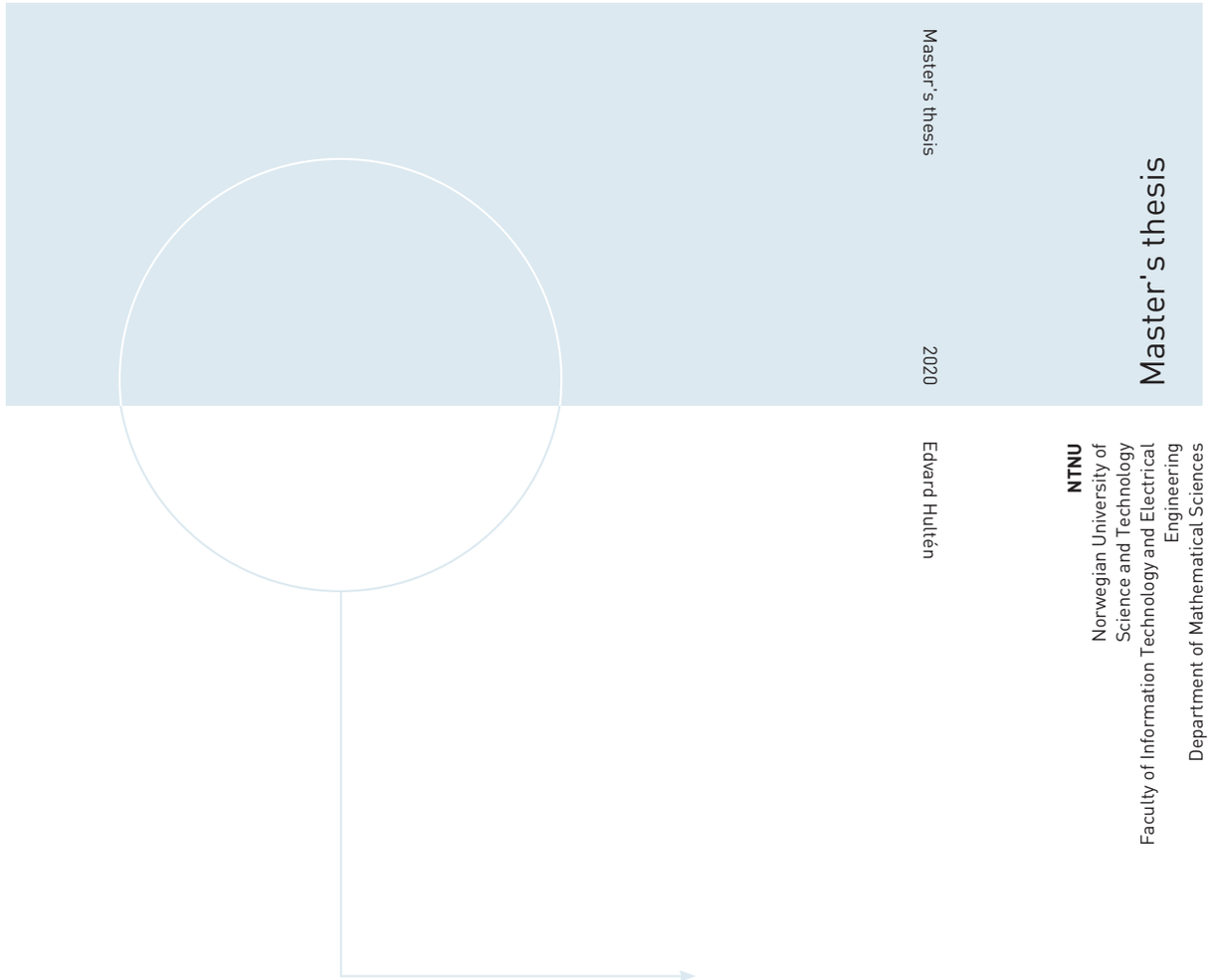


Edvard Hultén

Autoregressive Normalising Flows for Density Estimation and Variational Inference

A proper introduction and a novel flow

February 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Autoregressive Normalising Flows for Density Estimation and Variational Inference

A proper introduction and a novel flow

Edvard Hultén

Industrial Mathematics
Submission date: February 2020
Supervisor: Gunnar Taraldsen

Norwegian University of Science and Technology
Department of Mathematical Sciences

Til mamma og pappa.

Summary

In this thesis, we present a class of models called *normalising flows*. This class of models utilises the flexibility and computational advantages offered by the deep learning paradigm to define a general framework for modelling flexible, high-dimensional probability densities. We motivate the use of normalising flows by arguing that modelling flexible densities has uses within a wide range of problems in statistics and machine learning, but concentrate our focus on normalising flows for density estimation and approximate Bayesian inference. As a part of the introduction to normalising flows, we conduct a set of experiments with the *planar flow* to help the reader build intuition about normalising flows.

Further, we aim to give a clear exposition of the field of *autoregressive normalising flows*, which combines classical autoregressive modelling and modern normalising flows. This has been a highly influential class of models in the literature up until now. We provide a coherent presentation of the necessary concepts, and fill in some crucial details about the *Gaussian MADE* that are often found lacking elsewhere in the literature. Three different autoregressive flows, the *inverse autoregressive flow (IAF)*, *the masked autoregressive flow (MAF)*, and *Real NVP*, are presented and compared, highlighting their relative strengths and weaknesses.

Heavily inspired by two of the previously presented autoregressive flows, the masked autoregressive flow and Real NVP, we propose a novel flow for density estimation, which we call the *hybrid autoregressive flow*. We conduct a variety of experiments with MADE, Real NVP, and MAF, and successfully reproduce results from seminal papers in the literature, before we put the novel hybrid autoregressive flow to the test to compare it to existing models in a standardised experimental setting.

The new flow shows promising initial performance, outperforming its competitors on one of the density estimation benchmarks, but more empirical evidence is needed in order to draw any conclusions about the hybrid autoregressive flow. Finally, we summarise the thesis, before we discuss how to proceed the investigations of the hybrid autoregressive flows, and point out some directions for the future research on normalising flows in general.

Sammendrag

I denne oppgaven presenterer vi en klasse med modeller kalt "normalising flows". Dette er en klasse med modeller som drar nytte av fleksibiliteten og de beregningsmessige fordelene som tilbys av det moderne dyp lærings-paradigmet, og bruker det til å definere et generelt rammeverk for modellering av fleksible og høy-dimensjonale sannsynlighetstettheter. Vi motiverer bruken av normalising flows gjennom å argumentere med at modellering av fleksible sannsynlighetsfordelinger har anvendelser innenfor et bredt spekter av statstikk- og maskinlærings-problemer. Vi retter fokuset vårt spesielt mot tetthetsestimering og tilnærmet Bayesisk inferens. Som en del av introduksjonen til normalising flows, gjennomfører vi et sett med eksperimenter med en "planar flow", med mål om å hjelpe leseren med å bygge intuisjon for normalising flows.

Videre gir vi en tydelig innføring i feltet *autoregressive normalising flows*, som kombinerer klassisk autoregressiv modellering med moderne normalising flow. Dette har lenge vært en av de mest innflytelsesrike klassene med modeller i litteraturen om normalising flows. Vi gir en sammenhengende presentasjon av de nødvendige konseptene for å sette seg inn i denne litteraturen, inkludert å fylle inn noen detaljer om *Gaussisk MADE* som ser ut til å mangle i den øvrige litteraturen. Videre presenterer vi tre autoregressive flows: *inverse autoregressive flow (IAF)*, *masked autoregressive flow (MAF)*, og *Real NVP*, og sammenligner dem, med vekt på deres relative styrker og svakheter.

Kraftig inspirert av to av de presenterte modellene, masked autoregressive flow og Real NVP, foreslår vi en ny flow som egner seg til tetthetsestimering og gir den navnet *hybrid autoregressive flow*. Vi gjennomfører en rekke eksperimenter med modellene MADE, Real NVP, og MAF, og lykkes med å gjenskape resultater fra flere av de mest innflytelsesrike artiklene i litteraturen. Til slutt gjennomfører vi eksperimenter med vår foreslåtte flow, og sammenligner den med eksisterende modeller i et standardisert eksperimentelt oppsett.

Den nye flowen viser lovende ytelse i disse innledende eksperimentene, og gjør det bedre enn de andre modellene i et av eksperimentene. Mer empirisk materiale er nødvendig for å kunne trekke noen sikre konklusjoner om den foreslåtte modellen. Avslutningsvis oppsummerer vi oppgaven og diskuterer hvordan hybrid autoregressive flow bør testes videre. Helt til slutt peker vi ut noen mulige retninger for fremtidig forskning på normalising flows.

Preface

This thesis was written as the final part of my master's studies in *Industrial mathematics*, and concludes my studies at the Norwegian University of Science and Technology (NTNU). Studying at NTNU, including my year abroad at Nanyang Technological University in Singapore, has been the most rewarding experience of my life thus far, both academically and personally.

I want to thank my supervisor, Professor Gunnar Taraldsen, for all our meetings, and for giving me the opportunity to write about a topic that I find truly exciting. Normalising flows really are at the intersection of two of my primary academic interests; statistics and deep learning, and I have thoroughly enjoyed learning about this class of models and writing this thesis.

Finally and most importantly, I want to thank my family for their everlasting and unconditional support throughout the years, and my fellow students for making my time in Trondheim so special. Until next time!

Trondheim, February 2020
Edvard Hultén

Contents

Chapter 1

Introduction

In this thesis, we investigate a family of models called *normalising flows*, and some of their applications within statistics and machine learning. A normalising flow is at its core a model that can perform two simple operations; density evaluation and sampling. Use-cases can therefore be found in any probabilistic model that requires either of these two operations. A normalising flow, often only referred to as a *flow*, transforms a simple base density into a more complex density, and the parameter values defining the transformation are learned from data. In particular, we will focus on normalising flows for *density estimation*, and normalising flows for *variational inference*.

