# Generative Models (Exact)

DL2 – 2024

Wilker Aziz
w.aziz@uva.nl

University of Amsterdam
Institute for Logic, Language and Computation

# Module Overview

1. Representing uncertainty in ML
   uncertainty — probabilistic models — tools for prescribing distributions
2. Generative models (exact)
   autoregressive models — normalising flows
3. Generative models (approximate)
   energy-based models — score-matching and diffusion
4. Latent variable models
   exact inference — variational inference

How to use this: during the class, don't go on reading everything you see on this side. I will walk you through what's needed. I hope these notes will help you when you refer back to the content in your own time.

# Outline

# Prescribing distributions

We will now discuss various ways to prescribe distributions using deep learning. For each technique, we will keep an eye on two things:

- our ability to assess the probability mass/density of a given outcome
- our ability to sample outcomes from the corresponding distribution

We begin with the univariate case and then discuss the multivariate case.

- assessment: useful for learning (e.g., via approximate MLE).
- sampling: useful for making predictions.

# Univariate case

- enumeration
- known parametric form
- transform a known random source
- data augmentation and marginalisation

# Outline

# Multivariate case

For the fixed-dimensional case, we may have access to multivariate generalisations of known pdfs (e.g., MVN).

In general (fixed-dimension or not), we can exploit a *factorisation* with or without conditional independence assumptions.

Then, we predict the factors:

- direct, or cdf, or sampler/simulator
- unnormalised

# Directed

Decompose an outcome into parts $y = (w_1, \ldots, w_N)$, for example, a sentence is a sequence of tokens, an image is a sequence of pixels. We fix an order (e.g., left-to-right, row-wise or column-wise, etc).

Factorise $p_{Y|X}(y|x)$ using univariate conditionals.

Examples of factorisation

1. full conditional independence $p_{Y|X}(y|x) \overset{\text{ind.}}{=} \prod_{n=1}^{N} p_{W|X}(w_n|x)$
2. Markov model $p_{Y|X}(y|x) \overset{\text{ind.}}{=} \prod_{n=1}^{N} p_{W|XH}(w_n|x, w_{n-k+1:n-1})$
3. chain rule $p_{Y|X}(y|x) = \prod_{n=1}^{N} p_{W|XH}(w_n|x, w_{<n})$
   aka *autoregressive factorisation*

Given a flexible-enough family for the conditionals, (3) can identify *any* probability measure, in principle.

See Germain et al. (MADE; 2015a) and any decoder-only or encoder-decoder model (Mikolov et al., 2010; Van den Oord et al., 2016; Oord et al., 2016; Vaswani et al., 2017)

# Undirected

It's possible to factorise an unnormalised version of $p_{Y|X}(y|x)$ using unnormalised factors.

For example, a first-order conditional random field $p_{Y|X}(y|x) \propto \prod_{n=1}^{N} \Phi(x, n, w_{n-1}, w_n)$ uses factors like $\Phi(x, n, w_{n-1}, w_n) > 0$.

The familiar constraints apply: non-negativity, finite normalisation constant.

Density/mass assessment, sampling may be possible in some cases (eg, first-order CRF) but are intractable in general.



In the linear-chain CRF, the normalisation constant for a given $x$

$$\mathcal{Z}(x) = \sum_{w_1=1}^{V} \cdots \sum_{w_N=1}^{V} \prod_{n=1}^{N} \Phi(x, n, w_{n-1}, w_n) \tag{1}$$

can be computed via $\mathcal{Z}(x) = \alpha(N+1, \mathrm{id}(\mathrm{EoS}))$ where

$$\alpha(n, c) = \begin{cases} \Phi(x, 1, \mathrm{id}(\mathrm{BoS}), c) & \text{if } n = 1 \\ \sum_{r=1}^{V} \alpha(n-1, r) \Phi(x, n, r, c) & \text{if } n > 1 \end{cases} \tag{2}$$

($1 \leq n \leq N+1$ is a position, and $c \in [V]$ is an outcome of $W_n$)

Some modern presentations of this recursion (the forward algorithm) can be found in (Eisner, 2016; Smith, 2011). Vlad Niculae's course is excellent: https://vene.ro/mlsd/.

EBMs: with rather flexible NNs, we can parameterise an unnormalised model **without a factorisation**: $p_{Y|X}(y|x, \theta) \propto \mathrm{NN}(x, y; \theta)$. Efficient computation of $\mathcal{Z}(x)$ is not possible.

# Honourable mentions

- Sparse continuous distributions (Martins et al., 2022)
- Score matching (implicit generative models) (Vincent, 2011; Song and Ermon, 2019; Song et al., 2020)
- Diffusion processes (Sohl-Dickstein et al., 2015; Kingma et al., 2021)

We will cover score matching and diffusion in this module.

# Summary

There are various ways to prescribe distributions both univariate and multivariate.

- Predict parameters for known pdfs and cdfs: we predict some finite (typically small) number of parameters, and evaluate the mass/density of an outcome using a known function.
- For more flexibility we construct novel pdfs or cdfs by predicting unnormalised densities or parameterising flows and simulators.
- For multivariate and structured data we typically exploit a factorisation into simpler distributions (NNs are particularly good at representing complex conditioning contexts).

There are various tradeoffs: is mass/density assessment tractable? can we sample? do we need backward passes? do we need to approximate normalisation constants?

# Outline

# Maximum likelihood estimation

We have a probability model of a random variable $Y$, and this model may condition on available covariates $X$. This model has parameters $\theta$ and assigns probability mass/density $p(y|x, \theta)$ to an observation.

I may omit the subscripts from the pdfs whenever I find it unambiguous. That is, I write $p(y|x, \theta)$ instead of $p_{Y|X}(y|x, \theta)$.

# Maximum likelihood estimation

We have a probability model of a random variable $Y$, and this model may condition on available covariates $X$. This model has parameters $\theta$ and assigns probability mass/density $p(y|x, \theta)$ to an observation.

Given a dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(N)}, y^{(N)})\}$ of i.i.d. observations,

I may omit the subscripts from the pdfs whenever I find it unambiguous. That is, I write $p(y|x, \theta)$ instead of $p_{Y|X}(y|x, \theta)$.

# Maximum likelihood estimation

We have a probability model of a random variable $Y$, and this model may condition on available covariates $X$. This model has parameters $\theta$ and assigns probability mass/density $p(y|x, \theta)$ to an observation.

Given a dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(N)}, y^{(N)})\}$ of i.i.d. observations, the log-likelihood function gives us a criterion for parameter estimation

$$\mathcal{L}_{\mathcal{D}}(\theta) =$$

I may omit the subscripts from the pdfs whenever I find it unambiguous. That is, I write $p(y|x, \theta)$ instead of $p_{Y|X}(y|x, \theta)$.

# Maximum likelihood estimation

We have a probability model of a random variable $Y$, and this model may condition on available covariates $X$. This model has parameters $\theta$ and assigns probability mass/density $p(y|x, \theta)$ to an observation.

Given a dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(N)}, y^{(N)})\}$ of i.i.d. observations, the log-likelihood function gives us a criterion for parameter estimation

$$\mathcal{L}_{\mathcal{D}}(\theta) = \log \prod_{s=1}^{N} p(y^{(s)}|x^{(s)}, \theta) =$$

I may omit the subscripts from the pdfs whenever I find it unambiguous. That is, I write $p(y|x, \theta)$ instead of $p_{Y|X}(y|x, \theta)$.

# Maximum likelihood estimation

We have a probability model of a random variable $Y$, and this model may condition on available covariates $X$. This model has parameters $\theta$ and assigns probability mass/density $p(y|x,\theta)$ to an observation.

Given a dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(N)}, y^{(N)})\}$ of i.i.d. observations, the log-likelihood function gives us a criterion for parameter estimation

$$\mathcal{L}_{\mathcal{D}}(\theta) = \log \prod_{s=1}^{N} p(y^{(s)}|x^{(s)},\theta) = \sum_{s=1}^{N} \log p(y^{(s)}|x^{(s)},\theta)$$

I may omit the subscripts from the pdfs whenever I find it unambiguous. That is, I write $p(y|x,\theta)$ instead of $p_{Y|X}(y|x,\theta)$.

# MLE via gradient-based optimisation

If the log-likelihood is **differentiable** and **tractable** then backpropagation gives us the gradient

$$\nabla_\theta \mathcal{L}_\mathcal{D}(\theta) =$$

**Differentiable**

Consider the example of a Categorical likelihood:

- for a data point $(x, y)$ the log-likelihood is $\log \text{Cat}(y|f(x;\theta)) = \log f_y(x;\theta)$
  This shows that the Categorical likelihood $\text{Cat}(y|f(x;\theta))$ is differentiable with respect to its parameter $f_y(x;\theta)$.
- To satisfy differentiability with respect to $\theta$ for any $(x, y)$, we need $f(\cdot;\theta)$, to be differentiable with respect to $\theta$ in its domain (the space $\mathcal{X}$ of all covariates).

**Tractable** The evaluation of $f(x;\theta)$ is tractable for any $x \in \mathcal{X}$.

**Beyond** Think about other likelihoods (e.g., Bernoulli, Binomial, Multinomial, Poisson, Geometric, Gaussian, Exponential, Gamma), can you imagine differentiable and tractable parameterisations of the model?

# MLE via gradient-based optimisation

If the log-likelihood is **differentiable** and **tractable** then backpropagation gives us the gradient

$$\nabla_\theta \mathcal{L}_\mathcal{D}(\theta) = \nabla_\theta \sum_{s=1}^{N} \log p(y^{(s)}|x^{(s)}, \theta) \quad =$$

**Differentiable**

Consider the example of a Categorical likelihood:

- for a data point $(x, y)$ the log-likelihood is $\log \text{Cat}(y|f(x;\theta)) = \log f_y(x;\theta)$
  This shows that the Categorical likelihood $\text{Cat}(y|f(x;\theta))$ is differentiable with respect to its parameter $f_y(x;\theta)$.
- To satisfy differentiability with respect to $\theta$ for any $(x, y)$, we need $f(\cdot;\theta)$, to be differentiable with respect to $\theta$ in its domain (the space $\mathcal{X}$ of all covariates).

**Tractable** The evaluation of $f(x;\theta)$ is tractable for any $x \in \mathcal{X}$.

**Beyond** Think about other likelihoods (e.g., Bernoulli, Binomial, Multinomial, Poisson, Geometric, Gaussian, Exponential, Gamma), can you imagine differentiable and tractable parameterisations of the model?

# MLE via gradient-based optimisation

If the log-likelihood is **differentiable** and **tractable** then backpropagation gives us the gradient

$$\nabla_\theta \mathcal{L}_\mathcal{D}(\theta) = \nabla_\theta \sum_{s=1}^{N} \log p(y^{(s)}|x^{(s)}, \theta) \quad = \sum_{s=1}^{N} \nabla_\theta \log p(y^{(s)}|x^{(s)}, \theta)$$

**Differentiable**

Consider the example of a Categorical likelihood:

- for a data point $(x, y)$ the log-likelihood is $\log \mathrm{Cat}(y|f(x;\theta)) = \log f_y(x;\theta)$
  This shows that the Categorical likelihood $\mathrm{Cat}(y|f(x;\theta))$ is differentiable with respect to its parameter $f_y(x;\theta)$.
- To satisfy differentiability with respect to $\theta$ for any $(x, y)$, we need $f(\cdot;\theta)$, to be differentiable with respect to $\theta$ in its domain (the space $\mathcal{X}$ of all covariates).

**Tractable** The evaluation of $f(x;\theta)$ is tractable for any $x \in \mathcal{X}$.

**Beyond** Think about other likelihoods (e.g., Bernoulli, Binomial, Multinomial, Poisson, Geometric, Gaussian, Exponential, Gamma), can you imagine differentiable and tractable parameterisations of the model?

# MLE via gradient-based optimisation

If the log-likelihood is **differentiable** and **tractable** then backpropagation gives us the gradient

$$\nabla_\theta \mathcal{L}_\mathcal{D}(\theta) = \nabla_\theta \sum_{s=1}^{N} \log p(y^{(s)}|x^{(s)}, \theta) \quad = \sum_{s=1}^{N} \nabla_\theta \log p(y^{(s)}|x^{(s)}, \theta)$$

and we can update $\theta$ in the direction

$$\gamma \nabla_\theta \mathcal{L}_\mathcal{D}(\theta)$$

to attain a local maximum of the likelihood function

**Differentiable**

Consider the example of a Categorical likelihood:

- for a data point $(x, y)$ the log-likelihood is $\log \mathrm{Cat}(y|f(x;\theta)) = \log f_y(x;\theta)$
  This shows that the Categorical likelihood $\mathrm{Cat}(y|f(x;\theta))$ is differentiable with respect to its parameter $f_y(x;\theta)$.
- To satisfy differentiability with respect to $\theta$ for any $(x, y)$, we need $f(\cdot;\theta)$, to be differentiable with respect to $\theta$ in its domain (the space $\mathcal{X}$ of all covariates).

**Tractable** The evaluation of $f(x;\theta)$ is tractable for any $x \in \mathcal{X}$.

**Beyond** Think about other likelihoods (e.g., Bernoulli, Binomial, Multinomial, Poisson, Geometric, Gaussian, Exponential, Gamma), can you imagine differentiable and tractable parameterisations of the model?

# Big Data

For large $N$, computing the gradient is inconvenient

$$\nabla_\theta \mathcal{L}_\mathcal{D}(\theta) = \underbrace{\sum_{s=1}^{N} \nabla_\theta \log p(y^{(s)}|x^{(s)}, \theta)}_{\text{too many terms}}$$

We are looking for a principled way to approximate the exact gradient. Being principled here means enjoying some guarantees (this usually requires satisfying certain properties, as we shall see).

