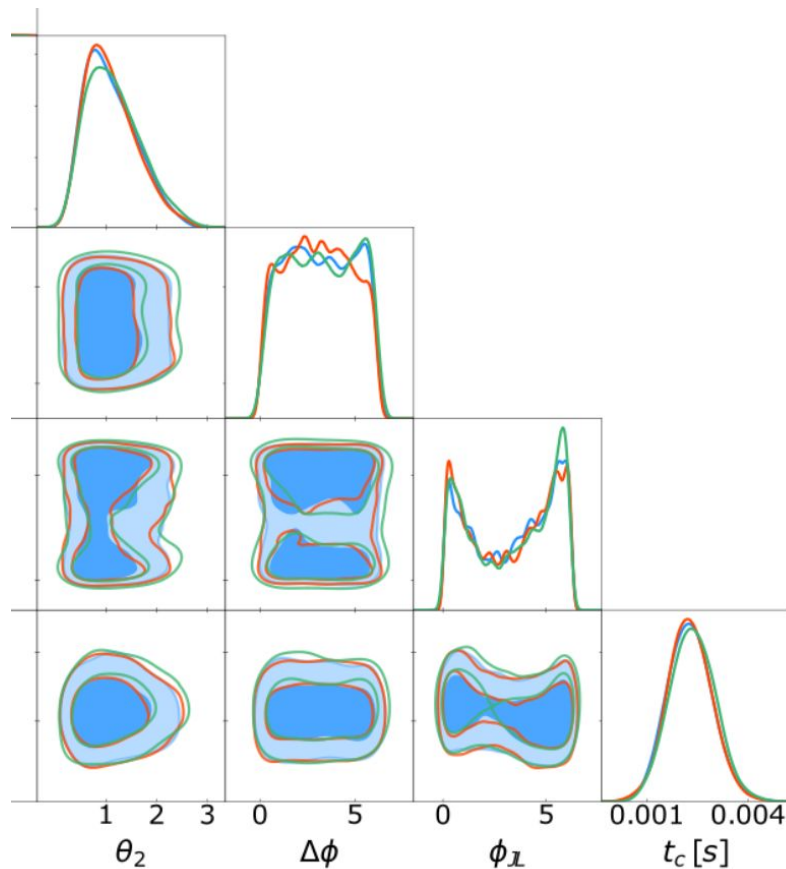
We can measure gravitational waves using specialized observatories.



# Can we see some examples?

For a system of two black holes, we can use gravitational wave measurements to infer black hole parameters (e.g. mass, spin, etc.).

Normalizing flows can be used within Markov chain Monte Carlo to obtain the distributions of these parameters.



# How can we try this out ourselves?

Packages:

- [github.com/davidnabergoj/normalizing-flows](https://github.com/davidnabergoj/normalizing-flows) (releasing soon, general)
- [github.com/VincentStimper/normalizing-flows](https://github.com/VincentStimper/normalizing-flows) (general)
- [github.com/bayesiains/nflows](https://github.com/bayesiains/nflows) (general)
- Tensorflow probability: <https://www.tensorflow.org/probability> (general)
- Distrax: <https://github.com/google-deepmind/distrax> (general)
- pocoMC: [github.com/minaskar/pocomc](https://github.com/minaskar/pocomc) (flows in cosmology)
- [github.com/janosh/awesome-normalizing-flows](https://github.com/janosh/awesome-normalizing-flows) (overview)



# What does the code look like?

Could not be simpler.

```
bijection = RealNVP(n_dim) # Create the bijection
flow = Flow(bijection) # Create the normalizing flow

flow.fit(x) # Fit the normalizing flow to training data
log_prob = flow.log_prob(x) # Compute the log probability of training data
x_new = flow.sample(50) # Sample 50 new data points
```

## Where can I read more?

- **Good review:** Papamakarios et al. *Normalizing Flows for Probabilistic Modeling and Inference*. 2021.
- **Our coupling flow:** Dinh et al. *Density estimation using Real NVP*. 2017.
- **Our masked AR flow:** Papamakarios et al. *Masked Autoregressive Flow for Density Estimation*. 2018.