While normalising flows offer a powerful framework for modelling complex densities, but the basic principle of normalising flows is strikingly simple:

i take some simple density, typically an isotropic Gaussian

ii transform it using a composition of differentiable and bijective transformations.

The initial density "flows" through a sequence of transformations, and at the end of the flow we obtain a valid, *normalised* probability density. We are able to sample from, and evaluate the density of samples under the transformed density, due to the *change of variables formula*. This formula is taught in most introductory statistics courses, but the secret to why this seemingly rudimentary procedure is so useful, lies in the way the transformations are designed, and in the power of compositionality.

A large body of the research on normalising flows revolves is about how to design these transformations to achieve a flow with desirable properties. This thesis is mainly concerned with the class *autoregressive flows*, which combine autoregressive modelling and normalising flows to obtain transformations that are both highly flexible and tractable. The presented flows have applications within density estimation, variational inference, and generative modelling. Normalising flows were first proposed for density estimation by Tabak and Turner (2013), but were first popularised in a deep learning context by Rezende and Mohamed (2015) as a way to parameterise flexible approximate posterior densities for use in variational inference.

In the following years, there has been an emergence of normalising flows benefiting of neural networks, with successful applications in density estimation (Papamakarios et al., 2017; Dinh et al., 2014, 2017), variational inference (Kingma et al., 2016; van den Berg et al., 2018), generative modelling of images (Dinh et al., 2017; Kingma and Dhariwal, 2018) and audio (van den Oord et al., 2017). Applications to more traditional statistical problems include parameterising the auxiliary distribution in importance sampling (Müller et al., 2018), parameterising the proposal distribution in rejection sampling (Bauer and Mnih, 2019), and reparameterising the target distribution in MCMC sampling to make it more well-behaved (Hoffman et al., 2019).

Despite the vital role played by neural networks in these models, this is not a thesis about deep learning per se. We use the framework offered by deep learning to tackle classical problems from statistics, such as density estimation and approximate Bayesian inference, and the neural networks are in this context merely a tool used to enhance the performance of already existing techniques by adding the flexibility and learning capacity of neural networks to problems that call for modelling flexible probability distributions.

1.1 Why normalising flows?

In the wake of the popularisation of normalising flows that followed (Rezende and Mohamed, 2015), normalising flows have received increasing attention from the broader machine learning research community. Last year, in 2019, the first workshop on *Invertible Neural Networks and Normalising Flows* was arranged at the International Conference of Machine Learning (ICML), which, along with NeurIPS, is considered the premier conference for machine learning research. Just before, and during the writing of this thesis, the first comprehensive review papers on normalising flows were published by Kobyzev et al. (2019) (August) and (Papamakarios et al., 2019) (December).

As exemplified in the previous section, the merits of normalising flows are many and diverse, but the applications have in common that good density estimates are imperative for them to be successful. In this thesis, we restrict ourselves to discuss normalising flows for density estimation, and normalising flows for variational inference. Density estimation in the broad sense includes generative modelling of high-dimensional data like images and audio, but this thesis is concerned with density estimation that is purely quantitative, following along the lines of (Papamakarios et al., 2017).

Fundamentally, both density estimation and variational inference are concerned with estimating a probability density function from data. In the following, we motivate why these fields are of interest, and how normalising flows can be used to progress each field.

Density estimation

The problem of *density estimation* is at the heart of statistics and traditional machine learning methods. Given a set of observed samples produced by some unknown, stationary process, we want use the observed data to estimate the density function of the process that generated them. An estimated density function can be used to evaluate the density of an arbitrary observation, but does also provide a description of the generated data. A good density estimator is thus useful for many downstream tasks, and has numerous applications. We list some of them below:

i Estimate densities from data for use in Bayesian inference, e.g., by learning suitable priors from large datasets in an unsupervised manner.

ii Using the likelihood of the training data as a the objective function when training machine learning models. Allows for optimising the objective of interest directly.

iii To achieve better compression. A good density estimate implies a small Kullback Leibler divergence between the estimated model and the true model, which in turn means that expected number of lost bits by using the approximating density decreases.

iv Model scoring using metrics like entropy or maximum likelihood, which assume that we can evaluate the density under our model.

v Given the right model, we can generate new data very cheaply. This is useful for example for generating estimators of high-dimensional integrals like expected values, or to generate new samples when for training the inflow of real data is limited.

Traditional methods for density estimation like mixture models and kernel density estimators are efficient for learning low-dimensional densities, but suffer severely from the curse of dimensionality when dealing with data in the high-dimensional regime. Lately, density estimation models parameterised by neural networks have been successfully applied to density estimation problems, achieving state of the art performance on a range of high-dimensional density estimation benchmarks.

Particularly successful are the neural density estimators that combine autoregressive density estimation with normalising flows. Harnessing the flexibility of neural networks to learn complex dependencies in high-dimensional data, we can design autoregressive transformations that are suitable as components of a normalising flow. By stacking several such transformations in sequence, we can increase the flexibility compared to a regular autoregressive model. By using neural networks to define the transformations in the flow, we also get to easily utilise the powerful capabilities offered by modern deep learning frameworks to optimise the models. This is explained in more detail in Section 1.4.

Variational inference

Another central topic in statistics, is the one of performing posterior inference in Bayesian models. The gold standard in posterior inference is Markov chain Monte Carlo (MCMC) methods, but these methods can be very computationally expensive and slow when dealing with large amount of data and/or a large number of parameters. An optimisation-based alternative to MCMC is called *variational inference*. Variational inference relies of being able to define a family of approximate posterior densities that is flexible enough to approximate the true posterior well, while at the same being tractable and lending itself to gradient-based optimisation.

It was to this end normalising flows were first proposed in a deep learning setting by Rezende and Mohamed (2015). Just as when using normalising flows for density estimation, these models benefit from the flexibility of neural networks to model flexible and tractable posterior distributions. Whereas density evaluation typically is the primary functionality of density estimators, we are often more interested in being able to generate new samples from a posterior density. Normalising flows facilitate this, as sampling from the modelled distribution simply amounts to a forward pass of the flow, which for aptly designed models can be done efficiently on parallel hardware.

1.2 What is deep learning?

The normalising flows presented in this thesis rely heavily on neural networks, and they all fit under the umbrella of *deep learning* models. An introduction to neural networks is given in Section 2.2, and in the following we provide some background on what we mean by "deep learning".

The term "deep learning" refers to a wide range of universal learning techniques. Deep learning models are composed of parameterised modules that are trained using gradient-based optimisation. These modules are typically variations of neural networks, and have been successfully applied to problems within numerous and very distinct fields, ranging from computer vision and image generation, to natural language processing, physics and biology. The reinforcement learning (RL) model AlphaGo made the headlines when it beat the world's best Go[1] player in four out of five games after teaching itself through self-play how to play the game.

The idea of using neural networks for learning tasks is not new, and can be traced back more than half a century. The first mathematical model of a neuron was published by McCulloch and Pitts (1943), a model that some years later inspired the Perceptron learning algorithm presented by Rosenblatt (1958). This model received much attention and sparked a great deal of interest in the field, but challenges in training the neural networks limited their success in the subsequent decades. The definite comeback of neural network is quite recent, and is by many considered to be the moment when Professor Geoffrey Hinton and two of his graduate students at University of Toronto won the *ImageNet Large Scale Visual Recognition Challenge* (Russakovsky et al., 2015) in 2012, using a convolutional neural network that outperformed the runner-up by a remarkable margin of 10.8%.

The breakthrough of deep learning is often assigned to the combination of *(i)* improved techniques for training deep neural networks on large datasets, *(ii)* the exponential increase of available data, and *(iii)* more computational power allowing for training of deeper networks. However, deep learning has arguably been most successful within the supervised regime, on classical problems like regression and classification. Such models are trained on labelled datasets with known input-output pairs. The pool of unlabelled data is vastly larger than the available amount of labelled data. In order to make use of the information that is in this unlabelled data, we need *unsupervised learning*.

The goal of unsupervised learning is to learn the structure or distribution of the data directly from the input, clustering unlabelled data into groups in a meaningful way. The deep learning pioneer and Turing medal winner Yann LeCun famously advocates that "The next revolution will not be supervised" (LeCun, 2018), referring to the future of machine learning. The idea of unsupervised learning is appealing also due its analogy to how humans learn, but more importantly, because of the great availability of unlabelled data and the value of the information that lies therein. Both density estimation and variational inference are examples of unsupervised learning problems.

[1] Go is an old Chinese board game, known to be much more complex than chess in terms of legal number of moves per turn, and number of legal possible board positions.

1.3 Goals and structure of the thesis

This thesis aims to give an independent introduction to the field of normalising flows, with focus on the class of autoregressive flows. We aim to give a clear exposition of normalising flows in general, and autoregressive normalising flows in particular. The reader is assumed to have basic familiarity with calculus, linear algebra, and probability theory, but the thesis is aimed to be self-contained when it comes to deep learning and normalising flows.

The structure of the thesis is as follows:

Chapter 2 lays out the general theoretical foundations of the thesis, most importantly density estimation and Bayesian inference. The chapter also contains two introductions to neural networks and variational inference, respectively.

Chapter 3 introduces the general theory about normalising flows. It contains a discussion on how to design a tractable normalising flow, and how normalising flows are useful for variational inference and density estimation. The chapter is rounded off with some illustrative experiments inspired by Rezende and Mohamed (2015) to build intuition about how normalising flows work.