Note that we introduced the notion of a *stochastic gradient*, a random variable whose range is the space of gradient vectors of our model's log-likelihood function.

We have expressed the exact gradient as the expected value of that random variable. Can you see how we are going to estimate it with a computation that does not depend on $N$?

# Big Data

For large $N$, computing the gradient is inconvenient

$$\nabla_\theta \mathcal{L}_\mathcal{D}(\theta) = \underbrace{\sum_{s=1}^{N} \nabla_\theta \log p(y^{(s)}|x^{(s)}, \theta)}_{\text{too many terms}}$$

$$= \sum_{s=1}^{N} \frac{1}{N} N \nabla_\theta \log p(y^{(s)}|x^{(s)}, \theta)$$

We are looking for a principled way to approximate the exact gradient. Being principled here means enjoying some guarantees (this usually requires satisfying certain properties, as we shall see).

Note that we introduced the notion of a *stochastic gradient*, a random variable whose range is the space of gradient vectors of our model's log-likelihood function.

We have expressed the exact gradient as the expected value of that random variable. Can you see how we are going to estimate it with a computation that does not depend on $N$?

# Big Data

For large $N$, computing the gradient is inconvenient

$$\nabla_\theta \mathcal{L}_\mathcal{D}(\theta) = \underbrace{\sum_{s=1}^{N} \nabla_\theta \log p(y^{(s)}|x^{(s)}, \theta)}_{\text{too many terms}}$$

$$= \sum_{s=1}^{N} \frac{1}{N} N \nabla_\theta \log p(y^{(s)}|x^{(s)}, \theta)$$

$$= \sum_{s=1}^{N} \mathcal{U}(s|1/N) N \nabla_\theta \log p(y^{(s)}|x^{(s)}, \theta)$$

We are looking for a principled way to approximate the exact gradient. Being principled here means enjoying some guarantees (this usually requires satisfying certain properties, as we shall see).

Note that we introduced the notion of a *stochastic gradient*, a random variable whose range is the space of gradient vectors of our model's log-likelihood function.

We have expressed the exact gradient as the expected value of that random variable. Can you see how we are going to estimate it with a computation that does not depend on $N$?

# Big Data

For large $N$, computing the gradient is inconvenient

$$\begin{aligned}\nabla_\theta \mathcal{L}_{\mathcal{D}}(\theta) &= \underbrace{\sum_{s=1}^{N} \nabla_\theta \log p(y^{(s)}|x^{(s)},\theta)}_{\text{too many terms}} \\ &= \sum_{s=1}^{N} \frac{1}{N} N \nabla_\theta \log p(y^{(s)}|x^{(s)},\theta) \\ &= \sum_{s=1}^{N} \mathcal{U}(s|1/N) N \nabla_\theta \log p(y^{(s)}|x^{(s)},\theta) \\ &= \mathbb{E}_{S\sim\mathcal{U}(1/N)}\left[N \nabla_\theta \log p(y^{(S)}|x^{(S)},\theta)\right]\end{aligned}$$

$S$ selects data points uniformly at random

We are looking for a principled way to approximate the exact gradient. Being principled here means enjoying some guarantees (this usually requires satisfying certain properties, as we shall see).

Note that we introduced the notion of a *stochastic gradient*, a random variable whose range is the space of gradient vectors of our model's log-likelihood function.

We have expressed the exact gradient as the expected value of that random variable. Can you see how we are going to estimate it with a computation that does not depend on $N$?

# Stochastic optimisation

For large $N$, we can use a gradient estimate

$$\nabla_\theta \mathcal{L}_\mathcal{D}(\theta) = \underbrace{\mathbb{E}_{S\sim\mathcal{U}(1/N)}\left[N\nabla_\theta \log p(y^{(S)}|x^{(S)},\theta)\right]}_{\text{expected gradient :)}}$$

The theory of stochastic optimisation (Robbins and Monro, 1951) tells us that we will converge to a local optimum of the objective as long as we take steps that are correct *on average*. This means we can optimise with stochastic gradient estimates, for as long as they are unbiased estimates of the exact gradient.

Do you see the guarantee and the condition?

There are more conditions, however. The learning rate must comply with some key properties. Luckily many learning rate schedules have been documented in the literature, and most our famous optimisers meet the Robbis and Monro conditions (though not all).

If you want to read more, but need something more accessible than the 1951 paper, check (Bottou, 2010).

# Stochastic optimisation

For large $N$, we can use a gradient estimate

$$\nabla_\theta \mathcal{L}_\mathcal{D}(\theta) = \underbrace{\mathbb{E}_{S\sim\mathcal{U}(1/N)}\left[N\nabla_\theta \log p(y^{(S)}|x^{(S)},\theta)\right]}_{\text{expected gradient :)}}$$

$$\overset{\text{MC}}{\approx} \frac{1}{M}\sum_{m=1}^{M} N\nabla_\theta \log p(y^{(s_m)}|x^{(s_m)},\theta) \qquad \text{with } S_m \sim \mathcal{U}(1/N)$$

The theory of stochastic optimisation (Robbins and Monro, 1951) tells us that we will converge to a local optimum of the objective as long as we take steps that are correct *on average*. This means we can optimise with stochastic gradient estimates, for as long as they are unbiased estimates of the exact gradient.

Do you see the guarantee and the condition?

There are more conditions, however. The learning rate must comply with some key properties. Luckily many learning rate schedules have been documented in the literature, and most our famous optimisers meet the Robbis and Monro conditions (though not all).

If you want to read more, but need something more accessible than the 1951 paper, check (Bottou, 2010).

# Stochastic optimisation

For large $N$, we can use a gradient estimate

$$\nabla_\theta \mathcal{L}_\mathcal{D}(\theta) = \underbrace{\mathbb{E}_{S\sim\mathcal{U}(1/N)}\left[N\nabla_\theta \log p(y^{(S)}|x^{(S)},\theta)\right]}_{\text{expected gradient :)}}$$

$$\overset{\text{MC}}{\approx} \frac{1}{M}\sum_{m=1}^{M} N\nabla_\theta \log p(y^{(s_m)}|x^{(s_m)},\theta) \qquad \text{with } S_m \sim \mathcal{U}(1/N)$$

$$= \nabla_\theta \underbrace{\frac{N}{M}\sum_{m=1}^{M} \log p(y^{(s_m)}|x^{(s_m)},\theta)}_{\mathcal{L}_\mathcal{B}(\theta)}$$

The theory of stochastic optimisation (Robbins and Monro, 1951) tells us that we will converge to a local optimum of the objective as long as we take steps that are correct *on average*. This means we can optimise with stochastic gradient estimates, for as long as they are unbiased estimates of the exact gradient.

Do you see the guarantee and the condition?

There are more conditions, however. The learning rate must comply with some key properties. Luckily many learning rate schedules have been documented in the literature, and most our famous optimisers meet the Robbis and Monro conditions (though not all).

If you want to read more, but need something more accessible than the 1951 paper, check (Bottou, 2010).

# Stochastic optimisation

For large $N$, we can use a gradient estimate

$$\nabla_\theta \mathcal{L}_\mathcal{D}(\theta) = \underbrace{\mathbb{E}_{S\sim\mathcal{U}(1/N)}\left[N\nabla_\theta \log p(y^{(S)}|x^{(S)},\theta)\right]}_{\text{expected gradient :)}}$$

$$\overset{\text{MC}}{\approx} \frac{1}{M}\sum_{m=1}^{M} N\nabla_\theta \log p(y^{(s_m)}|x^{(s_m)},\theta) \qquad \text{with } S_m \sim \mathcal{U}(1/N)$$

$$= \nabla_\theta \underbrace{\frac{N}{M}\sum_{m=1}^{M} \log p(y^{(s_m)}|x^{(s_m)},\theta)}_{\mathcal{L}_\mathcal{B}(\theta)}$$

and take a step in the direction

$$\gamma \frac{N}{M} \underbrace{\nabla_\theta \mathcal{L}_\mathcal{B}(\theta)}_{\text{stochastic gradient}}$$

where $\mathcal{B} = \{(x^{(s_1)}, y^{(s_1)}), \ldots, (x^{(s_M)}, y^{(s_M)})\}$ is a random mini-batch

The theory of stochastic optimisation (Robbins and Monro, 1951) tells us that we will converge to a local optimum of the objective as long as we take steps that are correct *on average*. This means we can optimise with stochastic gradient estimates, for as long as they are unbiased estimates of the exact gradient.

Do you see the guarantee and the condition?

There are more conditions, however. The learning rate must comply with some key properties. Luckily many learning rate schedules have been documented in the literature, and most our famous optimisers meet the Robbis and Monro conditions (though not all).

If you want to read more, but need something more accessible than the 1951 paper, check (Bottou, 2010).

# Summary – a recipe for supervised learning

Maximum likelihood estimation

- tells you which loss to optimise
  (i.e. negative log-likelihood)

Automatic differentiation (*backprop*) with gradient surrogates

- a tractable and differentiable forward computation whose backward is an unbiased estimate of the intended gradient

Stochastic optimisation powered by backprop

- general purpose gradient-based optimisers

Our main job is to pick an appropriate family of distributions.

Paper recommendation: for a comprehensive understanding of stochastic computation graphs (Schulman et al., 2015).

# Outline

# Let's generate some images

Conditional generation task: image generation from text (or from some tabular data). Our images are all of fixed dimensionality ($D$).

Given the an input $x$, how about we assume images are drawn from an MVN?

Each $x$ is mapped to an average output $\mu(x;\theta)$ and a covariance matrix $\Sigma(x;\theta)$

$$Y|X = x \sim \mathcal{N}(\mu(x;\theta), \Sigma(x;\theta)) \tag{3a}$$
$$\mathbf{h} = \text{encode}(x;\theta_{\text{enc}}) \tag{3b}$$
$$\mathbf{L} = \text{linear}_{(D-1)\times D/2}(\mathbf{h};\theta_{\text{off}}) \tag{3c}$$
$$\mathbf{s} = \text{softplus}(\text{linear}_D(\mathbf{h};\theta_{\text{diag}})) \tag{3d}$$
$$\mathbf{C} = \text{lowtri}(\mathbf{L}) + \text{diag}(\mathbf{s}) \tag{3e}$$
$$\mu(x;\theta) = \text{linear}_D(\mathbf{h};\theta_{\text{loc}}) \tag{3f}$$
$$\Sigma(x;\theta) = \mathbf{C}\mathbf{C}^\top \tag{3g}$$

- if $x$ is a `cute cat`, then $\mu(x;\theta)$ is something like the 'average cat';
- in an MVN, output dimensions are (linearly) correlated: $y = \mu(x;\theta) + \mathbf{C}\mathbf{u}$ where $u_d \sim \mathcal{N}(0,1)$;

# Let's generate some images

Conditional generation task: image generation from text (or from some tabular data). Our images are all of fixed dimensionality ($D$).

Given the an input $x$, how about we assume images are drawn from an MVN? What's wrong with this idea?

Each $x$ is mapped to an average output $\mu(x;\theta)$ and a covariance matrix $\Sigma(x;\theta)$

$$Y|X = x \sim \mathcal{N}(\mu(x;\theta), \Sigma(x;\theta)) \tag{3a}$$
$$\mathbf{h} = \text{encode}(x;\theta_{\text{enc}}) \tag{3b}$$
$$\mathbf{L} = \text{linear}_{(D-1)\times D/2}(\mathbf{h};\theta_{\text{off}}) \tag{3c}$$
$$\mathbf{s} = \text{softplus}(\text{linear}_D(\mathbf{h};\theta_{\text{diag}})) \tag{3d}$$
$$\mathbf{C} = \text{lowtri}(\mathbf{L}) + \text{diag}(\mathbf{s}) \tag{3e}$$
$$\mu(x;\theta) = \text{linear}_D(\mathbf{h};\theta_{\text{loc}}) \tag{3f}$$
$$\Sigma(x;\theta) = \mathbf{C}\mathbf{C}^\top \tag{3g}$$

- if $x$ is a `cute cat`, then $\mu(x;\theta)$ is something like the 'average cat';
- in an MVN, output dimensions are (linearly) correlated: $y = \mu(x;\theta) + \mathbf{C}\mathbf{u}$ where $u_d \sim \mathcal{N}(0,1)$;

# Let's generate some images

Conditional generation task: image generation from text (or from some tabular data). Our images are all of fixed dimensionality ($D$).

Given the an input $x$, how about we assume images are drawn from an MVN? What's wrong with this idea?

Do we really believe that cats follow a Gaussian distribution?



Each $x$ is mapped to an average output $\mu(x;\theta)$ and a covariance matrix $\Sigma(x;\theta)$

$$Y|X = x \sim \mathcal{N}(\mu(x;\theta), \Sigma(x;\theta)) \tag{4a}$$
$$\mathbf{h} = \text{encode}(x;\theta_{\text{enc}}) \tag{4b}$$
$$\mathbf{L} = \text{linear}_{(D-1)\times D/2}(\mathbf{h};\theta_{\text{off}}) \tag{4c}$$
$$\mathbf{s} = \text{softplus}(\text{linear}_D(\mathbf{h};\theta_{\text{diag}})) \tag{4d}$$
$$\mathbf{C} = \text{lowtri}(\mathbf{L}) + \text{diag}(\mathbf{s}) \tag{4e}$$
$$\mu(x;\theta) = \text{linear}_D(\mathbf{h};\theta_{\text{loc}}) \tag{4f}$$
$$\Sigma(x;\theta) = \mathbf{C}\mathbf{C}^\top \tag{4g}$$

- if $x$ is a `cute cat`, then $\mu(x;\theta)$ is something like the 'average cat';
- in an MVN, output dimensions are (linearly) correlated: $y = \mu(x;\theta) + \mathbf{C}\mathbf{u}$ where $u_d \sim \mathcal{N}(0,1)$;

# Let's generate some images

It's hard to imagine that each cat is a simple linear transformation away from the *mean cat* (pun intended).

We chose the MVN because we needed a tractable density for MLE-training, not because we expected it to be appropriate for our data.

Well, textbooks don't have cat-specific distributions, nor do we want to develop one distribution for each type of category in imagenet.

We need a more general tool!

# Chain rule is your friend

There's this amazing result in probability theory, it tells us that every joint pdf over $D$ variables can be re-expressed as a product of univariate pdfs as follows:

$$p(\langle y_1, \ldots, y_D \rangle) = \prod_{d=1}^{D} p(y_d | y_{<d}) \tag{5}$$

Watch out! Chain rule does not tell you that your conditionals can be Gaussian (or any other choice for that matter), chain rule is a formal result assuring you that this factorisation exists, any parametric assumptions you make is on you.

We can generalise this result to sequences of variable finite length (i.e., where $D$ is itself a random variable), but it takes quite a bit more measure theory to do so. For a modern paper discussing this in the context of language models, see (Du et al., 2023).

# Chain rule is your friend

There's this amazing result in probability theory, it tells us that every joint pdf over $D$ variables can be re-expressed as a product of univariate pdfs as follows:

$$p(\langle y_1, \ldots, y_D \rangle) = \prod_{d=1}^{D} p(y_d | y_{<d}) \tag{5}$$

Conditional modelling is no problem:

$$p(\langle y_1, \ldots, y_D \rangle | x) = \prod_{d=1}^{D} p(y_d | x, y_{<d}) \tag{6}$$

Watch out! Chain rule does not tell you that your conditionals can be Gaussian (or any other choice for that matter), chain rule is a formal result assuring you that this factorisation exists, any parametric assumptions you make is on you.

We can generalise this result to sequences of variable finite length (i.e., where $D$ is itself a random variable), but it takes quite a bit more measure theory to do so. For a modern paper discussing this in the context of language models, see (Du et al., 2023).

# Autoregressive models – Option 1: known pdf

Chain rule:

$$p(\langle y_1, \ldots, y_D \rangle | x) = \prod_{d=1}^{D} p(y_d | x, y_{<d}) \tag{7}$$

We parameterise our conditional pdfs by encoding the conditioning context (any fixed inputs, such as $x$, and the history of already generated variables $y_{<d}$) into some fixed-dimensional vector $\mathbf{h}_d \in \mathbf{R}^H$ and then using this vector to predict the parameters of a known pdf.

Let $\mathbf{h}_d = \text{encode}(x, y_{<d}; \theta)$.

We could assume the conditional pixel distribution to be Gaussian:

$$p(\langle y_1, \ldots, y_D \rangle | x) \triangleq \prod_{d=1}^{D} \mathcal{N}(y_d | \mu(\mathbf{h}_d; \theta); \sigma^2(\mathbf{h}_d; \theta)) \tag{8}$$

Because $D$ is fixed, for an observed sequence $y_{1:D}$, we can compute all states with a single feed-forward network with masked weights that guarantee autoregressiveness (for details, see MADE (Germain et al., 2015b)).

But, who says pixel distributions resemble Gaussians? Perhaps there's skew and multimodality (even for a given prefix $y_{<d}$; e.g., given the first row of pixels, the next pixel could be part of a cat, of the furniture, of the background, etc.).

# Autoregressive models – Option 2: combining known pdfs

Chain rule:

$$p(\langle y_1, \ldots, y_D \rangle | x) = \prod_{d=1}^{D} p(y_d | x, y_{<d}) \tag{9}$$

We parameterise our conditional pdfs by encoding the conditioning context (any fixed inputs, such as $x$, and the history of already generated variables $y_{<d}$) into some fixed-dimensional vector $\mathbf{h}_d \in \mathbf{R}^H$ and then using this vector to predict the parameters of a known pdf.

Let $\mathbf{h}_d = \text{encode}(x, y_{<d}; \theta)$.

We could assume the conditional pixel distribution is a mixture of K Gaussians (with trainable mixing coefficients $(w_1, \ldots, w_K)^\top \in \Delta_{K-1}$, possibly predicted from $\mathbf{h}_d$):

$$p(\langle y_1, \ldots, y_D \rangle | x) \triangleq \prod_{d=1}^{D} \sum_{k=1}^{K} w_k \mathcal{N}(y_d | \mu_k(\mathbf{h}_d; \theta); \sigma_k^2(\mathbf{h}; \theta)) \tag{10}$$

These ideas were the essence of *neural autoregressive density estimation* (NADE; Uria et al., 2014). Note: NADE also had a clever way to take various different orders of the pixels into account (since images aren't actual sequences, this sometimes led to better models).

How could we go beyond a mixture of known pdfs and learn a univariate conditional that's more flexible?

# Autoregressive models – Option 3: known pmf

Chain rule:

$$p(\langle y_1, \ldots, y_D \rangle | x) = \prod_{d=1}^{D} p(y_d | x, y_{<d}) \tag{11}$$

We parameterise our conditional pdfs by encoding the conditioning context (any fixed inputs, such as $x$, and the history of already generated variables $y_{<d}$) into some fixed-dimensional vector $\mathbf{h}_d \in \mathbf{R}^H$ and then using this vector to predict the parameters of a known pdf.

Let $\mathbf{h}_d = \text{encode}(x, y_{<d}; \theta)$.

We could discretise the pixel intensities (e.g., using 256 levels) and assume the conditional pixel distribution is Categorical:

$$p(\langle y_1, \ldots, y_D \rangle | x) \triangleq \prod_{d=1}^{D} \text{Categorical}(y_d | \boldsymbol{\pi}(\mathbf{h}_d; \theta)) \tag{12}$$

Along with many other things (e.g., novel architectures), this extension of NADE was at the core of PixelRNNs (van den Oord et al., 2016) and PixelCNNs (van den Oord et al., 2016).

# Autoregressive models – Language models

The output variable is a sequence of $J$ discrete symbols ($J$ is finite but not fixed).

Chain rule:

$$p(\langle y_1, \ldots, y_J \rangle | x) = \prod_{j=1}^{J} p(y_j | x, y_{<j}) \tag{13}$$

We parameterise our conditional pdfs by encoding the conditioning context (any fixed inputs, such as $x$, and the history of already generated variables $y_{<j}$) into some fixed-dimensional vector $\mathbf{h}_j \in \mathbf{R}^H$ and then using this vector to predict the parameters of a known pdf.

Let $\mathbf{h}_j = \text{encode}(x, y_{<j}; \theta)$.

Assuming the vocabulary of known symbols is finite and that its size is manageable (given the typical hardware we have) and that conditionals are *dense* (every symbol is assigned strictly positive mass), then the conditional pixel distribution is Categorical:

$$p(\langle y_1, \ldots, y_J \rangle | x) \triangleq \prod_{j=1}^{J} \text{Categorical}(y_j | \boldsymbol{\pi}(\mathbf{h}_j; \theta)) \tag{14}$$

This is what we call an autoregressive language model, the 2010 rendition of it employed RNN cells (Mikolov et al., 2010).

# Autoregressive models – Time series

A factorisation using chain rule is such a general tool, you can use for any time-series-type data.

You will need to motivate a design choice for your conditionals. For example,

- in floods forecasting, it looks like they have good reasons to model with Laplace distributions (Nearing et al., 2024)
- in forecasting voting intentions, it's common to use Dirichlet (and related) distributions (Gordon-Rodriguez et al., 2020)

# Autoregressive models – Details

At this level of generality, very little changes from one application to another, we essentially only motivate a design choice for the conditional factors.

It's the implementation of the encoding function that typically holds the key to a successful application. The options are numerous:

- Recurrent networks (Hochreiter and Schmidhuber, 1997; Cho et al., 2014)
- Masked dense networks (Germain et al., 2015b)
- Convolutional networks (van den Oord et al., 2016; Kalchbrenner et al., 2016)
- Transformers (Vaswani et al., 2017)
- Structured state-space models (Gu et al., 2022; Gu and Dao, 2023)

The output variable is a sequence of finite length $J$, which may or may not vary.

Chain rule:

$$p(\langle y_1, \ldots, y_J \rangle | x) = \prod_{j=1}^{J} p(y_j | x, y_{<j}) \tag{15}$$

We parameterise our conditional pdfs by encoding the conditioning context (any fixed inputs, such as $x$, and the history of already generated variables $y_{<j}$) into some fixed-dimensional vector $\mathbf{h}_j \in \mathbf{R}^H$ and then using this vector to predict the parameters of a known pdf.

# Assessing the joint pdf

This is efficient by design. For a reasonable choice, the worst case should be linear in $J$.

Consider an LM as example:

$$p(\langle y_1, \ldots, y_J \rangle | x) \triangleq \prod_{j=1}^{J} \text{Categorical}(y_j | \boldsymbol{\pi}(\mathbf{h}_j; \theta)) \tag{16}$$

# Assessing the joint pdf

This is efficient by design. For a reasonable choice, the worst case should be linear in $J$.

Consider an LM as example:

$$p(\langle y_1, \ldots, y_J \rangle | x) \triangleq \prod_{j=1}^{J} \text{Categorical}(y_j | \boldsymbol{\pi}(\mathbf{h}_j; \theta)) \tag{16}$$

- we observe $y_{1:J}$, with computation time linear in $J$ we gather encodings $\mathbf{h}_1, \ldots, \mathbf{h}_J$ for the prediction of each cpd;

# Assessing the joint pdf

This is efficient by design. For a reasonable choice, the worst case should be linear in $J$.

Consider an LM as example:

$$p(\langle y_1, \ldots, y_J \rangle | x) \triangleq \prod_{j=1}^{J} \text{Categorical}(y_j | \boldsymbol{\pi}(\mathbf{h}_j; \theta)) \tag{16}$$

- we observe $y_{1:J}$, with computation time linear in $J$ we gather encodings $\mathbf{h}_1, \ldots, \mathbf{h}_J$ for the prediction of each cpd;
- for each of these $\mathbf{h}_j$, we predict a Categorical parameter (e.g., using an FFNN with softmax output) and look up the probability mass corresponding to the observed $y_j$

# Assessing the joint pdf

This is efficient by design. For a reasonable choice, the worst case should be linear in $J$.

Consider an LM as example:

$$p(\langle y_1, \ldots, y_J \rangle | x) \triangleq \prod_{j=1}^{J} \text{Categorical}(y_j | \boldsymbol{\pi}(\mathbf{h}_j; \theta)) \tag{16}$$

- we observe $y_{1:J}$, with computation time linear in $J$ we gather encodings $\mathbf{h}_1, \ldots, \mathbf{h}_J$ for the prediction of each cpd;
- for each of these $\mathbf{h}_j$, we predict a Categorical parameter (e.g., using an FFNN with softmax output) and look up the probability mass corresponding to the observed $y_j$

When we assess the mass/density of a complete sequence, the entire sequence is already known to us, so, depending on the architecture that computes the encodings $\mathbf{h}_1, \ldots, \mathbf{h}_J$ this computation can be parallelised.

For example, the state $\mathbf{h}_j$ of an LSTM cannot be re-expressed as a computation that's independent of $\mathbf{h}_{j-1}$, so LSTMs can hardly be accelerated to sub-linear computation time.

The state $\mathbf{h}_j$ of a Transformer depends on all of $y_{<j}$, but is independent of $\mathbf{h}_{j-1}$, hence with enough GPU cores and memory, we can parallelise the computation of the $J$ states $\mathbf{h}_1, \ldots, \mathbf{h}_J$.

For illustration's sake, here's a choice that's worst than linear: $\mathbf{h}_j = \text{BiLSTM}(y_{<j})$. Do you see why that's so?

# Sampling

This is efficient by design.

We obtain a sample by drawing iteratively:

# Sampling

This is efficient by design.

We obtain a sample by drawing iteratively:

1. $j \leftarrow 0$ and $s \leftarrow \langle\rangle$;

# Sampling

This is efficient by design.

We obtain a sample by drawing iteratively:

1. $j \leftarrow 0$ and $s \leftarrow \langle\rangle$;
2. increment $j$ and draw outcome $o$ with probability mass/density $p(Y_j = o | X = x, Y_{<j} = s)$

# Sampling

This is efficient by design.

We obtain a sample by drawing iteratively:

1. $j \leftarrow 0$ and $s \leftarrow \langle\rangle$;
2. increment $j$ and draw outcome $o$ with probability mass/density $p(Y_j = o | X = x, Y_{<j} = s)$
3. append $o$ to $s$, repeat (2) until a termination criterion is met
   e.g., $j = J$ (if $J$ never varies), or $j$ achieves a predefined maximum length, or $o$ is a terminating outcomes (EoS in LMs), etc.

# Sampling

This is efficient by design.

We obtain a sample by drawing iteratively:

1. $j \leftarrow 0$ and $s \leftarrow \langle\rangle$;
2. increment $j$ and draw outcome $o$ with probability mass/density $p(Y_j = o | X = x, Y_{<j} = s)$
3. append $o$ to $s$, repeat (2) until a termination criterion is met
   e.g., $j = J$ (if $J$ never varies), or $j$ achieves a predefined maximum length, or $o$ is a terminating outcomes (EoS in LMs), etc.