Chapter 4 introduces the concepts of autoregressive density estimation and *autoregressive flows*, which is the central topic of the remaining chapters of the thesis. The *masked autoregressive distribution estimator (MADE)* (Germain et al., 2015) is thoroughly presented, as it is the main ingredient in two of the autoregressive flows presented in Chapter 5. Finally, we make a tiny contribution to the existing literature by formalising how to design a MADE with Gaussian conditionals.

Chapter 5 presents the *inverse autoregressive flow (IAF)* (Kingma et al., 2016), the closely related *masked autoregressive flow (MAF)*, and the *real-valued non-volume preserving (Real NVP) flow* (Dinh et al., 2017), including a comparison of the three flows. We briefly present the *neural autoregressive flow* (Huang et al., 2018) to provide an example of neural networks can be utilised to design even more expressive flows. Lastly, we present a batch normalisation layer adapted by (Papamakarios et al., 2017) to be used as a component in a normalising flow.

Chapter 6 introduces a novel flow for density estimation. We combine the coupling layer from Real NVP with the autoregressive layer from MAF into a crossover layer, hoping to increase the flexibility compared to the coupling layer. We name the resulting flow *hybrid autoregressive flow (HAF)*. The essential theory of the HAF is presented, followed by a discussion of some of its properties in relation to the MAF and the Real NVP.

Chapter 7 contains a variety of density estimation experiments: *(i)* we reproduce some experiments from the MADE paper on the binary MNIST dataset, and extend on these by conducting a set of new experiments, including some using the Gaussian MADE. *(ii)* we test the Real NVP model on a couple of two-dimensional toy densities. *(iii)* We compare the performance of MADE, Real NVP, MAF, and the new HAF on the two different datasets. These experiments include reproducing some results from the MAF paper (Papamakarios et al., 2017).

Chapter 8 summarises the thesis, and points out some directions for future research.

1.4 Implementation

PyTorch (Paszke et al., 2019) from Facebook and *TensorFlow* from Google (Abadi et al., 2015) are the two leading deep learning frameworks today. The core functionalities of a deep learning framework are automatic differentiation, easy-to-build neural network modules, and making distributed computing and training on multiple GPUs accessible for the user. A good deep learning framework should also facilitate easy exploration of ideas and allow for rapid model iteration. Both PyTorch and TensorFlow possess all the aforementioned qualities, but while TensorFlow still has an edge in the industry when it comes putting deep learning models into production, PyTorch has lately become the preferred framework for researchers (He, 2019), likely due to its flexibility and ease of use.

We chose to use PyTorch for all our experiments, because of its clean API and intuitive syntax. PyTorch feels "pythonic" and familiar for someone used to Python programming, and integrates seamlessly with common Python debugging tools and libraries. All models and experiments were implemented from scratch in Python using PyTorch, and the code is publicly available at `https://github.com/e-hulten`. The relevant repositories are `planar-flows`, `made`, `maf`, and `realnvp`.

Chapter 2

Preliminaries

In this chapter, we present the foundational theory that underpins the rest of the thesis. Statistics and many applications of machine learning are fundamentally concerned with quantifying uncertainty and probabilities. In this chapter, we formally define probability spaces and probability density functions, and introduce concepts like density estimation and Bayesian inference which give rise to the need for normalising flows. Lastly, we give a brief introduction to the essentials of neural networks, and to the field of variational inference.

2.1 Statistical preliminaries

2.1.1 Probability theory

A probability space (Klenke, 2013) is defined by a *sample space* S, equipped with a family of events $\mathcal{F}$ and a probability measure, P. This triplet denoted by $(S, \mathcal{F}, P)$. The sample space S is an arbitrary, non-empty set that contains all possible outcomes that we want to consider. An *event* A is a subset of the sample space, and $\mathcal{F}$ denotes the family of all events in a sample space. $\mathcal{F}$ defines a σ-algebra over S, and is defined as a collection of subsets of S that satisfies

i $\mathcal{F}$ contains the sample space: $S \in \mathcal{F}$.
ii $\mathcal{F}$ is closed under complements: If $A \in \mathcal{F}$, then $A^c \in \mathcal{F}$.
iii $\mathcal{F}$ is closed under countable unions: If $(A_i)_{i=1}^{\infty} \in \mathcal{F}$, then $\bigcup_{i=1}^{\infty} A_i \in \mathcal{F}$.

Elements of a σ-algebra are also called *measurable sets*, and the pair $(S, \mathcal{F})$ is called a *measurable space*. To define a *probability space*, we need a probability measure P, i.e., a function that maps the elements of $\mathcal{F}$ to the unit interval $[0, 1]$ and assigns a *likelihood* to each event in $\mathcal{F}$. A probability measure satisfies the axioms of Kolmogorov:

i $P(A) \geq 0$ for all events in $A \in \mathcal{F}$.
ii $P(S) = 1$.
iii Any countable sequence of disjoint events $(A_i)_{i \geq 1}$ satisfies $P(\cup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} P(A_i)$.

A real-valued *random variable* X is a function $X : (S, \mathcal{F}) \to (\mathbb{R}, \mathcal{B}(\mathbb{R}))$, where $\mathcal{B}(\mathbb{R})$ is the *Borel σ-algebra* over the real numbers. For every Borel subset $B \in \mathcal{B}(\mathbb{R})$, we denote $\{X \in B\} := \{X^{-1}(B)\}$ and $P(B) := P(\{X \in B\}) = P(X^{-1}(B))$. The *distribution function* of a random variable X is defined as the map $F_X : x \mapsto P(X \leq x)$. From a distribution function $F : \mathbb{R}^D \to [0, 1]$, we can define the *probability density function*, or just *density function*, as $p(\mathbf{x})$ such that

i $F(\mathbf{x}) = \int_{-\infty}^{x_1} \cdots \int_{-\infty}^{x_D} p(\mathbf{x}')d\mathbf{x}'$ for $\mathbf{x} = (x_1, \ldots, x_D) \in \mathbb{R}^D$.
ii $p(\mathbf{x}) \geq 0$ for all $\mathbf{x} \in \mathbb{R}^D$.

The existence and uniqueness of the density function is ensured by the *Radon-Nikodym theorem*, which ensures that two density functions can only differ over a set of measure zero. Please refer to (Klenke, 2013) for the theorem, and a thorough introduction to probability theory.

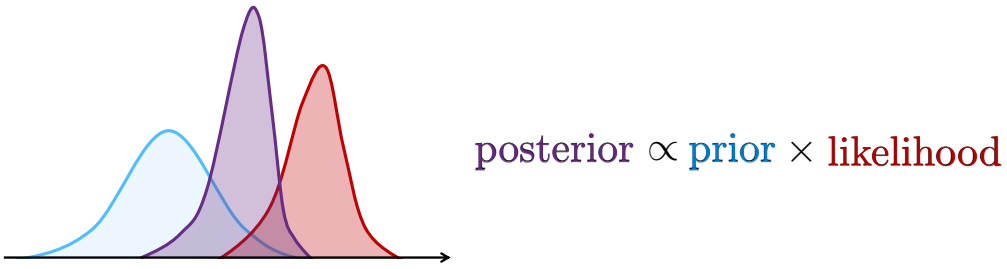


Figure 2.1: Illustration of Bayes' rule showing the relationship between the prior, likelihood, and posterior. This figure is best viewed in colour.

Density functions will directly and indirectly be the underlying topic of interest for the rest of the thesis. We will only be concerned with continuous random variables, and assume that the probability density function always exists. Note that a random variable can also be a vector, in which case the vector has scalar elements that are random variables on the same probability space $(S, \mathcal{F}, P)$.

Notation: Throughout the thesis, we will use bold, lowercase letters and symbols like $\mathbf{x}$ and $\boldsymbol{\phi}$ to denote vectors, including random vectors. It will be made clear from the context whether a vector is a random variable or an observation. Where necessary, we will use subscripts to indicate samples to avoid ambiguity, so a set of observations will typically be written as $\{\mathbf{x}_1, ..., \mathbf{x}_N\}$. All vectors are given as column vectors unless else is specified. Bold, uppercase letters like $\mathbf{X}$ denote matrices, and lowercase roman symbols such as b denote scalars.

2.1.2 Bayesian inference

The Bayesian framework provides a mathematical toolbox that can be used for modelling probabilistic systems while taking uncertainties into account. Bayesian models are formulated in terms of probability densities used to express beliefs about unknown quantities. We consider some unobserved parameter of interest $\boldsymbol{\theta}$. The prior beliefs about this quantity are represented through the prior distribution $p(\boldsymbol{\theta})$. Further, we assume there is some statistical relationship by the observable quantity $\mathbf{x}$ and the parameter of interest. We can then use new observations of $\mathbf{x}$ to update our beliefs about the parameters. The updated beliefs about the parameters are expressed through a *posterior distribution*. In particular, we combine *likelihood* $p(\mathbf{x} \mid \boldsymbol{\theta})$ with the prior through Bayes' rule:

$$p(\boldsymbol{\theta} \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid \boldsymbol{\theta})p(\boldsymbol{\theta})}{\int p(\mathbf{x} \mid \boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{\theta}} = \frac{p(\mathbf{x} \mid \boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathbf{x})} \propto p(\mathbf{x} \mid \boldsymbol{\theta})p(\boldsymbol{\theta}). \tag{2.1}$$

See also Figure 2.1. The posterior distribution is the conditional distribution of the parameters given the observed data, and represents the best information we have available about the parameters, taking both our prior beliefs and the observations into account. When working with simple models, the practitioner is able to choose the conjugate prior for the likelihood, in which case the posterior is available analytically. Alas, the true posterior parameter distribution is not easily available for most models of interest, due to the integral in the denominator of Equation (2.1) being intractable. In such cases, we have to resort to approximate Bayesian inference methods. Some of these will be presented more thoroughly in Section 2.3.

2.1.3 Density estimation

The importance of density estimation has been properly motivated in Section 1.1. In this section, formalise the problem of density estimation in more theoretical terms than what was done in the introduction. Density estimation is a classical problem in statistics, and can roughly be posed as: *Given a finite set of i.i.d. samples, we want to recover the probability density function associated with their underlying generative process.* The true density function provides a description of the joint statistical properties of the data, and an estimate of it can be used to evaluate the likelihood of arbitrary new observations.

A finite set of samples gives limited insight into the generative process, and the job of the density estimator is to use the information provided by the samples in conjunction with any prior knowledge of the generative process to estimate the true density function as well as possible. Classical statistical methods for density estimation are to a large extent concerned with fitting the data using some *parameterised* model family. Parametric models have a pre-determined number of learnable parameters, and the problem density estimation translates into finding the set of parameters that makes the parametric model as similar as possible to the true density. The parameters are typically learned by maximising the average log-likelihood of the training data under the parametric model.

The space of densities that can be represented by a simple parametric family is fairly limited, and parametric models rely heavily on prior knowledge about the data that can be incorporated through the choices we make when modelling the probability distribution, e.g., through the parametric shape of the estimating model. The flexibility of parametric models can be increased by combining parametric models into mixture models. Gaussian mixtures and smoothing splines are examples of parametric approaches that can be very flexible, and Gaussian mixtures are in fact universal density estimators in the limit when the number of Gaussian components goes to infinity (McLachlan and Basford, 1988).

The fauna of classical density estimation models also include *non-parametric* approaches such as histograms and kernel density estimators. These methods make weaker prior assumptions about the density that is to be estimated, and their complexity grow with the complexity and shape of the estimated density. Non-parametric methods are perhaps the most popular and widely used ones for density estimation, but they suffer severely from the curse of dimensionality. High dimensional spaces will in practice be very sparsely populated by data points, requiring exponentially more data to get sufficient coverage of the data space to get a good estimate of the density function.

Neural density estimators

Recently, a new line of research using neural networks to parameterise density estimators has emerged (Germain et al., 2015; Papamakarios et al., 2017; Dinh et al., 2017). Adapting the terminology from (Papamakarios, 2019), we hereafter refer to such models as *neural density estimators*. These models utilise the flexibility and large learning capacity of neural networks and the computational advantages of the deep learning paradigm to approximate very high-dimensional densities.

A neural density estimator takes in some D-dimensional data $\mathbf{x}$ and returns a real number $f_{\boldsymbol{\theta}}(\mathbf{x})$. A neural density estimator is characterised by having the property:

$$\int_{\mathbb{R}^D} \exp f_{\boldsymbol{\theta}}(\mathbf{x}) d\mathbf{x} = 1 \tag{2.2}$$

for all sets of parameters $\boldsymbol{\theta}$. As a consequence, $q_{\boldsymbol{\theta}}(\mathbf{x}) = \exp f_{\theta}(\mathbf{x})$ is a valid density function, and the neural network that can be used to estimate a probability density. As for other parametric model, the parameters of the neural density estimator are learned by maximising the average log-likelihood of the training data under the density defined by the neural density estimator:

$$\max_{\boldsymbol{\theta}} \frac{1}{N} \sum_i \log q_{\boldsymbol{\theta}}(\mathbf{x}_i) = \max_{\boldsymbol{\theta}} \frac{1}{N} \sum_i f_{\boldsymbol{\theta}}(\mathbf{x}_i) \tag{2.3}$$

Maximum likelihood density estimators have desirable asymptotical properties like consistency (maximum likelihood estimators converge in probability to the true density), it is efficient (it attains the Cramér-Rao lower bound when the number of observations goes to infinity, i.e., it gives the lowest mean square error among all estimators), and in the limit, learning a distribution q as an approximation to a true density p through maximum likelihood estimation is equivalent to minimising the KL-divergence between p and q (Papamakarios, 2019). Being neural networks, neural density estimators lend themselves naturally to gradient-based training. We will elaborate more on neural density estimators, and how to design them, in Chapter 3, 4, 5, and 6.

2.2 The fundamentals of neural networks

2.2.1 The neuron

The basic unit of a neural network is for historical reasons called a *neuron*, reflecting that the computational neuron is loosely inspired by how neurons operate in the brain. A neuron in a neural

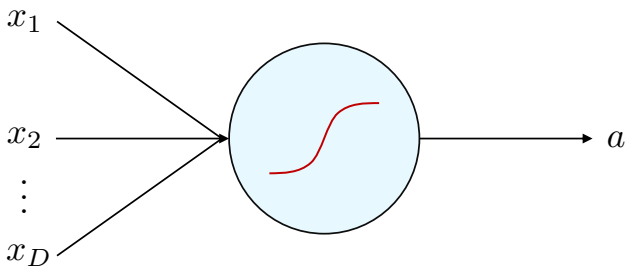


Figure 2.2: Overview of a computational neuron. The red line represents a non-linear activation function.

network performs two operations; it computes a weighted sum of its inputs, adds a bias term, and then passes this sum through some (usually) non-linear activation function. A neuron outputs a scalar a, which we refer to as its *activation*. A neuron is depicted in Figure 2.2.

An activation function $g(\cdot)$ is a nonlinear function that takes the weighted sum and bias described above and returns the scalar activation $a = g(f_{\mathbf{w}}(\mathbf{x})) = g(\mathbf{w}^T\mathbf{x} + b)$, where $\mathbf{w}$ denotes a weight vector of the same dimensionality as $\mathbf{x}$, and b denotes the scalar bias term of the given neuron. By *layer*, we refer to a group of neurons operating at the same depth in a neural network. Each layer has an associated weight matrix $\mathbf{W}$ with rows corresponding to the transposed weight vector of each neuron in the layer. In general, the input to a neuron will not be the feature vector $\mathbf{x}$, but rather the activation vector from the previous layer in the network.

Common choices of activation functions are:

- The rectified linear unit (ReLU) activation function: $g(x) = \text{ReLU}(x) = \max(0, x)$.
- The sigmoid (logistic) activation function: $g(x) = \sigma(x) = \frac{1}{1+e^{-x}}$.
- The hyperbolic tangent activation function: $g(x) = \tanh(x)$.

To allow for gradient-based training of the neural network, an activation function also has to be differentiable almost everywhere. For example, the ReLU activation function, which perhaps is the most widely used activation function today, is differentiable everywhere but in $x = 0$.

Also note that for regression tasks, the "activation function" of the last layer is set to be the identity mapping, i.e., $g(x) = x$. The reason for this, is that the activation functions "squash" their inputs from $\mathbb{R}$ to some sub-domain of $\mathbb{R}$, which is not compatible with the domain of the typical response variable in a regression, e.g., ReLU : $\mathbb{R} \to \mathbb{R}_{\geq 0}$, $\sigma : \mathbb{R} \to [0, 1]$, and tanh : $\mathbb{R} \to [-1, 1]$.

2.2.2 Feed-forward neural networks

Traditional linear regression can be viewed as the simplest case of a neural network, with only one hidden unit. The activation function of this neuron is simply the identity mapping, $g(x) = x$. The model assumption in linear regression is that there exists a *linear* function $f : \mathbb{R}^D \to \mathbb{R}$ between the inputs and outputs. The model is assumed to be on the form $\hat{y}_i = f_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}_i + b$, and the residuals $\epsilon_i = y_i - \hat{y}_i$ are usually assumed to follow a zero-mean Gaussian distribution with some finite variance, σ^2. If we assume the variance to be constant for all observation pairs, we have what is called *homoscedastic* noise.

During training of the network, we seek to learn the network parameters that minimise some loss function measuring how good our current approximation is. For regression tasks, we often use the mean squared loss (MSE) loss function. Optimising the weights and biases of the network with respect to the MSE loss yields the maximum likelihood estimate of the weights, because minimising the MSE loss is equivalent to minimising the negative log-likelihood of the data when we have assumed a Gaussian likelihood[1] over the outputs.

[1]Note that minimising the MSE loss yields maximum likelihood estimates with respect to a Gaussian likelihood over the network outputs also for deeper networks than the described minimal regression model.

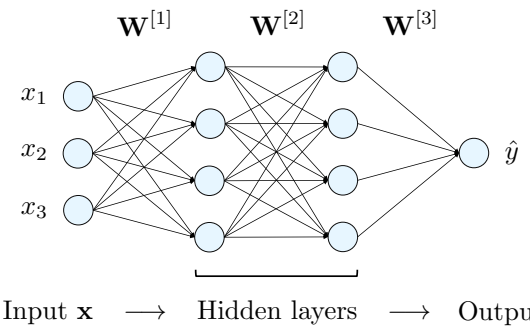


Figure 2.3: A feed-forward neural network with two hidden layers.

Whereas a linear model, such as a linear or logistic regression model, is only able to learn linear relationships in the data, we are in most real-world cases interested in learning more complex relationships between the inputs and the outputs. This motivates the introduction of non-linearity to the regression model, which is achieved by using nonlinear activation functions. By extending the one-neuron model in the natural way by adding more neurons to each layer and stacking several layers in sequence, we make a neural network that is able to capture relationships in the data that are highly nonlinear in both data and parameters. This is opposed to linear regression (linear in both data and parameters) and linear basis function regression (nonlinear in data, linear in parameters).

For a layer with more than one neuron, we have a corresponding weight matrix $\mathbf{W}^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$ where each row is a weight vector corresponding to the input to one neuron in the l-th layer, and where $n^{[l]}$ denotes the number of neurons in the l-th layer. The activation vector from layer l is given by $\mathbf{a}^{[l]} = g^{[l]}(\mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]})$ where the activation function $g^{[l]}(\cdot)$ operates element-wise on the elements in the vector, and the superscript in brackets enumerates the layers. The activation vector of each hidden layer becomes the input of the next one in the forward pass from the input layer to the output layer. Both the input layer and the output layer are in general vector valued.