In step (2) we draw from a cpd (this is typically efficient for known pdfs, and mixture of known pdfs). Before we can draw, we need to predict that cpd, which requires computing $\mathbf{h}_j$.

# Sampling

This is efficient by design.

We obtain a sample by drawing iteratively:

1. $j \leftarrow 0$ and $s \leftarrow \langle\rangle$;
2. increment $j$ and draw outcome $o$ with probability mass/density $p(Y_j = o | X = x, Y_{<j} = s)$
3. append $o$ to $s$, repeat (2) until a termination criterion is met
   e.g., $j = J$ (if $J$ never varies), or $j$ achieves a predefined maximum length, or $o$ is a terminating outcomes (EoS in LMs), etc.

In step (2) we draw from a cpd (this is typically efficient for known pdfs, and mixture of known pdfs). Before we can draw, we need to predict that cpd, which requires computing $\mathbf{h}_j$.

For an LSTM, $\mathbf{h}_j$ is a constant-time operation away from $\mathbf{h}_{j-1}$, which we have from the previous step. Say updating the state takes time $\mathcal{O}(C)$, then sampling runs in time $\mathcal{O}(J \times C)$

For a Transformer, knowing $\mathbf{h}_{j-1}$ (which we computed in the previous step) does not help, as the computation of $\mathbf{h}_j$ depends on $y_{<j}$ (and not on $\mathbf{h}_{<j}$). Hence we need to go through the entire Transformer stack with the extended history. Say running one Transformer layer takes time $\mathcal{O}(T)$, then sampling runs in time $\mathcal{O}(J \times L \times T)$, where $L$ is the depth of the Transformer stack. For a Transformer, $T$ is in fact quadratic in the length of the history, but on GPU, we can parallelise those computations to (at least) linear time.

# Honourable mentions

With careful assumptions on the encoding function, its computations may be equivalently expressible as a recurrence or a convolution. When that happens, assessing the pdf and sampling are equally efficient (provided we have appropriate hardware and primitives for maximum parallelism).

To learn more about this, read about S4-type models (Gu et al., 2022; Gu and Dao, 2023).

I am not the right person to teach you those.

Masked-dense layers, CNNs, and Transformers all enjoy fast density assessment (when the sequence is observed) and slower sampling.

S4-type models use their convolutional view for fast training, but offer an equivalent recurrent view for fast sampling.

# Search: making optimal decisions

Because autoregressive models make no conditional independence assumptions, this is as inefficient as it gets.

The notion of an optimal decision requires a decision rule (a 'decoding algorithm'). For example, with discrete data it's common to use the rule

$$y^\star = \arg\max_{c \in \mathcal{Y}} \sum_{j=1}^{J} \log p(c_j | x, c_{<j}) \tag{17}$$

Any choice $c_j$ will affect the probability mass/density of choices $c_{>j}$, hence there's no tractable solution to this search problem.

Common heuristics include: greedy search, beam-search, biased samplers (e.g., top-k, top-p, temperature, etc.).

# Search: making optimal decisions

Because autoregressive models make no conditional independence assumptions, this is as inefficient as it gets.

The notion of an optimal decision requires a decision rule (a 'decoding algorithm'). For example, with discrete data it's common to use the rule

$$y^\star = \arg\max_{c \in \mathcal{Y}} \sum_{j=1}^{J} \log p(c_j | x, c_{<j}) \tag{17}$$

Any choice $c_j$ will affect the probability mass/density of choices $c_{>j}$, hence there's no tractable solution to this search problem.

Common heuristics include: greedy search, beam-search, biased samplers (e.g., top-k, top-p, temperature, etc.).

A more general decision rule (which includes the former as special case) searches for:

$$y^\star = \arg\max_{c \in \mathcal{Y}} \mathbb{E}[u(c, Y)] \tag{18}$$

where $u(c, y)$ is a utility function assessing a candidate $c$ against a response $y$, and the expectation is taken with respect to the joint pdf $p(y|x)$.

Besides being more general, this can address fundamental limitations of the special case. See for example, (Eikema and Aziz, 2020) to learn about those limitations and (Eikema and Aziz, 2022) to learn about algorithmic approximations to the general decision rule.

# What should I use?

Look around for relevant literature, base your choices on what is known to work well for similar data.

Some considerations are somewhat logical:

- Transformers need many layers to learn complex composition functions; that is, Transformers need to be *deep*. This likely means they need bigger data.
- The fact that all positions are a constant number of operations away from one another gives (deep enough) Transformers a good chance to learn long-range dependencies (that are spread arbitrarily far away along a very long sequence). But you will need topnotch hardware for that.
- LSTMs are in principle able to learn a wider class of functions than Transformers (because Transformers learn composition functions of fixed depth).

For various results relating Transformers to different formal languages, see for example (Hahn, 2020; Bhattamishra et al., 2020; Strobl et al., 2023; Hahn and Rofin, 2024).

# Summary

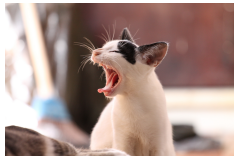
- Chain rule allows us to build complex joint pdfs using simpler tools (typically, known pdfs).
- We need to make a design choice for our conditional pdfs: this choice is mostly constrained by data type.
- We need an encoding function to process conditioning context: this choice depends on various factors (complexity of data, availability of training data, etc.).
- Convolutional encoding functions are typically highly parallelisable during training (density computation).
- Recurrent encoding function are efficient for sampling.
- Search is generally intractable

# Outline

# The problem with known pdfs: the case of pictures

Have you modelled images (or their pixels) as Gaussian variables? Do we really believe that they follow a Gaussian distribution?





# The problem with known pdfs: the case of word embeddings

What do you think word embeddings distribute like?

# The problem with known pdfs: the case of word embeddings

What do you think word embeddings distribute like?

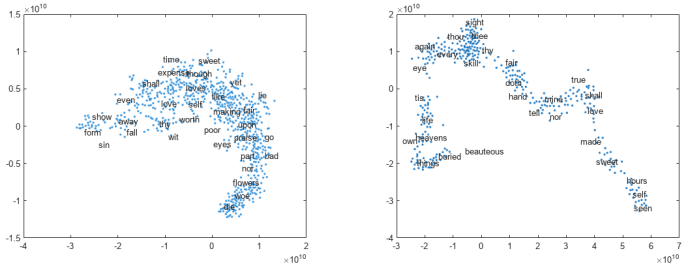



Figure: tSNE projection from $\mathbb{R}^{50}$ (left) and $\mathbb{R}^{100}$ (right); see https://it.mathworks.com/help/textanalytics/ref/trainwordembedding.html

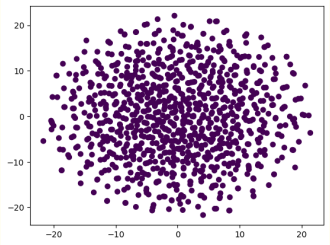


Figure: tSNE projection of 1000 samples from $\mathcal{N}(0, I_{50})$.

# Calculus for the rescue: reparametrisation using a bijection

Express the density of a variable $Y$ in terms of the density of a variable $E$. Assume that a differentiable, invertible mapping $h : \mathcal{E} \to \mathcal{Y}$ exists.

$$h(\epsilon) = y$$

For example, if $y \in \mathbb{R}^D$, we could use a multivariate Normal for $p_E$.

For an in-depth view of the maths behind this, check Sections 1 and 2 of https://probabll.github.io/slides/DL2/2023/vi-continuous-appendix.pdf

# Calculus for the rescue: reparametrisation using a bijection

Express the density of a variable $Y$ in terms of the density of a variable $E$. Assume that a differentiable, invertible mapping $h : \mathcal{E} \to \mathcal{Y}$ exists.

$$h(\epsilon) = y$$
$$p_Y(y) = p_E(h^{-1}(y))|\det J_{h^{-1}}(y)|$$

For example, if $y \in \mathbb{R}^D$, we could use a multivariate Normal for $p_E$.

For an in-depth view of the maths behind this, check Sections 1 and 2 of https://probabll.github.io/slides/DL2/2023/vi-continuous-appendix.pdf

# Calculus for the rescue: reparametrisation using a bijection

Express the density of a variable $Y$ in terms of the density of a variable $E$. Assume that a differentiable, invertible mapping $h : \mathcal{E} \to \mathcal{Y}$ exists.

$$h(\epsilon) = y$$
$$p_Y(y) = p_E(h^{-1}(y))|\det J_{h^{-1}}(y)|$$
$$p_E(\epsilon) = p_Y(h(\epsilon))|\det J_h(\epsilon)|$$

For example, if $y \in \mathbb{R}^D$, we could use a multivariate Normal for $p_E$.

For an in-depth view of the maths behind this, check Sections 1 and 2 of https://probabll.github.io/slides/DL2/2023/vi-continuous-appendix.pdf

# Calculus for the rescue: reparametrisation using a bijection

Express the density of a variable $Y$ in terms of the density of a variable $E$. Assume that a differentiable, invertible mapping $h : \mathcal{E} \to \mathcal{Y}$ exists.

$$h(\epsilon) = y$$
$$p_Y(y) = p_E(h^{-1}(y))|\det J_{h^{-1}}(y)|$$
$$p_E(\epsilon) = p_Y(h(\epsilon))|\det J_h(\epsilon)|$$

Challenge the mapping $h$ (or its inverse) needs to be defined.

For example, if $y \in \mathbb{R}^D$, we could use a multivariate Normal for $p_E$.

For an in-depth view of the maths behind this, check Sections 1 and 2 of https://probabll.github.io/slides/DL2/2023/vi-continuous-appendix.pdf

# Normalising Flows

Use an NN to learn the transformation $h$ (or its inverse). We will have to constrain our NN carefully to guarantee that $h$ is bijective/invertible.

If we want $p_Y(y)$, we need to provide $|\det J_{h^{-1}}(y)|$ **in the forward pass**.

We are going to devise ways to get $|\det J_{h^{-1}}(y)|$ efficiently.

# Core idea

Decompose mapping $h : \mathcal{E} \to \mathcal{Y}$ into

$$h = h_1 \circ h_2 \circ \ldots \circ h_K$$

Because $h$ is bijective by design, the two views are equivalent.

But, as we will be parameterising $h$ using an NN, we might prefer to parameterise $h$ or $h^{-1}$.

In fact, even though $h$ is bijective by design (and we will make sure this is true), depending on how we parameterise it (or its inverse), the other direction might not be known to us (i.e., we may not be able to computationally invert the function, even though it's constrained to be bijective).

# Core idea

Decompose mapping $h : \mathcal{E} \to \mathcal{Y}$ into

$$h = h_1 \circ h_2 \circ \ldots \circ h_K$$

or, equivalently,

$$h^{-1} = h_K^{-1} \circ h_{K-1}^{-1} \circ \ldots \circ h_1^{-1} \ .$$

Because $h$ is bijective by design, the two views are equivalent.

But, as we will be parameterising $h$ using an NN, we might prefer to parameterise $h$ or $h^{-1}$.

In fact, even though $h$ is bijective by design (and we will make sure this is true), depending on how we parameterise it (or its inverse), the other direction might not be known to us (i.e., we may not be able to computationally invert the function, even though it's constrained to be bijective).

# Core idea

Decompose mapping $h : \mathcal{E} \to \mathcal{Y}$ into

$$h = h_1 \circ h_2 \circ \ldots \circ h_K$$

or, equivalently,

$$h^{-1} = h_K^{-1} \circ h_{K-1}^{-1} \circ \ldots \circ h_1^{-1} \ .$$

Now we can learn $K$ mappings with simple Jacobian determinants.

Because $h$ is bijective by design, the two views are equivalent.

But, as we will be parameterising $h$ using an NN, we might prefer to parameterise $h$ or $h^{-1}$.

In fact, even though $h$ is bijective by design (and we will make sure this is true), depending on how we parameterise it (or its inverse), the other direction might not be known to us (i.e., we may not be able to computationally invert the function, even though it's constrained to be bijective).

# Core idea

Decompose mapping $h : \mathcal{E} \to \mathcal{Y}$ into

$$h = h_1 \circ h_2 \circ \ldots \circ h_K$$

or, equivalently,

$$h^{-1} = h_K^{-1} \circ h_{K-1}^{-1} \circ \ldots \circ h_1^{-1} \ .$$

Now we can learn $K$ mappings with simple Jacobian determinants.

$$p_E(\epsilon) = p_Y(y)\left|\det J_{h_1}(y^{(1)})\right|\left|\det J_{h_2}(y^{(2)})\right| \ldots |\det J_{h_K}(\epsilon)|$$

Because $h$ is bijective by design, the two views are equivalent.

But, as we will be parameterising $h$ using an NN, we might prefer to parameterise $h$ or $h^{-1}$.

In fact, even though $h$ is bijective by design (and we will make sure this is true), depending on how we parameterise it (or its inverse), the other direction might not be known to us (i.e., we may not be able to computationally invert the function, even though it's constrained to be bijective).