Each distinct way of arranging and connecting neurons in a neural network is referred to as the neural network *architecture*. The *feed-forward* neural network as depicted in Figure 2.3 is characterised by having no loops, such that the information only flows in one direction from the input layer to the output layer, without intermediate outputs from the hidden layers ever being fed back into the model. The simplest architecture is the *fully connected* feed-forward neural network where all outputs from each layer are passed on to all the neurons in the next layer repeatedly from the input layer through the hidden layers and to the output layer. A variety of more complex classes of architectures exist, and we refer to (Goodfellow et al., 2016) for a general introduction to neural networks.

In a deep network, the information from the input layer goes through many layers of non-linear transformations. Each layer is a function, h_l, of its input vector, and by stacking the layers, we can view the output of a feed-forward neural network as a function composition applied to the input, namely $\hat{\mathbf{y}}_i = (h_L \circ h_{L-1} \circ \cdots \circ h_1)(\mathbf{x}_i)$, approximating the true mapping between inputs and outputs. The *universal approximation theorem* for neural networks states that infinitely wide one-layer feed-forward NNs are universal approximators of Borel measurable functions between finite spaces (Hornik et al., 1989), illustrating the power of the feed-forward architecture. In practice, deeper and narrower networks are easier to train and have shown to be incredibly much more useful than shallow and wide networks.

The ultimate goal of training feed-forward neural networks is to approximate the true mapping between the inputs and the outputs as well as possible. This is done by incrementally adjusting the parameters to minimise some loss function $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$, where $\hat{\mathbf{y}}$ is the output from the network and $\mathbf{y}$ is the true output as defined by the training data. The loss function has to be differentiable with respect to the parameters of the network in order to update the parameter values using gradient-based methods, as is in practice the only successful approach to training deep neural networks. Since the model is learning from given input-output pairs, training a feed-forward neural network for a regression or classification task is an example of what we call *supervised learning*.

2.2.3 Gradient-based learning

In a feed-forward neural network as described in the previous sections, we often refer to the process from when an input $\mathbf{x}$ is fed to the network to when the network outputs a prediction $\hat{\mathbf{y}}$ as the *forward pass*. When the output of the network is obtained, we compute the value of a scalar cost function $\mathcal{L}(\boldsymbol{\theta})$ using the entire training set (or mini-batch – a subset of the training set). The cost function is the average of the loss function for all data points we have used for training, plus an optional regularisation term. E.g., if we use the MSE loss, the cost function is:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} ||\hat{\mathbf{y}}_i \mathbf{y}_i||_2^2, \tag{2.4}$$

where N is the number of training samples. The network parameters $\boldsymbol{\theta}$ are randomly initialised, and then updated at the end of each forward pass, in order to minimise this cost function. Because the minimum of the cost function is not available analytically, the parameters are updated through minimisation using gradient-based methods. Basic gradient descent updates are given by:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \mathbf{d}, \tag{2.5}$$

where $\mathbf{d}$ is some function describing the descent direction as a function of the gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L}$. In basic gradient descent, $\mathbf{d}$ is simply equal to $\eta\nabla_{\boldsymbol{\theta}}\mathcal{L}$ where η is some small stepsize.

To use any variant of gradient descent, we need the numerical value of the gradients of the cost function with respect to each network parameter. The gradients are computed using the *backpropagation algorithm* (Goodfellow et al., 2016, Chapter 6.5), which utilises the chain rule of calculus to recursively compute the numerical derivatives of the parameters in an efficient manner. The parameter update in Equation (2.5) is thus referred to as the *backward pass*, as the error is propagated backwards from the output layer towards the input layer when computing the gradients of the cost function.

There is a variety of different stochastic optimisation methods exist that are more commonly used than gradient descent for training neural networks today. Such methods apply more complex strategies for computing $\mathbf{d}$ than just using the current gradient, and use additional information to compute the descent direction, like previous gradients and the current number of iterations. In particular, the *Adam* optimiser (Kingma and Ba, 2014) has become very popular due to its versatility and good performance across different domains in machine learning.

Mini-batch stochastic gradient descent

In practice, we do not loop through the entire training set for each parameter update. When working with big datasets, looping through millions of samples for a single gradient descent update would lead to very slow learning. Whereas vanilla, or *batch*, gradient descent involves averaging of the cost gradient over all training samples per iteration, *stochastic* gradient descent (SGD) computes the cost gradient of only one randomly chosen training sample per iteration. This yields a noisier path towards the minimum, which could potentially hinder the optimisation procedure from settling down at the optimal point, but it speeds up the learning process compared to using batch gradient descent.

A compromise between batch gradient descent and stochastic gradient descent is to compute the gradient on a random subset of the training data. This gives a reasonable estimate of the true gradient without the computational cost of using the entire training set to compute the gradient. Using random subsets to compute the gradient smooths out most of the noisiness from SGD, while offering significantly accelerated learning compared to batch gradient descent. This is called *mini-batch* stochastic gradient descent. The term *batch size* refers to the number of training samples used to calculate the gradient per iteration, and for mini-batch SGD, the batch size is typically or order $\sim 10 - 10^3$. The optimisation runs until some stopping criterion is met, e.g., until a pre-determined number of iterations through the entire training set is completed, referred to as the number of *epochs*.

2.2.4 Overfitting and how to avoid it

The goal of training a machine learning model, is to learn a model that generalises well to new and unseen data. An underlying assumption is that the unseen data we will make predictions on is expected to follow the same data generating process as the training data. The problem of *overfitting* surfaces when we fit a model too closely to the distribution of our training set. That is, we fit our

model not only to the structure of the data, but also to the noise present in the dataset. This can lead to a very low training loss, but a model that will fail to generalise to unseen data.

The problem of overfitting is particularly present when working with heavily parameterised neural networks, and consequently, many solutions have been proposed in the deep learning literature to prevent overfitting. A classical and widely used solution is to add l_1 or l_2 regularisation to the weights to keep them small (analogous to Lasso and Ridge regression, respectively), but more commonly used in neural networks today are stochastic regularisation techniques (SRTs).

The most widely adopted technique is called **dropout** (Hinton et al., 2012). Dropout is implemented per-layer, and can be applied to any layer except the output layer. For each iteration, each hidden unit in the regularised layers is switched off with some dropout probability $1 - p$. Consequently, we train different configurations of the layer at each iteration. If a network has H hidden units regularised by dropout, there are 2^H different network configurations, so by using dropout in the training phase, we train an exponential number of thinned models in parallel, with extensive weight sharing.

Training these thinned networks is believed to lead to more robust feature representation internally in the network, as a network trained with dropout can not rely too heavily on any particular weight, effectively spreading the weight learning between the nodes. Dropout has had a tremendous empirical success, and has shown to prevent co-adaption between neurons, keep the weights small, and prevent overfitting. At test time, the thinned networks are averaged by downscaling the activations of one unthinned network by factor p, i.e., by the probability that each neuron remains in the network.

Finally, **early stopping** is an intuitive and easy-to-implement technique to prevent overfitting. The technique requires that the data is split into a training set and a validation set (and preferably also a test set). The model is trained only on the training set, but during training, the loss is also computed on the held-out validation set with regular intervals, e.g., after each epoch. The error on the validation set acts as a proxy for the generalisation error, and when the validation error is no longer decreasing with more training, this is a good indicator that the model has started to overfit the training set.

As long as the validation loss keeps decreasing, we save the weights of the network after every epoch. When the validation loss is no longer improving, we continue the training for a pre-determined number of epochs referred to as the *patience*. When the patience runs out, we use the best saved network weights as our model. If the validation loss suddenly improves (over the all-time best) after a period without improvement, the patience counter is reset and we repeat the process described above.

2.3 Approximate posterior inference

A general challenge in Bayesian inference is to approximate the posterior parameter density $p(\boldsymbol{\theta} \mid \mathbf{x})$. We are only able to obtain the true posterior density of relatively simple models, e.g., by using conjugate priors. For more complex problems, we have to resort to approximate posterior inference algorithms to obtain an estimate of the true posterior density. The research on approximate posterior is largely divided into two lines. Sampling-based Markov chain Monte Carlo (MCMC) algorithms are considered the gold standard for many applications, but can be very computationally demanding, particularly when working with large datasets. An alternative, optimisation-based framework for performing posterior inference is called *variational inference (VI)* (Jordan et al., 1999; Blei et al., 2016). Variational inference lacks some of the statistical rigour and guarantees that come with MCMC methods, but it has gained popularity in recent years because it scales better to large datasets and is compatible with the gradient based paradigm of modern machine learning. In this section, we aim to give an introduction to variational inference and highlight its strengths and weaknesses.

2.3.1 Variational inference

We formalise variational inference in the context where we have i.i.d. samples $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ coming from some distribution $p(\mathbf{x} \mid \mathbf{z})$ where $\mathbf{z}$ is an unobserved continuous latent variable with prior $p(\mathbf{z})$. Performing Bayesian inference in this setting amounts to finding the posterior density of the latent variables conditioned on the data as given by Bayes' theorem:

$$p(\mathbf{z} \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid \mathbf{z})p(\mathbf{z})}{p(\mathbf{x})} = \frac{p(\mathbf{x} \mid \mathbf{z})p(\mathbf{z})}{\int_{\mathbf{z}} p(\mathbf{x}, \mathbf{z}) d\mathbf{z}}.$$

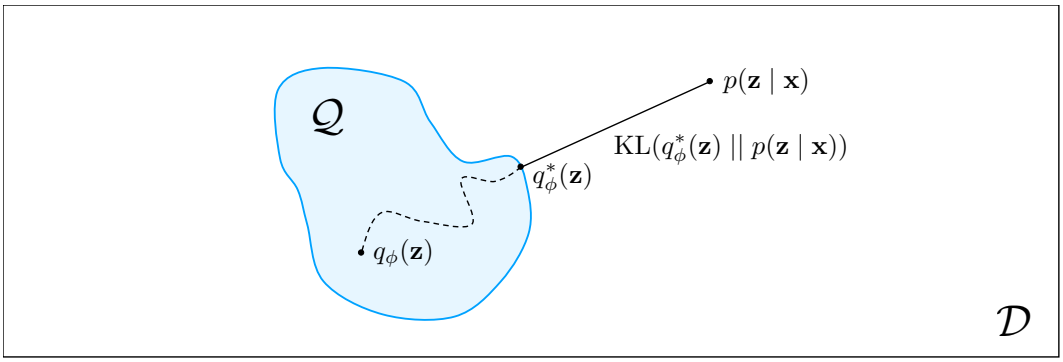


Figure 2.4: Visualising variational inference. $\mathcal{Q}$ denotes the variational family, and $\mathcal{D}$ denotes the space of all possible distributions over $\mathbf{z}$.

Alas, the integral in the denominator is usually intractable because it involves integrating over all values of the latent variables. That means that finding the analytical (or *true*) posterior of the latent variables is not feasible, and we have to perform approximate inference in order to get insights about the posterior distribution of the latent variables.

The key idea in variational inference is to avoid the integral in the above expression as a whole, by approximating the posterior $p(\mathbf{z} \mid \mathbf{x})$ directly by some simpler density. We propose a family of approximate densities $\mathcal{Q}$ parameterised by the variational parameters $\boldsymbol{\phi}$, and seek to find the density $q_\phi^*(\mathbf{z})$ within this family that is most similar to the true posterior density by some dissimilarity measure. The density that minimises this dissimilarity measure is found by minimising the divergence between the approximate and the true posterior with respect to the parameters of the approximate posterior. The idea is to use $q_\phi^*(\mathbf{z})$ as a proxy for the true posterior and use it for downstream tasks.

Technically, variational inference includes any procedure using optimisation to approximate a density (Wainwright and Jordan, 2008), so *any* dissimilarity measure between distributions can be used as the objective function. However, most literature on variational inference focus on finding the density $q_{\boldsymbol{\phi}}^*(\mathbf{z}) \in \mathcal{Q}$ that minimises the reverse[2] *Kullback-Leibler (KL) divergence* (Kullback and Leibler, 1951) to the exact posterior. We will elaborate on the choice of using KL divergence as a dissimilarity measure in Section 2.4. The reverse KL divergence between the parametric approximating density $q_{\boldsymbol{\phi}}(\mathbf{z})$ and the target density $p(\mathbf{z} \mid \mathbf{x})$ is defined as:

$$\mathrm{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}) \mid\mid p(\mathbf{z} \mid \mathbf{x})) := \int_{\mathbf{z}} q_{\boldsymbol{\phi}}(\mathbf{z}) \log \frac{q_{\boldsymbol{\phi}}(\mathbf{z})}{p(\mathbf{z} \mid \mathbf{x})} d\mathbf{z} \tag{2.6}$$

The KL divergence is defined if and only if for all $\mathbf{z}$, $p(\mathbf{z} \mid \mathbf{x}) = 0$ implies $q_{\boldsymbol{\phi}}(\mathbf{z}) = 0$, i.e., we require that $q_{\boldsymbol{\phi}}(\mathbf{z})$ is absolutely continuous with respect to $p(\mathbf{z} \mid \mathbf{x})$. The KL divergence is always non-negative, and equal to zero if and only if $q_{\boldsymbol{\phi}}(\mathbf{z}) = p(\mathbf{z} \mid \mathbf{x})$ almost everywhere. Although the KL divergence shares some properties with a metric on the set of probability densities, we note that the KL divergence is not symmetric, nor does it satisfy the triangle inequality.

We denote the approximating density that minimises the KL divergence in Equation (2.6) to the true posterior by $q_{\boldsymbol{\phi}}^*(\mathbf{z})$. The (locally) optimal parameters $\boldsymbol{\phi}^* = \arg\min_{\boldsymbol{\phi}} \mathrm{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}) \mid\mid p(\mathbf{z} \mid \mathbf{x}))$ are found through gradient-based optimisation with respect to the variational parameters. The optimisation process is illustrated in Figure 2.4, where we start from some initial approximate density q_ϕ and end up at q_ϕ^* by minimising the Kullback-Leibler divergence to the true posterior density. Depending on the choice of variational family $\mathcal{Q}$, the optimal approximation contained in this family may be arbitrarily close to, or far away from the density that we try to approximate.

2.3.2 Deriving the variational lower bound

Recall that the posterior density we seek to approximate is given by $p(\mathbf{z} \mid \mathbf{x}) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z})/p(\mathbf{x})$. The marginal likelihood $p(\mathbf{x})$ in the denominator is in the context of Bayesian statistics often referred to as the *evidence*, and gives us the probability density of the observations after integrating over

[2] As opposed to the *forward* KL divergence $\mathrm{KL}(p(\mathbf{z} \mid \mathbf{x}) \mid\mid q_{\boldsymbol{\phi}}(\mathbf{z}))$.

the latent space. As previously mentioned, it is exactly this integral that usually makes finding the posterior intractable. However, if we repeat the minimisation objective given in Equation (2.6):

$$\mathrm{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}) \mid\mid p(\mathbf{z} \mid \mathbf{x})) := \int_{\mathbf{z}} q_{\boldsymbol{\phi}}(\mathbf{z}) \log \frac{q_{\boldsymbol{\phi}}(\mathbf{z})}{p(\mathbf{z} \mid \mathbf{x})} d\mathbf{z} \tag{2.7}$$

we see that the intractable true posterior $p(\mathbf{z} \mid \mathbf{x})$ is still present inside the integral that we need to compute to obtain the KL divergence. It is therefore clear that we cannot compute the KL divergence directly. Naively it therefore seems as if using the KL divergence does not bring us any closer to finding an approximate posterior. To enable the use of the KL divergence as the minimisation object, we proceed to show that the KL divergence can be minimised indirectly by maximising the marginal probability of our observations. First, we find an expression for a lower bound for the log-evidence:

$$\begin{aligned} \log p(\mathbf{x}) &= \log \left(\int_{\mathbf{z}} p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z}) d\mathbf{z} \right) \\ &= \log \left(\int_{\mathbf{z}} \frac{p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})}{q_\phi(\mathbf{z})} q_\phi(\mathbf{z}) d\mathbf{z} \right) \\ &\geq \int_{\mathbf{z}} \log \left(\frac{p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})}{q_\phi(\mathbf{z})} \right) q_\phi(\mathbf{z}) d\mathbf{z} =: \mathrm{ELBO} \end{aligned} \tag{2.8}$$

In the third line we make use of Jensen's inequality and the concavity of the $\log$ function[3]. Note that the logarithm is a monotonically increasing function with respect to its argument, so maximising the lower bound for the log-evidence will also maximise the lower bound for the evidence itself. This lower bound is referred to as the *variational lower bound*, or the *evidence lower bound (ELBO)*.

We show how the ELBO relates to the KL divergence by expanding the expression in Equation (2.6):

$$\begin{aligned} \mathrm{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}) \mid\mid p(\mathbf{z} \mid \mathbf{x})) &= \int_{\mathbf{z}} q_{\boldsymbol{\phi}}(\mathbf{z}) \log \frac{q_{\boldsymbol{\phi}}(\mathbf{z})}{p(\mathbf{z} \mid \mathbf{x})} d\mathbf{z} \\ &= \int_{\mathbf{z}} q_{\boldsymbol{\phi}}(\mathbf{z}) \log \frac{q_{\boldsymbol{\phi}}(\mathbf{z})}{p(\mathbf{z}, \mathbf{x}) / p(\mathbf{x})} d\mathbf{z} \\ &= \int_{\mathbf{z}} q_{\boldsymbol{\phi}}(\mathbf{z}) \log \frac{q_{\boldsymbol{\phi}}(\mathbf{z})}{p(\mathbf{z}, \mathbf{x})} d\mathbf{z} + \int_{\mathbf{z}} q_\phi(\mathbf{z}) \log p(\mathbf{x}) d\mathbf{z} \\ &= \int_{\mathbf{z}} q_{\boldsymbol{\phi}}(\mathbf{z}) \log \frac{q_{\boldsymbol{\phi}}(\mathbf{z})}{p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})} d\mathbf{z} + \log p(\mathbf{x}) \\ &= -\int_{\mathbf{z}} q_{\boldsymbol{\phi}}(\mathbf{z}) \log \frac{p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})}{q_{\boldsymbol{\phi}}(\mathbf{z})} d\mathbf{z} + \log p(\mathbf{x}) \\ &= -\mathrm{ELBO} + \log p(\mathbf{x}) \end{aligned} \tag{2.9}$$

Rearranging the last line of Equation (2.9), we obtain

$$\log p(\mathbf{x}) = \mathrm{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}) \mid\mid p(\mathbf{z} \mid \mathbf{x})) + \mathrm{ELBO} \tag{2.10}$$

The left side of Equation (2.10) is independent of $\boldsymbol{\phi}$, and is hence constant for any choice of variational distribution. Because the sum of the two terms on the right side of Equation (2.10) is constant, it follows that increasing the value of one term will decrease the value of the other term by the same amount. In particular, maximising the ELBO with respect to the variational parameters is equivalent to minimising the Kullback-Leibler divergence between the approximate and the true posterior. This is an important result, because it allows us to minimise the KL divergence between the approximating posterior distribution and the true posterior density without having explicit knowledge about the shape of the true posterior density itself.