# Core idea

Decompose mapping $h : \mathcal{E} \to \mathcal{Y}$ into

$$h = h_1 \circ h_2 \circ \ldots \circ h_K$$

or, equivalently,

$$h^{-1} = h_K^{-1} \circ h_{K-1}^{-1} \circ \ldots \circ h_1^{-1} \ .$$

Now we can learn $K$ mappings with simple Jacobian determinants.

$$p_E(\epsilon) = p_Y(y)\left|\det J_{h_1}(y^{(1)})\right|\left|\det J_{h_2}(y^{(2)})\right| \ldots |\det J_{h_K}(\epsilon)|$$

$$p_Y(y) = p_E(\epsilon)\left|\det J_{h_K^{-1}}(y^{(K-1)})\right|\left|\det J_{h_{K-1}^{-1}}(y^{(K-2)})\right| \ldots \left|\det J_{h_1^{-1}}(y)\right|$$

Because $h$ is bijective by design, the two views are equivalent.

But, as we will be parameterising $h$ using an NN, we might prefer to parameterise $h$ or $h^{-1}$.

In fact, even though $h$ is bijective by design (and we will make sure this is true), depending on how we parameterise it (or its inverse), the other direction might not be known to us (i.e., we may not be able to computationally invert the function, even though it's constrained to be bijective).

# Normalising Flows

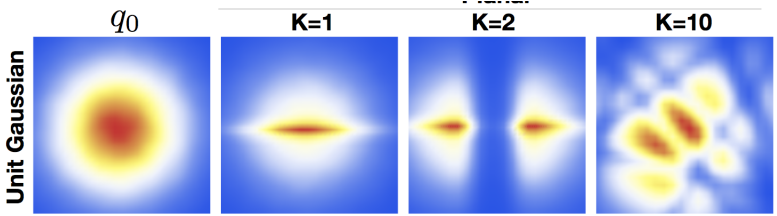



Figure: Taken from Rezende and Mohamed (2015)

# Applications

Density estimation (today): $Y$ is our data variable with unknown distribution, we know $p_E$ and have the means to learn $h^{-1} : \mathcal{Y} \to \mathcal{E}$.

Inference model (second half of the module): $Y$ is a latent variable with unknown distribution, we know $p_E$ and have the means to learn $h : \mathcal{E} \to \mathcal{Y}$.

# Normalising Flows for Density Estimation

## Setting

Our data $y$ has unknown continuous density $p_Y(y)$. We can therefore not handcraft a likelihood.

# Normalising Flows for Density Estimation

## Setting

Our data $y$ has unknown continuous density $p_Y(y)$. We can therefore not handcraft a likelihood.

**Examples:** word embeddings, pictures

# Normalising Flows for Density Estimation

## Setting

Our data $y$ has unknown continuous density $p_Y(y)$. We can therefore not handcraft a likelihood.

**Examples:** word embeddings, pictures

## Goal

Transform observed variable $y$ into $\epsilon = h^{-1}(y)$ with **known** density $p_E(\epsilon)$ and express the likelihood as

# Normalising Flows for Density Estimation

## Setting

Our data $y$ has unknown continuous density $p_Y(y)$. We can therefore not handcraft a likelihood.

**Examples:** word embeddings, pictures

## Goal

Transform observed variable $y$ into $\epsilon = h^{-1}(y)$ with **known** density $p_E(\epsilon)$ and express the likelihood as

$$p_Y(y) = p_E(h^{-1}(y))|\det J_{h^{-1}}(y)|$$

# Normalising Flows for Density Estimation

## Setting

Our data $y$ has unknown continuous density $p_Y(y)$. We can therefore not handcraft a likelihood.

**Examples:** word embeddings, pictures

## Goal

Transform observed variable $y$ into $\epsilon = h^{-1}(y)$ with **known** density $p_E(\epsilon)$ and express the likelihood as

$$\begin{aligned} p_Y(y) &= p_E(h^{-1}(y))|\det J_{h^{-1}}(y)| \\ &= p_E(h_K^{-1}(y^{(K-1)}))\left|\det J_{h_{K-1}^{-1}}(y^{(K-2)})\right| \dots \left|\det J_{h_1^{-1}}(y)\right| \end{aligned}$$

# 2-step Flow

$$p_Y(y) = p_E(\overbrace{\epsilon}^{y^{(2)}}) \left| \det J_{h_2^{-1}}(y^{(1)}) \right| \left| \det J_{h_1^{-1}}(y) \right|$$

# 2-step Flow

$$
\begin{aligned}
p_Y(y) &= p_E(\overbrace{\epsilon}^{y^{(2)}}) \left|\det J_{h_2^{-1}}(y^{(1)})\right| \left|\det J_{h_1^{-1}}(y)\right| \\
&= p_E(h_2^{-1}(h_1^{-1}(y))) \left|\det J_{h_2^{-1}}(h_1^{-1}(y))\right| \left|\det J_{h_1^{-1}}(y)\right|
\end{aligned}
$$

# 2-step Flow

$$
\begin{aligned}
p_Y(y) &= p_E(\overbrace{\epsilon}^{y^{(2)}}) \left| \det J_{h_2^{-1}}(y^{(1)}) \right| \left| \det J_{h_1^{-1}}(y) \right| \\
&= p_E(h_2^{-1}(h_1^{-1}(y))) \left| \det J_{h_2^{-1}}(h_1^{-1}(y)) \right| \left| \det J_{h_1^{-1}}(y) \right|
\end{aligned}
$$

The transformations $h_2^{-1}$ and $h_1^{-1}$ are learned by backprop (while still being invertible). The determinants need to be computed analytically.

# Designing a Transformation

Assume: $y = (y_1, y_2, \ldots, y_J)$. Then factorise the density according to the chain rule.

$$\log p(y|\theta) = \sum_{j=1}^{J} \log p(y_j|y_{<j}, \theta)$$

We will specify the bijection one coordinate at a time. This just makes it easier to specify a bijection (it's easier to work on $\mathbb{R}$ than $\mathbb{R}^D$).

By having $\epsilon_j$ depend on $y_j$ and $y_{<j}$, but not on $y_{>j}$, we will obtain a convenient Jacobian matrix.

# Designing a Transformation

Assume: $y = (y_1, y_2, \ldots, y_J)$. Then factorise the density according to the chain rule.

$$\log p(y|\theta) = \sum_{j=1}^{J} \log p(y_j|y_{<j}, \theta)$$

Next assume an invertible mapping $\epsilon_j = h^{-1}(y_j)$, the mapping is parameterised using $y_{<j}$.

We will specify the bijection one coordinate at a time. This just makes it easier to specify a bijection (it's easier to work on $\mathbb{R}$ than $\mathbb{R}^D$).

By having $\epsilon_j$ depend on $y_j$ and $y_{<j}$, but not on $y_{>j}$, we will obtain a convenient Jacobian matrix.

# Designing a Transformation (first step of the flow)

We use an NN $g_\theta^{(1)}$ to predict the parameters of the first transformation: $\begin{bmatrix}\mu_j & \sigma_j\end{bmatrix} = g_\theta^{(1)}(y_{<j})$. Then we apply the first transformation.

$$y_j^{(1)} = [h_1^{-1}(y)]_j \quad = \frac{y_j - \mu_1(y_{<j})}{\sigma_1(y_{<j})}$$

A simple invertible transformation: the affine function (with non-zero slope).

$$a = \sigma b + \mu \tag{19}$$

$$b = \frac{a - \mu}{\sigma} \tag{20}$$

We typically constrain $\sigma > 0$. For stability, we may use $\sigma \in (0, 1)$.

# Designing a Transformation (first step of the flow)

We use an NN $g_\theta^{(1)}$ to predict the parameters of the first transformation: $\begin{bmatrix}\mu_j & \sigma_j\end{bmatrix} = g_\theta^{(1)}(y_{<j})$. Then we apply the first transformation.

$$y_j^{(1)} = [h_1^{-1}(y)]_j = \frac{y_j - \mu_1(y_{<j})}{\sigma_1(y_{<j})}$$

$$y^{(1)} = h_1^{-1}(y) = \frac{y - \mu_1}{\sigma_1}$$

A simple invertible transformation: the affine function (with non-zero slope).

$$a = \sigma b + \mu \tag{19}$$

$$b = \frac{a - \mu}{\sigma} \tag{20}$$

We typically constrain $\sigma > 0$. For stability, we may use $\sigma \in (0, 1)$.

# Designing a Transformation (first step of the flow)

We use an NN $g_\theta^{(1)}$ to predict the parameters of the first transformation: $\begin{bmatrix}\mu_j & \sigma_j\end{bmatrix} = g_\theta^{(1)}(y_{<j})$. Then we apply the first transformation.

$$y_j^{(1)} = [h_1^{-1}(y)]_j = \frac{y_j - \mu_1(y_{<j})}{\sigma_1(y_{<j})}$$
$$y^{(1)} = h_1^{-1}(y) = \frac{y - \mu_1}{\sigma_1}$$

The Jacobian is

$$J_{h_1^{-1}}(y) =$$

A simple invertible transformation: the affine function (with non-zero slope).

$$a = \sigma b + \mu \tag{19}$$
$$b = \frac{a - \mu}{\sigma} \tag{20}$$

We typically constrain $\sigma > 0$. For stability, we may use $\sigma \in (0, 1)$.

# Designing a Transformation (first step of the flow)

We use an NN $g_\theta^{(1)}$ to predict the parameters of the first transformation: $\begin{bmatrix}\mu_j & \sigma_j\end{bmatrix} = g_\theta^{(1)}(y_{<j})$. Then we apply the first transformation.

$$y_j^{(1)} = [h_1^{-1}(y)]_j = \frac{y_j - \mu_1(y_{<j})}{\sigma_1(y_{<j})}$$
$$y^{(1)} = h_1^{-1}(y) = \frac{y - \mu_1}{\sigma_1}$$

The Jacobian is

$$J_{h_1^{-1}}(y) = \mathrm{I}\,\sigma_1^{-1} + J_{\frac{-\mu_1}{\sigma_1}}(y)$$

A simple invertible transformation: the affine function (with non-zero slope).

$$a = \sigma b + \mu \tag{21}$$
$$b = \frac{a - \mu}{\sigma} \tag{22}$$

We typically constrain $\sigma > 0$. For stability, we may use $\sigma \in (0, 1)$.

# Designing a Transformation

Define $\alpha_{i,j} = \frac{d}{dy_i} \frac{-\mu_j}{\sigma_j}$.

$$J_{h_K^{-1}}(y) = \mathbf{I}\,\sigma^{-1} + J_{\frac{-\mu}{\sigma}}(y) =$$

# Designing a Transformation

Define $\alpha_{i,j} = \frac{d}{dy_i} \frac{-\mu_j}{\sigma_j}$.

$$J_{h_K^{-1}}(y) = \mathrm{I}\,\sigma^{-1} + J_{\frac{-\mu}{\sigma}}(y) =$$

$$\begin{bmatrix} \sigma_{1,1}^{-1} & 0 & \cdots & 0 & 0 \\ 0 & \sigma_{2,2}^{-1} & \cdots & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & \sigma_{m,m}^{-1} \end{bmatrix}$$

# Designing a Transformation

Define $\alpha_{i,j} = \frac{d}{dy_i} \frac{-\mu_j}{\sigma_j}$.

$$J_{h_K^{-1}}(y) = \mathrm{I}\,\sigma^{-1} + J_{\frac{-\mu}{\sigma}}(y) =$$

$$\begin{bmatrix} \sigma_{1,1}^{-1} & 0 & \cdots & 0 & 0 \\ 0 & \sigma_{2,2}^{-1} & \cdots & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & \sigma_{m,m}^{-1} \end{bmatrix} + \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ \alpha_{2,1} & 0 & \cdots & 0 & 0 \\ \alpha_{3,1} & \alpha_{3,2} & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ \alpha_{m,1} & \alpha_{m,2} & \cdots & \alpha_{m,m-1} & 0 \end{bmatrix}$$

# Designing an efficient transformation


The solid lines are simple linear functions, whose slope and intercept are autoregressively predicted by an NN for each step of the flow (the dashes lines show these dependencies).

$$y_j^{(1)} = [h_1^{-1}(y)]_j \quad = \frac{y_j - \mu_1(y_{<j})}{\sigma_1(y_{<j})}$$

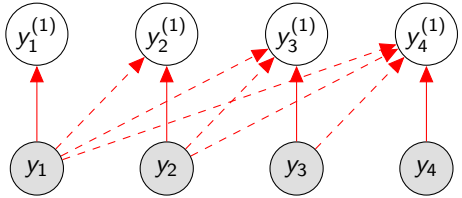
$$\epsilon_j = [h_2^{-1}(y^{(1)})]_j \quad = \frac{y_j^{(1)} - \mu_2(y_{<j}^{(1)})}{\sigma_2(y_{<j}^{(1)})}$$

**For fixed $J$, we can use a MADE (Germain et al., 2015b)**

An autoregressive network that takes constant time (it's implemented using an FFNN). Its connectivity matrix is lower-triangular.

$$\begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix}$$

# Designing an efficient transformation




The solid lines are simple linear functions, whose slope and intercept are autoregressively predicted by an NN for each step of the flow (the dashes lines show these dependencies).

$$y_j^{(1)} = [h_1^{-1}(y)]_j \quad = \frac{y_j - \mu_1(y_{<j})}{\sigma_1(y_{<j})}$$

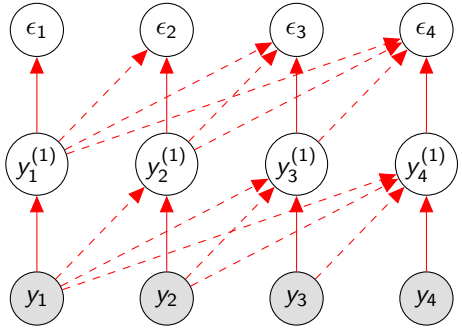
$$\epsilon_j = [h_2^{-1}(y^{(1)})]_j \quad = \frac{y_j^{(1)} - \mu_2(y_{<j}^{(1)})}{\sigma_2(y_{<j}^{(1)})}$$

For fixed $J$, we can use a MADE (Germain et al., 2015b)

An autoregressive network that takes constant time (it's implemented using an FFNN). Its connectivity matrix is lower-triangular.

$$\begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix}$$

# Designing an efficient transformation




The solid lines are simple linear functions, whose slope and intercept are autoregressively predicted by an NN for each step of the flow (the dashes lines show these dependencies).

$$y_j^{(1)} = [h_1^{-1}(y)]_j \quad = \frac{y_j - \mu_1(y_{<j})}{\sigma_1(y_{<j})}$$

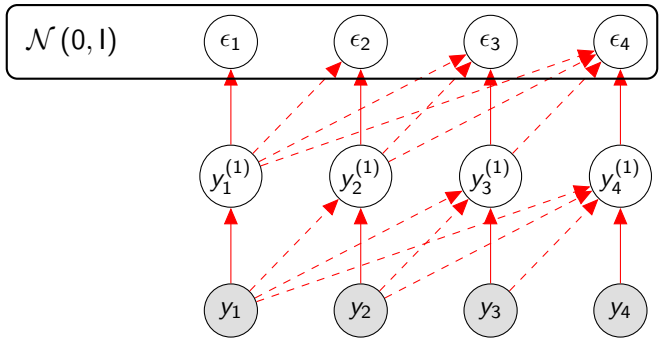
$$\epsilon_j = [h_2^{-1}(y^{(1)})]_j \quad = \frac{y_j^{(1)} - \mu_2(y_{<j}^{(1)})}{\sigma_2(y_{<j}^{(1)})}$$

**For fixed $J$, we can use a MADE (Germain et al., 2015b)**

An autoregressive network that takes constant time (it's implemented using an FFNN). Its connectivity matrix is lower-triangular.

$$\begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix}$$

# Designing an efficient transformation




The solid lines are simple linear functions, whose slope and intercept are autoregressively predicted by an NN for each step of the flow (the dashes lines show these dependencies).

$$y_j^{(1)} = [h_1^{-1}(y)]_j \quad = \frac{y_j - \mu_1(y_{<j})}{\sigma_1(y_{<j})}$$

$$\epsilon_j = [h_2^{-1}(y^{(1)})]_j \quad = \frac{y_j^{(1)} - \mu_2(y_{<j}^{(1)})}{\sigma_2(y_{<j}^{(1)})}$$

**For fixed $J$, we can use a MADE (Germain et al., 2015b)**

An autoregressive network that takes constant time (it's implemented using an FFNN). Its connectivity matrix is lower-triangular.

$$\begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix}$$

# Designing a Transformation

## Simple Jacobian Determinant

$$\left|\det J_{h_1^{-1}}(y)\right| = \prod_{j=1}^{J} \sigma_j^{-1}$$

# Designing a Transformation

## Simple Jacobian Determinant

$$\left|\det J_{h_1^{-1}}(y)\right| = \prod_{j=1}^{J} \sigma_j^{-1}$$

In practice we work with the log-likelihood.

$$\log \left|\det J_{h_1^{-1}}(y)\right| = -\sum_{j=1}^{J} \log \sigma_j$$

# 2-step Flow

$$
\begin{aligned}
p_Y(y) &= p_E(\epsilon)\left|\det J_{h_2^{-1}}(y^{(1)})\right|\left|\det J_{h_1^{-1}}(y)\right| \\
&= p_E(h_2^{-1}(h_1^{-1}(y)))\left|\det J_{h_2^{-1}}(h_1^{-1}(y))\right|\left|\det J_{h_1^{-1}}(y)\right|
\end{aligned}
$$

# 2-step Flow

$$
\begin{aligned}
p_Y(y) &= p_E(\epsilon)\left|\det J_{h_2^{-1}}(y^{(1)})\right|\left|\det J_{h_1^{-1}}(y)\right| \\
&= p_E(h_2^{-1}(h_1^{-1}(y)))\left|\det J_{h_2^{-1}}(h_1^{-1}(y))\right|\left|\det J_{h_1^{-1}}(y)\right|
\end{aligned}
$$

$$
\log p_Y(y) = \log p_E(h_2^{-1}(h_1^{-1}(y)))
$$

# 2-step Flow

$$
\begin{aligned}
p_Y(y) &= p_E(\epsilon)\left|\det J_{h_2^{-1}}(y^{(1)})\right|\left|\det J_{h_1^{-1}}(y)\right| \\
&= p_E(h_2^{-1}(h_1^{-1}(y)))\left|\det J_{h_2^{-1}}(h_1^{-1}(y))\right|\left|\det J_{h_1^{-1}}(y)\right| \\
\log p_Y(y) &= \log p_E(h_2^{-1}(h_1^{-1}(y))) - \sum_{j=1}^{J} \log \sigma_j^{(2)} - \sum_{j=1}^{J} \log \sigma_j^{(1)}
\end{aligned}
$$

# 2-step Flow

$$
\begin{aligned}
p_Y(y) &= p_E(\epsilon)\left|\det J_{h_2^{-1}}(y^{(1)})\right|\left|\det J_{h_1^{-1}}(y)\right| \\
&= p_E(h_2^{-1}(h_1^{-1}(y)))\left|\det J_{h_2^{-1}}(h_1^{-1}(y))\right|\left|\det J_{h_1^{-1}}(y)\right| \\
\log p_Y(y) &= \log p_E(h_2^{-1}(h_1^{-1}(y))) - \sum_{j=1}^{J} \log \sigma_j^{(2)} - \sum_{j=1}^{J} \log \sigma_j^{(1)}
\end{aligned}
$$

$$
y^{(1)} = h_1^{-1}(y) = \frac{y - \mu^{(1)}}{\sigma^{(1)}} \text{ where } \left[\mu^{(1)}, \sigma^{(1)}\right] = g^{(1)}(y)
$$

# 2-step Flow

$$
\begin{aligned}
p_Y(y) &= p_E(\epsilon)\left|\det J_{h_2^{-1}}(y^{(1)})\right|\left|\det J_{h_1^{-1}}(y)\right| \\
&= p_E(h_2^{-1}(h_1^{-1}(y)))\left|\det J_{h_2^{-1}}(h_1^{-1}(y))\right|\left|\det J_{h_1^{-1}}(y)\right| \\
\log p_Y(y) &= \log p_E(h_2^{-1}(h_1^{-1}(y))) - \sum_{j=1}^{J} \log \sigma_j^{(2)} - \sum_{j=1}^{J} \log \sigma_j^{(1)}
\end{aligned}
$$

$$
y^{(1)} = h_1^{-1}(y) = \frac{y - \mu^{(1)}}{\sigma^{(1)}} \text{ where } \left[\mu^{(1)}, \sigma^{(1)}\right] = g^{(1)}(y)
$$

$$
\epsilon = h_2^{-1}(y^{(1)}) = \frac{y^{(1)} - \mu^{(2)}}{\sigma^{(2)}} \text{ where } \left[\mu^{(2)}, \sigma^{(2)}\right] = g^{(2)}(y^{(1)})
$$

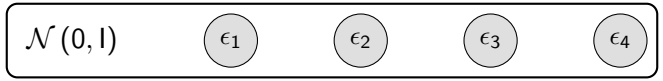
# Intermediate Summary

- NFs map transform complex distributions to simpler ones (or vice versa)
- Use in density estimation for complex distributions
- Jacobian needs to be carefully designed
- Sampling is slow because sequential

# Sampling From the Flow

Recall that $y_j^{(1)} = \frac{y_j - \mu^{(1)}(y_{<j})}{\sigma^{(1)}(y_{<j})}$ and $\epsilon_j = \frac{y_j^{(1)} - \mu^{(2)}(y_{<j}^{(1)})}{\sigma^{(2)}(y_{<j}^{(1)})}$

Hence, $y_j^{(1)} = \mu^2(y_{<j}^{(1)}) + \epsilon_j \sigma^{(2)}(y_{<j}^{(1)})$ and $y_j = \mu^{(1)}(y_{<j}) + y_j^{(1)} \sigma^{(1)}(y_{<j})$



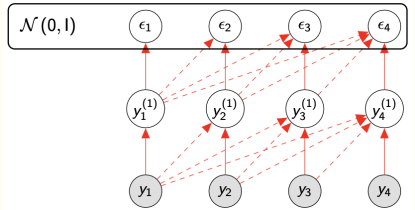



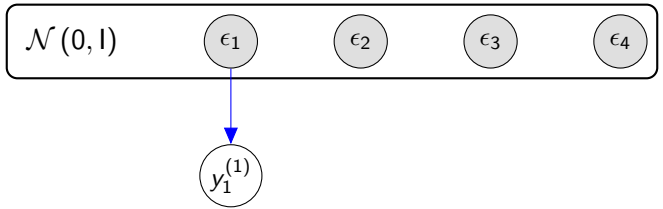

Figure: Reminder: autoregressive dependencies in the parameterisation (dashed lines). Given the dashed dependencies, the solid lines are invertible.

- We start with a base sample (all $J$ dimensions).

# Sampling From the Flow

Recall that $y_j^{(1)} = \frac{y_j - \mu^{(1)}(y_{<j})}{\sigma^{(1)}(y_{<j})}$ and $\epsilon_j = \frac{y_j^{(1)} - \mu^{(2)}(y_{<j}^{(1)})}{\sigma^{(2)}(y_{<j}^{(1)})}$

Hence, $y_j^{(1)} = \mu^2(y_{<j}^{(1)}) + \epsilon_j \sigma^{(2)}(y_{<j}^{(1)})$ and $y_j = \mu^{(1)}(y_{<j}) + y_j^{(1)} \sigma^{(1)}(y_{<j})$



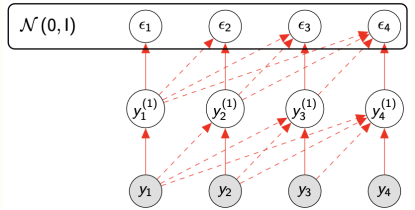



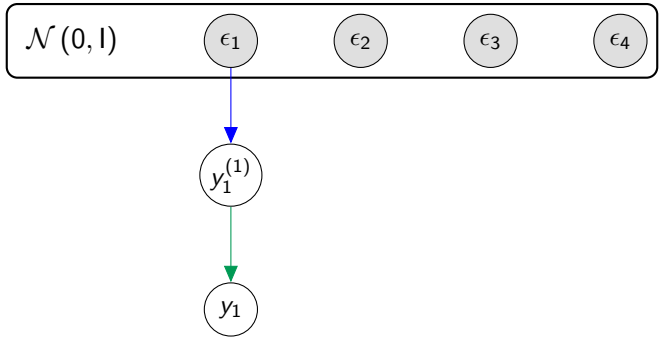

Figure: Reminder: autoregressive dependencies in the parameterisation (dashed lines). Given the dashed dependencies, the solid lines are invertible.

- We start with a base sample (all $J$ dimensions).
- And transform coordinates, one at a time. We invert the second step of the flow.

# Sampling From the Flow

Recall that $y_j^{(1)} = \frac{y_j - \mu^{(1)}(y_{<j})}{\sigma^{(1)}(y_{<j})}$ and $\epsilon_j = \frac{y_j^{(1)} - \mu^{(2)}(y_{<j}^{(1)})}{\sigma^{(2)}(y_{<j}^{(1)})}$

Hence, $y_j^{(1)} = \mu^2(y_{<j}^{(1)}) + \epsilon_j \sigma^{(2)}(y_{<j}^{(1)})$ and $y_j = \mu^{(1)}(y_{<j}) + y_j^{(1)} \sigma^{(1)}(y_{<j})$



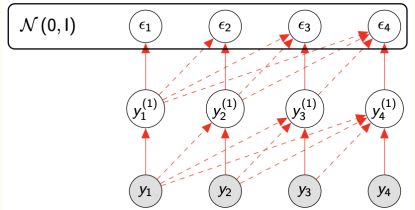



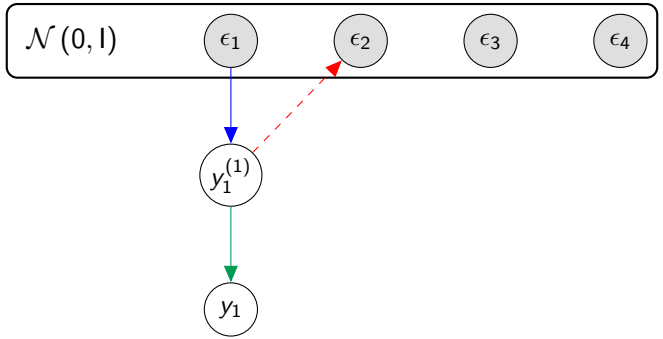

Figure: Reminder: autoregressive dependencies in the parameterisation (dashed lines). Given the dashed dependencies, the solid lines are invertible.

- We start with a base sample (all $J$ dimensions).
- And transform coordinates, one at a time. We invert the second step of the flow.
- Then the first.