2.3.3 The variational objective

The observation that the KL divergence can be minimised indirectly by maximising the ELBO motivates our choice of objective function to be the negative ELBO. Clearly, minimising the negative

[3] Jensen's inequality is commonly presented as a result for a convex function $f(\cdot)$ as $f(\mathrm{E}(X)) \leq \mathrm{E}(f(X))$. Recalling that a concave function g is the negative of a convex function f, we get by Jensen's inequality $-g(\mathrm{E}(X)) \leq \mathrm{E}(-g(X))$, so $g(\mathrm{E}(X)) \geq \mathrm{E}(g(X))$ for a concave function g.

ELBO is the same as maximising the ELBO, but by convention in machine learning and optimisation literature and frameworks, we prefer minimising a cost function over maximising a reward function. We define the relevant cost function, often referred to as the *variational objective*, as:

$$\begin{aligned}\mathcal{L}_{VI}(q_{\boldsymbol{\phi}}) &:= -\text{ELBO} \\ &= -\int_{\mathbf{z}} \log\left(\frac{p(\mathbf{x} \mid \mathbf{z})p(\mathbf{z})}{q_{\phi}(\mathbf{z})}\right) q_{\phi}(\mathbf{z}) d\mathbf{z} \\ &= -\int_{\mathbf{z}} \left(\log p(\mathbf{x} \mid \mathbf{z}) - \log\frac{q_{\phi}(\mathbf{z})}{p(\mathbf{z})}\right) q_{\phi}(\mathbf{z}) d\mathbf{z} \\ &= -\mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z})}\left(\log p(\mathbf{x} \mid \mathbf{z})\right) + \mathrm{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}) \,||\, p(\mathbf{z})).\end{aligned} \tag{2.11}$$

When the cost function is written out in this form, we see that minimising the first term of $\mathcal{L}_{VI}(q_{\boldsymbol{\phi}})$ can be interpreted as maximising the expected log-likelihood of the observations with respect to the variational distribution over the latent variables, while the minimisation of the second term at the same time ensures that the variational distribution does not diverge too much from the chosen prior density over the latent variables. The second term penalises complex approximating densities and acts as an Occam's razor term, effectively prohibiting unnecessarily complex posteriors.

Monte Carlo gradient estimation

A challenge with minimising the variational objective in Equation (2.11) directly is that each evaluation of the objective seemingly involves summing over the entire dataset (which may be very big) to compute the expected log-likelihood under the variational distribution. To perform gradient-based optimisation of the variational parameters, we need to compute $\nabla_{\boldsymbol{\phi}} \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z})}\left(\log p(\mathbf{x} \mid \mathbf{z})\right)$. Following Graves (2011); Kingma and Welling (2013); Blundell et al. (2015), mini-batch optimisation together with Monte Carlo gradient estimation to compute the gradient of the ELBO. Mini-batch optimisation using stochastic gradient descent is often referred to as *doubly-stochastic estimation* because there are two sources of stochastisity; one in the partitioning of the mini-batches in stochastic gradient descent, and one in the Monte Carlo approximation of the expectation of the gradient. An extensive survey on Monte Carlo gradient estimators has been conducted by Mohamed et al. (2019).

To compute the gradient of the ELBO in Equation (2.11) we are also reliant on being able to evaluate or estimate the prior KL term $\mathrm{KL}(q_{\boldsymbol{\phi}}(\mathbf{z}) \,||\, p_{\theta}(\mathbf{z})))$. If we choose the variational distribution $q_{\phi}(\mathbf{z})$ and the prior distribution $p_{\theta}(\mathbf{z})$ to have the same form (e.g. choose both to be Gaussians), then the prior KL term has a closed form expression that can be evaluated directly. If the term cannot be evaluated analytically, it is estimated using Monte Carlo estimation as well.

2.3.4 Limitations of variational inference

Compared to MCMC methods, variational inference is often faster and scales better to large data. One caveat that comes with the variational inference approach is that its statistical properties are not as well understood as the ones of traditional MCMC techniques (Blei et al., 2016). Given a suitable Markov chain as the starting point, MCMC is guaranteed to asymptotically produce samples from the target distribution, giving a numerical approximation of the true posterior. Variational inference does not come with such promises, as it only provides a locally-optimal analytical approximation to the true posterior. In fact, variational inference techniques are known to underestimate the posterior variance, and we do in general not know how accurate our approximation is (Pati et al., 2017).

The ideal choice of variational family should be flexible enough to contain the true posterior density, but also be simple enough to make the optimisation tractable. One of the most widely used approximations, is the *mean-field approximation*, which compromises the flexibility of the approximate distribution to get a convenient optimisation procedure. The method takes its name from physics and assumes independence between all latent variables, yielding a fully factorised distribution:

$$q_{\boldsymbol{\phi}}(\mathbf{z}) = \prod_{i=1}^{M} q_{\boldsymbol{\phi_i}}(z_i) \tag{2.12}$$

where $\mathbf{z}_i$ denotes the latent variables associated with the i-th observation, and $\boldsymbol{\phi_i}$ denotes the parameters of the distribution of $\mathbf{z}_i$. E.g., $\boldsymbol{\phi}_i = \{\mu_i, \sigma_i\}$ if the i-th variational factor is a Gaussian. By factorising, the mean-field approximation does not capture any correlations between weights,

Figure 2.5: **Left:** Bi-modal posterior approximated using a single Gaussian. **Right:** Contour plot of a non-diagonal Gaussian approximated using a diagonal Gaussian.

and it is not clear exactly how much information is lost by ignoring all correlations between latent variables in the model. Mean-field variational inference widely used due to its simplicity.

The sacrifice of correlations is one of the reasons why variational approximations are known to underestimate the variance. How information about the true posterior is lost by using a mean-field approximation is illustrated in one and two dimensions in Figure 2.5. In the two-dimensional case, a bi-variate Gaussian with a non-diagonal covariance matrix is approximated by a factorised Gaussian distribution. The mean of the true posterior is accurately recovered, but the variance is underestimated because correlations are ignored by the factorised variational distribution. In the one-dimensional case, the dominant mode of the true posterior is fitted closely, but the second mode is ignored.

In the both cases in Figure 2.5, the variational approximation will underestimate the true variance and not reflect the behaviour of the true posterior. In the 1D case because of the second mode being ignored, and in the 2D case because the correlations are ignored. The backward Kullback Leibler divergence also contributes to encourage this behaviour because there is no penalty for letting the variational posterior be zero in regions where the true posterior is non-zero. This leads to potentially large portions of the true posterior not being approximated by the variational posterior, often referred to as the *mode-seeking* or *zero forcing* behaviour.

One last drawback of ELBO-based variational inference worth mentioning here, is that because the ELBO merely is a lower bound on the marginal log-likelihood, it does not necessarily have the same maxima as the true log-likelihood, leading to biased maximum likelihood estimates. The maximum a posteriori parameter estimates also tend to be biased, because we have no guarantees that the variational density fits the dominant mode of the true posterior. Some of these unfavourable behaviours can be mended by using a different divergence measure between the approximating and true densities, e.g., by using Rényi α-divergence for variational inference as has been proposed in (Li and Turner, 2016; Hernandez-Lobato et al., 2016). Finding alternative objective function for variational inference is an active line of research, but these methods have yet to gain significant traction, and we will not describe them more in detail here.

Amortised variational inference

We revisit the mean-field expression for the density over the latent variables given in Equation (2.12):

$$q_{\boldsymbol{\phi}}(\mathbf{z}) = \prod_{i=1}^{M} q_{\boldsymbol{\phi}_i}(z_i) \tag{2.13}$$

If we were optimise the variational parameters over a dataset $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ naively, we would assign a set of local parameters $\boldsymbol{\phi}_i$ to each of the N observations with no parameter sharing across data points. It follows that the set of variational parameters $\boldsymbol{\phi} = \{\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_N\}$ grows linearly with the number of data. This does not scale when the number of data is large. It also requires the computation of new variational parameters by maximising for each new data point we want to evaluate at test time.

This is where the idea of *inference networks* (Kingma and Welling, 2013; Rezende et al., 2014) comes in. Instead of keeping the variational parameters associated with each data point stored in memory,

we train a neural (inference) network that takes in a data point $\mathbf{x}_*$ and returns its local variational parameters $\boldsymbol{\phi}_*$. This reduces the optimisation procedure to training the parameters of the neural network, and after training we only have to store the parameters of this inference network in memory. These are global parameters, as we use the same network parameters for all data, also at test time and Using an inference network is strictly less expressive than assigning local parameters to each data point, but the number of parameters saved is huge, particularly when working with big datasets. The use of an inference network then allows for scalable and tractable inference.

2.4 Likelihood training and the Kullback-Leibler divergence

We have already shown how to perform approximate Bayesian inference using the reverse Kullback-Leibler divergence. In this section, we show that density estimation through maximum likelihood training can be viewed as minimising the *forward* Kullback-Leibler divergence. By doing so, we show how the KL divergence relates to both of the main problems covered by this thesis, namely variational inference and density estimation. The argument goes as follows:

Suppose we have a set of i.i.d. samples $\{\mathbf{x}_1, \dots, \mathbf{x}_N\}$ coming from $p_*(\mathbf{x})$. In density estimation, we want to approximate the true density $p_*(\mathbf{x})$ by some parameterised density $p_\theta(\mathbf{x})$. The parameters of this density are learned by maximising the likelihood of the observations under $p_\theta(\cdot)$:

$$
\begin{aligned}
\boldsymbol{\theta}_{\text{MLE}} &= \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}(x_i) \\
&= \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}(x_i) - \sum_{i=1}^{N} \log p_*(x_i) \\
&= \arg\max_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} \log \frac{p_{\boldsymbol{\theta}}(x_i)}{p_*(x_i)} \\
&\xrightarrow{N\to\infty} \arg\min_{\boldsymbol{\theta}} \int_x p_*(x) \log \frac{p_*(x)}{p_{\boldsymbol{\theta}}(x)} dx =: \mathrm{KL}(p_*(\mathbf{x}) \mid\mid p_{\boldsymbol{\theta}}(\mathbf{x}))
\end{aligned}
$$

where the last line follows from *the law of large numbers*. From this observation it that maximum likelihood training implicitly tries to fit a parameterised distribution to the true density $p_*(\mathbf{x})$ by minimising the forward Kullback-Leibler divergence. This further motivates the use of likelihood as a principled training objective for machine learning models.