# Sampling From the Flow

Recall that $y_j^{(1)} = \frac{y_j - \mu^{(1)}(y_{<j})}{\sigma^{(1)}(y_{<j})}$ and $\epsilon_j = \frac{y_j^{(1)} - \mu^{(2)}(y_{<j}^{(1)})}{\sigma^{(2)}(y_{<j}^{(1)})}$

Hence, $y_j^{(1)} = \mu^2(y_{<j}^{(1)}) + \epsilon_j \sigma^{(2)}(y_{<j}^{(1)})$ and $y_j = \mu^{(1)}(y_{<j}) + y_j^{(1)}\sigma^{(1)}(y_{<j})$



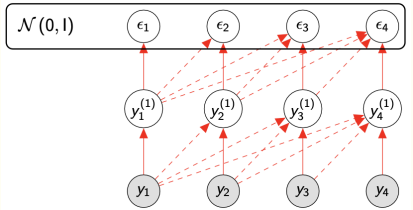



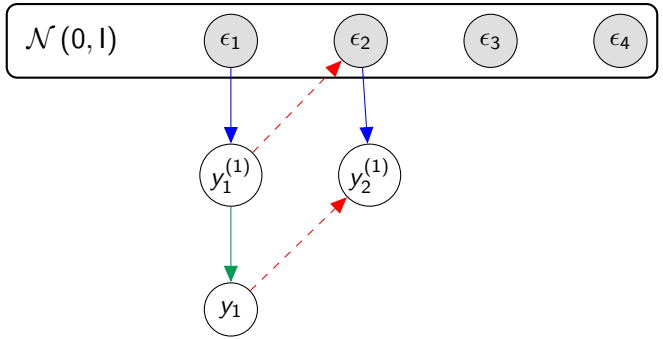

Figure: Reminder: autoregressive dependencies in the parameterisation (dashed lines). Given the dashed dependencies, the solid lines are invertible.

- We start with a base sample (all $J$ dimensions).
- And transform coordinates, one at a time. We invert the second step of the flow.
- Then the first.
- After that, we can run the MADE (with input $(y_1, 0, 0, 0)^\top$, 'dummy' values for $y_{>1}$) to obtain the mus and sigmas needed to invert the second coordinate.

# Sampling From the Flow

Recall that $y_j^{(1)} = \frac{y_j - \mu^{(1)}(y_{<j})}{\sigma^{(1)}(y_{<j})}$ and $\epsilon_j = \frac{y_j^{(1)} - \mu^{(2)}(y_{<j}^{(1)})}{\sigma^{(2)}(y_{<j}^{(1)})}$

Hence, $y_j^{(1)} = \mu^2(y_{<j}^{(1)}) + \epsilon_j \sigma^{(2)}(y_{<j}^{(1)})$ and $y_j = \mu^{(1)}(y_{<j}) + y_j^{(1)}\sigma^{(1)}(y_{<j})$



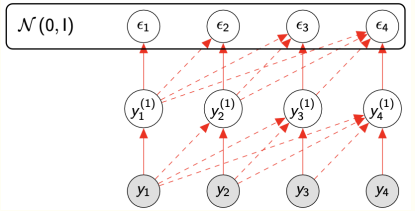



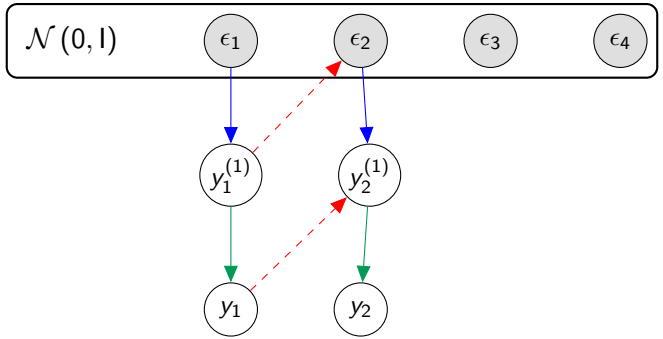

Figure: Reminder: autoregressive dependencies in the parameterisation (dashed lines). Given the dashed dependencies, the solid lines are invertible.

- We start with a base sample (all $J$ dimensions).
- And transform coordinates, one at a time. We invert the second step of the flow.
- Then the first.
- After that, we can run the MADE (with input $(y_1, 0, 0, 0)^\top$, 'dummy' values for $y_{>1}$) to obtain the mus and sigmas needed to invert the second coordinate.
- We then invert it

# Sampling From the Flow

Recall that $y_j^{(1)} = \frac{y_j - \mu^{(1)}(y_{<j})}{\sigma^{(1)}(y_{<j})}$ and $\epsilon_j = \frac{y_j^{(1)} - \mu^{(2)}(y_{<j}^{(1)})}{\sigma^{(2)}(y_{<j}^{(1)})}$

Hence, $y_j^{(1)} = \mu^2(y_{<j}^{(1)}) + \epsilon_j \sigma^{(2)}(y_{<j}^{(1)})$ and $y_j = \mu^{(1)}(y_{<j}) + y_j^{(1)} \sigma^{(1)}(y_{<j})$



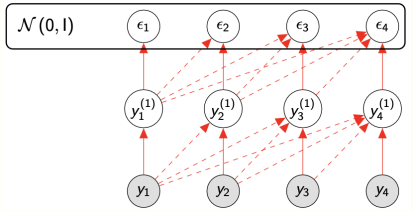



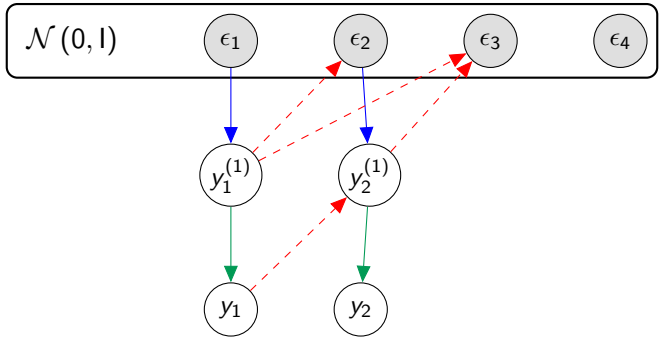

Figure: Reminder: autoregressive dependencies in the parameterisation (dashed lines). Given the dashed dependencies, the solid lines are invertible.

- We start with a base sample (all $J$ dimensions).
- And transform coordinates, one at a time. We invert the second step of the flow.
- Then the first.
- After that, we can run the MADE (with input $(y_1, 0, 0, 0)^\top$, 'dummy' values for $y_{>1}$) to obtain the mus and sigmas needed to invert the second coordinate.
- We then invert it (one step at a time).

# Sampling From the Flow

Recall that $y_j^{(1)} = \frac{y_j - \mu^{(1)}(y_{<j})}{\sigma^{(1)}(y_{<j})}$ and $\epsilon_j = \frac{y_j^{(1)} - \mu^{(2)}(y_{<j}^{(1)})}{\sigma^{(2)}(y_{<j}^{(1)})}$

Hence, $y_j^{(1)} = \mu^2(y_{<j}^{(1)}) + \epsilon_j \sigma^{(2)}(y_{<j}^{(1)})$ and $y_j = \mu^{(1)}(y_{<j}) + y_j^{(1)} \sigma^{(1)}(y_{<j})$



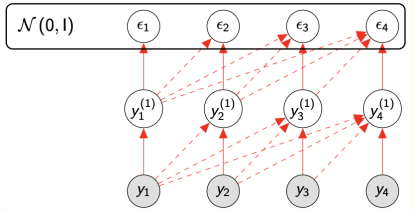



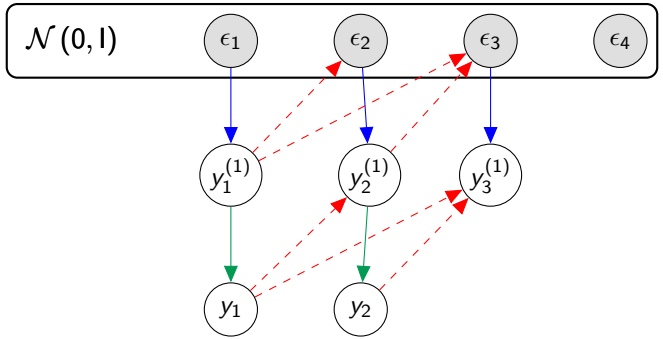

Figure: Reminder: autoregressive dependencies in the parameterisation (dashed lines). Given the dashed dependencies, the solid lines are invertible.

- We start with a base sample (all $J$ dimensions).
- And transform coordinates, one at a time. We invert the second step of the flow.
- Then the first.
- After that, we can run the MADE (with input $(y_1, 0, 0, 0)^\top$, 'dummy' values for $y_{>1}$) to obtain the mus and sigmas needed to invert the second coordinate.
- We then invert it (one step at a time).
- and repeat till we are done.

# Sampling From the Flow

Recall that $y_j^{(1)} = \frac{y_j - \mu^{(1)}(y_{<j})}{\sigma^{(1)}(y_{<j})}$ and $\epsilon_j = \frac{y_j^{(1)} - \mu^{(2)}(y_{<j}^{(1)})}{\sigma^{(2)}(y_{<j}^{(1)})}$

Hence, $y_j^{(1)} = \mu^2(y_{<j}^{(1)}) + \epsilon_j \sigma^{(2)}(y_{<j}^{(1)})$ and $y_j = \mu^{(1)}(y_{<j}) + y_j^{(1)}\sigma^{(1)}(y_{<j})$



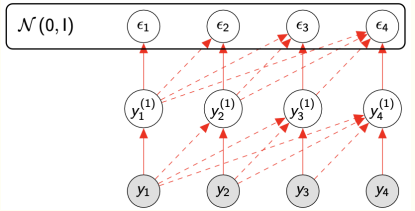



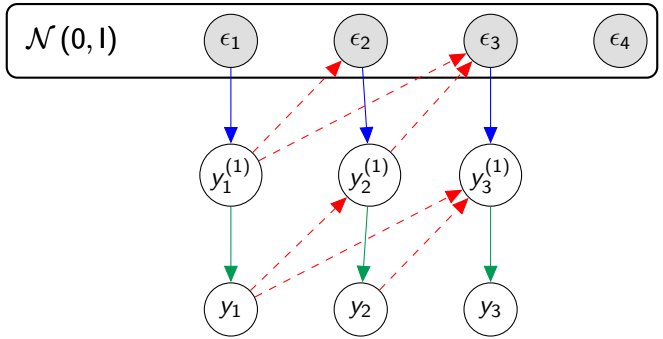

Figure: Reminder: autoregressive dependencies in the parameterisation (dashed lines). Given the dashed dependencies, the solid lines are invertible.

- We start with a base sample (all $J$ dimensions).
- And transform coordinates, one at a time. We invert the second step of the flow.
- Then the first.
- After that, we can run the MADE (with input $(y_1, 0, 0, 0)^\top$, 'dummy' values for $y_{>1}$) to obtain the mus and sigmas needed to invert the second coordinate.
- We then invert it (one step at a time).
- and repeat till we are done.

# Sampling From the Flow

Recall that $y_j^{(1)} = \frac{y_j - \mu^{(1)}(y_{<j})}{\sigma^{(1)}(y_{<j})}$ and $\epsilon_j = \frac{y_j^{(1)} - \mu^{(2)}(y_{<j}^{(1)})}{\sigma^{(2)}(y_{<j}^{(1)})}$

Hence, $y_j^{(1)} = \mu^2(y_{<j}^{(1)}) + \epsilon_j \sigma^{(2)}(y_{<j}^{(1)})$ and $y_j = \mu^{(1)}(y_{<j}) + y_j^{(1)}\sigma^{(1)}(y_{<j})$



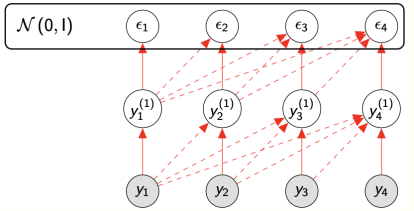



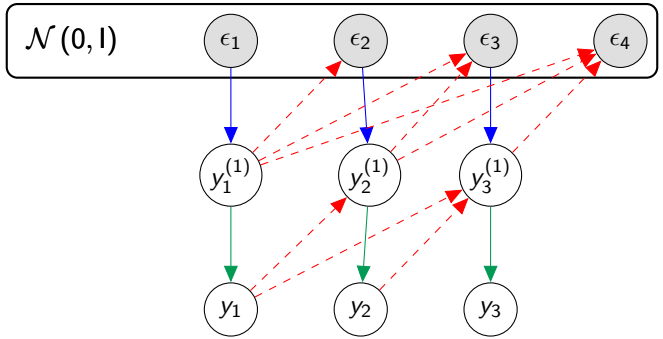

Figure: Reminder: autoregressive dependencies in the parameterisation (dashed lines). Given the dashed dependencies, the solid lines are invertible.

- We start with a base sample (all $J$ dimensions).
- And transform coordinates, one at a time. We invert the second step of the flow.
- Then the first.
- After that, we can run the MADE (with input $(y_1, 0, 0, 0)^\top$, 'dummy' values for $y_{>1}$) to obtain the mus and sigmas needed to invert the second coordinate.
- We then invert it (one step at a time).
- and repeat till we are done.

# Sampling From the Flow

Recall that $y_j^{(1)} = \frac{y_j - \mu^{(1)}(y_{<j})}{\sigma^{(1)}(y_{<j})}$ and $\epsilon_j = \frac{y_j^{(1)} - \mu^{(2)}(y_{<j}^{(1)})}{\sigma^{(2)}(y_{<j}^{(1)})}$

Hence, $y_j^{(1)} = \mu^2(y_{<j}^{(1)}) + \epsilon_j \sigma^{(2)}(y_{<j}^{(1)})$ and $y_j = \mu^{(1)}(y_{<j}) + y_j^{(1)}\sigma^{(1)}(y_{<j})$



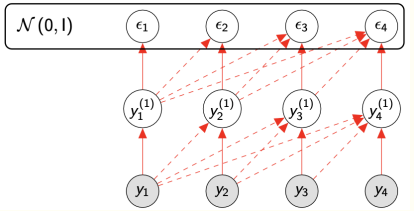



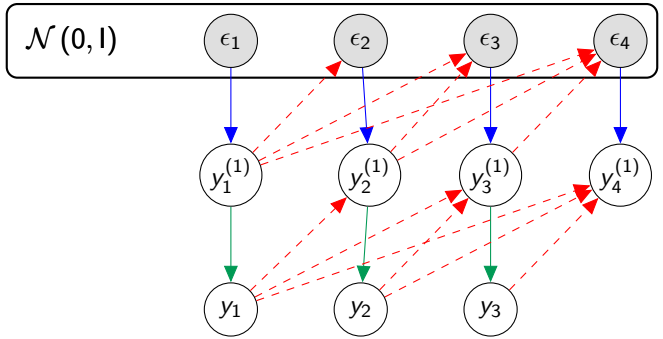

Figure: Reminder: autoregressive dependencies in the parameterisation (dashed lines). Given the dashed dependencies, the solid lines are invertible.

- We start with a base sample (all $J$ dimensions).
- And transform coordinates, one at a time. We invert the second step of the flow.
- Then the first.
- After that, we can run the MADE (with input $(y_1, 0, 0, 0)^\top$, 'dummy' values for $y_{>1}$) to obtain the mus and sigmas needed to invert the second coordinate.
- We then invert it (one step at a time).
- and repeat till we are done.

# Sampling From the Flow

Recall that $y_j^{(1)} = \frac{y_j - \mu^{(1)}(y_{<j})}{\sigma^{(1)}(y_{<j})}$ and $\epsilon_j = \frac{y_j^{(1)} - \mu^{(2)}(y_{<j}^{(1)})}{\sigma^{(2)}(y_{<j}^{(1)})}$

Hence, $y_j^{(1)} = \mu^2(y_{<j}^{(1)}) + \epsilon_j \sigma^{(2)}(y_{<j}^{(1)})$ and $y_j = \mu^{(1)}(y_{<j}) + y_j^{(1)}\sigma^{(1)}(y_{<j})$



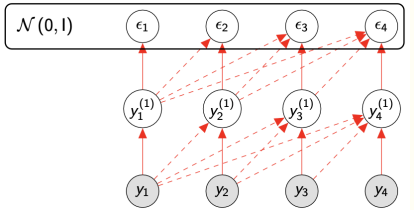



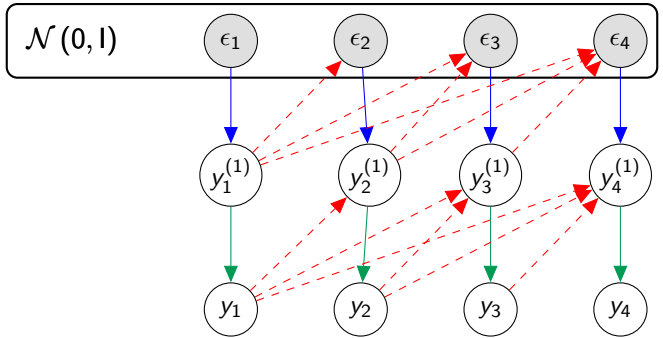

Figure: Reminder: autoregressive dependencies in the parameterisation (dashed lines). Given the dashed dependencies, the solid lines are invertible.

- We start with a base sample (all $J$ dimensions).
- And transform coordinates, one at a time. We invert the second step of the flow.
- Then the first.
- After that, we can run the MADE (with input $(y_1, 0, 0, 0)^\top$, 'dummy' values for $y_{>1}$) to obtain the mus and sigmas needed to invert the second coordinate.
- We then invert it (one step at a time).
- and repeat till we are done.

# Sampling From the Flow

Recall that $y_j^{(1)} = \frac{y_j - \mu^{(1)}(y_{<j})}{\sigma^{(1)}(y_{<j})}$ and $\epsilon_j = \frac{y_j^{(1)} - \mu^{(2)}(y_{<j}^{(1)})}{\sigma^{(2)}(y_{<j}^{(1)})}$

Hence, $y_j^{(1)} = \mu^2(y_{<j}^{(1)}) + \epsilon_j \sigma^{(2)}(y_{<j}^{(1)})$ and $y_j = \mu^{(1)}(y_{<j}) + y_j^{(1)}\sigma^{(1)}(y_{<j})$



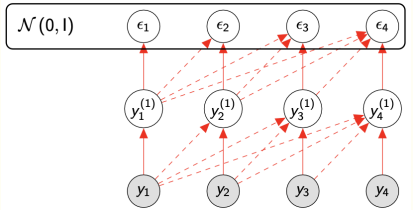




Figure: Reminder: autoregressive dependencies in the parameterisation (dashed lines). Given the dashed dependencies, the solid lines are invertible.

- We start with a base sample (all $J$ dimensions).
- And transform coordinates, one at a time. We invert the second step of the flow.
- Then the first.
- After that, we can run the MADE (with input $(y_1, 0, 0, 0)^\top$, 'dummy' values for $y_{>1}$) to obtain the mus and sigmas needed to invert the second coordinate.
- We then invert it (one step at a time).
- and repeat till we are done.

# Summary

- Given a deep enough flow, NFs model arbitrary continuous distributions
- They allow for density computation
- Need to have simple Jacobian determinants
- Depending on direction, one of the two operations (sampling or density computation) is slower (sequential)
- If one of the two computation graphs (for $h$ or $h^{-1}$) is not known, then one of the two operations becomes *very* difficult (if at all possible)

# Beyond

- We built flows using an affine transformation (with non-zero scale) because its is trivially invertible,

For variable length $J$, see (IAF; Kingma et al., 2016).

For an NFs that are universal pdf approximators, see (NAF and BNAF; Huang et al., 2018; De Cao et al., 2020).

# Beyond

- We built flows using an affine transformation (with non-zero scale) because its is trivially invertible,
- but we can construct other flows using other bijections, e.g. a permutation (volume preserving)

For variable length $J$, see (IAF; Kingma et al., 2016).

For an NFs that are universal pdf approximators, see (NAF and BNAF; Huang et al., 2018; De Cao et al., 2020).

# Beyond

- We built flows using an affine transformation (with non-zero scale) because its is trivially invertible,
- but we can construct other flows using other bijections, e.g. a permutation (volume preserving)
- or any strictly monotone function
  e.g. a neural network with positive weights and strictly monotone activations

For variable length $J$, see (IAF; Kingma et al., 2016).

For an NFs that are universal pdf approximators, see (NAF and BNAF; Huang et al., 2018; De Cao et al., 2020).

# Beyond

- We built flows using an affine transformation (with non-zero scale) because its is trivially invertible,
- but we can construct other flows using other bijections, e.g. a permutation (volume preserving)
- or any strictly monotone function
  e.g. a neural network with positive weights and strictly monotone activations
- also note that, we require invertibility (strict monotonicity), not **analytical** invertibility

For variable length $J$, see (IAF; Kingma et al., 2016).

For an NFs that are universal pdf approximators, see (NAF and BNAF; Huang et al., 2018; De Cao et al., 2020).

# References I

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the Ability and Limitations of Transformers to Recognize Formal Languages. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7096–7116, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.576. URL https://aclanthology.org/2020.emnlp-main.576.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

# References II

Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In Dekai Wu, Marine Carpuat, Xavier Carreras, and Eva Maria Vecchi, editors, *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/W14-4012. URL https://aclanthology.org/W14-4012.

Nicola De Cao, Wilker Aziz, and Ivan Titov. Block neural autoregressive flow. In Ryan P. Adams and Vibhav Gogate, editors, *UAI*, volume 115 of *Proceedings of machine learning research*, pages 1263–1273, Tel Aviv, Israel, July 2020. PMLR. URL http://proceedings.mlr.press/v115/de-cao20a.html. tex.pdf: http://proceedings.mlr.press/v115/de-cao20a/de-cao20a.pdf.

# References III

Li Du, Lucas Torroba Hennigen, Tiago Pimentel, Clara Meister, Jason Eisner, and Ryan Cotterell. A measure-theoretic characterization of tight language models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9744–9770, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.543. URL https://aclanthology.org/2023.acl-long.543.

Bryan Eikema and Wilker Aziz. Is MAP Decoding All You Need? The Inadequacy of the Mode in Neural Machine Translation. *arXiv:2005.10283 [cs]*, May 2020. URL http://arxiv.org/abs/2005.10283. arXiv: 2005.10283.

# References IV

Bryan Eikema and Wilker Aziz. Sampling-based approximations to minimum Bayes risk decoding for neural machine translation. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 10978–10993, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.754. URL https://aclanthology.org/2022.emnlp-main.754.

Jason Eisner. Inside-outside and forward-backward algorithms are just backprop (tutorial paper). In Kai-Wei Chang, Ming-Wei Chang, Alexander Rush, and Vivek Srikumar, editors, *Proceedings of the Workshop on Structured Prediction for NLP*, pages 1–17, Austin, TX, November 2016. Association for Computational Linguistics. doi: 10.18653/v1/W16-5901. URL https://aclanthology.org/W16-5901.

# References V

Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. Made: Masked autoencoder for distribution estimation. In *International conference on machine learning*, pages 881–889. PMLR, 2015a.

Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. Made: Masked autoencoder for distribution estimation. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 881–889, Lille, France, 07–09 Jul 2015b. PMLR. URL https://proceedings.mlr.press/v37/germain15.html.

# References VI

Elliott Gordon-Rodriguez, Gabriel Loaiza-Ganem, and John Cunningham. The continuous categorical: a novel simplex-valued exponential family. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 3637–3647. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/gordon-rodriguez20a.html.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Albert Gu, Karan Goel, and Christopher Re. Efficiently modeling long sequences with structured state spaces. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=uYLFoz1vlAC.

# References VII

Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171, 2020. doi: 10.1162/tacl_a_00306. URL https://aclanthology.org/2020.tacl-1.11.

Michael Hahn and Mark Rofin. Why are sensitive functions hard for transformers? *arXiv preprint arXiv:2402.09963*, 2024.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. Publisher: MIT Press.

Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville. Neural Autoregressive Flows. In *International Conference on Machine Learning*, pages 2078–2087. PMLR, July 2018. URL http://proceedings.mlr.press/v80/huang18d.html. ISSN: 2640-3498.

# References VIII

Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*, 2016.

Diederik Kingma, Tim Salimans, Ben Poole, and Jonathan Ho. Variational diffusion models. *Advances in neural information processing systems*, 34:21696–21707, 2021.

Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved Variational Inference with Inverse Autoregressive Flow. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 4743–4751. Curran Associates, Inc., 2016. URL http://papers.nips.cc/paper/6581-improved-variational-inference-with-inverse-autoregres pdf.

# References IX

André FT Martins, Marcos Treviso, António Farinhas, Pedro MQ Aguiar, Mário AT Figueiredo, Mathieu Blondel, and Vlad Niculae. Sparse continuous distributions and fenchel-young losses. *The Journal of Machine Learning Research*, 23(1):11728–11801, 2022.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, pages 1045–1048. Makuhari, 2010.

Grey Nearing, Deborah Cohen, Vusumuzi Dube, Martin Gauch, Oren Gilon, Shaun Harrigan, Avinatan Hassidim, Daniel Klotz, Frederik Kratzert, Asher Metzger, et al. Global prediction of extreme floods in ungauged watersheds. *Nature*, 627(8004):559–563, 2024.

# References X

Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International conference on machine learning*, pages 1530–1538. PMLR, 2015.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951. Publisher: JSTOR.

John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. In *Advances in neural information processing systems*, pages 3528–3536, 2015.

# References XI

Noah A. Smith. *Linguistic Structure Prediction*. Synthesis Lectures on Human Language Technologies. Morgan and Claypool, May 2011.

Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International conference on machine learning*, pages 2256–2265. PMLR, 2015.

Yang Song and Stefano Ermon. Generative modeling by estimating gradients of the data distribution. *Advances in neural information processing systems*, 32, 2019.

Yang Song, Sahaj Garg, Jiaxin Shi, and Stefano Ermon. Sliced score matching: A scalable approach to density and score estimation. In *Uncertainty in Artificial Intelligence*, pages 574–584. PMLR, 2020.

# References XII

Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. Transformers as recognizers of formal languages: A survey on expressivity. *arXiv preprint arXiv:2311.00208*, 2023.

Benigno Uria, Iain Murray, and Hugo Larochelle. A deep and tractable density estimator. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 467–475, Bejing, China, 22–24 Jun 2014. PMLR. URL https://proceedings.mlr.press/v32/uria14.html.

# References XIII

Aaron van den Oord, Nal Kalchbrenner, Lasse Espeholt, koray kavukcuoglu, Oriol Vinyals, and Alex Graves. Conditional image generation with pixelcnn decoders. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper_files/paper/2016/file/b1301141feffabac455e1f90a7de2054-Paper.pdf.

Aaron Van den Oord, Nal Kalchbrenner, Lasse Espeholt, Oriol Vinyals, Alex Graves, et al. Conditional image generation with pixelcnn decoders. *Advances in neural information processing systems*, 29, 2016.

# References XIV

Aäron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1747–1756, New York, New York, USA, 20–22 Jun 2016. PMLR. URL https://proceedings.mlr.press/v48/oord16.html.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Pascal Vincent. A connection between score matching and denoising autoencoders. *Neural computation*, 23(7):1661–1674, 2011.