Chapter 3

Normalising Flows

In this chapter, we introduce the foundational theory about normalising flows, and show how normalising flows are useful in density estimation and variational inference. We proceed to discuss how to design a tractable normalising flow, and provide two general strategies for how to make a transformation that is suitable for a flow. Finally, we present a simple flow from (Rezende and Mohamed, 2015), and conduct a set of illustrative experiments to help build up intuition and showcase its flexibility. This chapter does not assume any prior knowledge of normalising flows, and is aimed to be so self-contained that it can serve as an introduction to normalising flows on its own

3.1 Motivation

Normalising flows provide a general approach for defining expressive probability distributions, using only a simple base distribution that allows for easy sampling and density evaluations and a set of bijective transformations. Modern normalising flows combine the flexibility and learning capacity of neural networks with prior about the structure of the data to define these transformations. Distributions defined using normalising flows can be used for modelling, inference, and sampling, and have uses within a wide range of areas. Most research has gone into designing normalising flows for variational inference and density estimation, including generative modelling of images and audio.

In **variational inference**, an intractable posterior density is approximated by a member of some tractable variational family. As discussed in Section 2.3.4, the performance of the model depends on how well the approximate posterior approximates the true posterior, which is directly related to the flexibility of the variational family. When we use a limited class of variational densities like a diagonal Gaussian, we sacrifice a lot of this flexibility at the altar of computational convenience. To obtain an estimate of the posterior that is as close to the true posterior as possible, we want a variational that is highly flexible – preferably flexible enough to capture the true posterior density.

A promising line of research in this regard builds on normalising flows, which were popularised in a deep learning context by Rezende and Mohamed (2015), and later improved by others, most notably Kingma et al. (2016) and van den Berg et al. (2018). The core idea is to make repeated use of bijective and differentiable transformations to transform a simple base density into an increasingly complex target density that approaches the true posterior density. The parameters of the transformations are learned by maximising the ELBO through gradient-based training. Normalising flows can define a richer variational family that improves the posterior estimates in variational inference.

The other main application for normalising flows, is **density estimation**, as first proposed by Tabak and Turner (2013). The idea is the same as when using normalising flow for variational inference; to learn a bijective mapping between some base density and a more complex target density, which is represented by a set of samples. The parameters of the density estimator are learned by maximum likelihood training. We are able to evaluate and generate new samples from this density thanks to the change of variables formula, which lets us evaluate the exact likelihood of a given sample under the density that is defined by the flow from the base density.

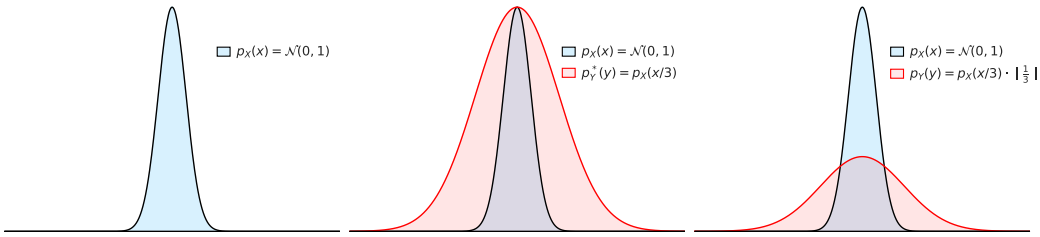


Figure 3.1: **Left:** Untransformed Gaussian density. **Middle:** Original and transformed, but not yet normalised density. **Right:** Original and transformed density.

3.2 The change of variables theorem

The change of variables formula for probability is the backbone of normalising flows. Given a (scalar or vector) random variable X with a known probability density function, and a differentiable and bijective transformation $f : X \to Y$, the formula lets us compute the density of the transformed random variable $Y = f(X)$. The theorem (Evans and Rosenthal, 2009) is given as:

Theorem. ***(Change of Variables Theorem)*** *Let Ω_X, Ω_Y be open subsets of $\mathbb{R}^D$ and let $f : \Omega_X \mapsto \Omega_Y$ be a bijective and differentiable map. Let X be a random variable taking values in Ω_X, and let X have density p_X with respect to the Lebesgue measure on Ω_X.*

Then, $Y = f(X)$ has density:

$$p_Y(y) = p_X(f^{-1}(y)) \cdot \left| \det \frac{\partial f^{-1}(y)}{\partial y} \right| \tag{3.1}$$

with respect to the Lebesgue measure on Ω_Y.

For one-dimensional densities, Equation (3.1) amounts to:

$$p_Y(y) = p_X(f^{-1}(y)) \cdot \left| \frac{d}{dy} f^{-1}(y) \right|, \tag{3.2}$$

where f is a strictly monotone[1] transformation of the random variable x. Intuitively, the last factor of Equation (3.2) compensates for the change of area under the density function imposed by the transformation, by scaling the transformed density function up or down accordingly. This scaling ensures that the probability mass is invariant under the change of variables, so that the density function still integrates to 1 over its support and remains a valid probability density after transformation. An illustration of this process is depicted in Figure 3.1 for a transformation $Y = 3X$ where X follows a univariate standard Gaussian distribution.

In the following, we operate in a multivariate setting and denote random vectors by bold lowercase letters, as this is the convention in the literature on normalising flows. Using this notation, we rewrite Equation (3.1), so that the transformed random vector $\mathbf{y} = f(\mathbf{x})$ has a density function given by:

$$\begin{aligned} p_Y(\mathbf{y}) &= p_X(f^{-1}(\mathbf{y})) \cdot \left| \det \frac{\partial f^{-1}(\mathbf{y})}{\partial \mathbf{y}} \right| \\ &= p_X(\mathbf{x}) \cdot \left| \det \left(\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right)^{-1} \right| \\ &= p_X(\mathbf{x}) \cdot \left| \det \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right|^{-1}. \end{aligned} \tag{3.3}$$

The second equality follows from the *inverse function theorem* which states that the derivative of the inverse function f^{-1} at the point $\mathbf{y}$ equals the reciprocal of the derivative of f at the inverse image

[1] The change of variables theorem on a slightly different form is applicable also to transformations that are not strictly monotonic. We only provide the result for strictly monotonic transformations here, as this is the theory that is relevant for understanding normalising flows.

point $\mathbf{x} = f^{-1}(\mathbf{y})$. The last equality follows from a property of determinants[2]. The term $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$ is the Jacobian matrix of the vector-valued function $f(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_D(\mathbf{x}))^T$ with elements:

$$\left(\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}\right)_{ij} = \frac{\partial f_i(\mathbf{x})}{\partial x_j}.$$

The advantage of rewriting the change of variables theorem to the form shown in Equation 3.3, is that it becomes clearer how to compute the density of $\mathbf{y}$. In particular, we do not need to compute the magnitude of the Jacobian of $f^{-1}(\mathbf{y})$ with respect to $\mathbf{y}$. Instead, we rather compute the Jacobian of $f(\mathbf{x})$ with respect to $\mathbf{x}$ directly, and take the reciprocal of its magnitude. This factor is analogous to the derivative factor from Equation (3.2). While the absolute value of the derivative accounts for change in area imposed by the transformation, the absolute value of the magnitude of the Jacobian more generally accounts for the change in *volume* induced by the transformation.

3.3 Normalising flows

Normalising flows make use of the change of variables theorem to transform a simple density into a more complex target density by applying multiple differentiable and bijective transformations in sequence. Each transformation is *normalising* in the sense that it outputs a valid, normalised density. Repeated application of thoughtfully designed transformations can capture increasingly complex mappings between two densities, and the path traversed by the initial density through the sequence of transformations is what we refer to as the *flow*.

To make a normalising flow, we start out with a random variable $\mathbf{u}$ in $\mathbb{R}^D$ coming from some simple density $p_{\mathbf{u}}(\mathbf{u})$ for which we are able to evaluate the likelihood and generate samples. We refer to this simple density as the *base density*, and it is generally assumed to be a standard Gaussian density. The parameters and shape of the base density are intuitively of lesser importance, as the transformations in a normalising flow can shift, scale, and transform the base density arbitrarily many times in order to approximate the desired target density. That being said, the role of the base density is flagged as an open question by Kobyzev et al. (2019), as they point out that the choice of base density may affect the complexity of the resulting transformations, and how easily they are learned.

Applying a differentiable and bijective transformation to $\mathbf{u}$ gives us a new random variable $\mathbf{x} = f(\mathbf{u})$. Using Equation (3.3), we find that the density function of $\mathbf{x}$ is given by

$$p_{\mathbf{x}}(\mathbf{x}) = p_{\mathbf{u}}(\mathbf{u}) \cdot \left| \det \frac{\partial f(\mathbf{u})}{\partial \mathbf{u}} \right|^{-1}.$$

One transformation does not make a flow, and a normalising flow is obtained by composing several such transformations in sequence. It is a key point that as long as both transformations f_1 and f_2 satisfy the criteria in the change of variables theorem, so does the composition of the two. In particular, if we have $\mathbf{x} = f_2 \circ f_1(\mathbf{u})$, the inverse transformation is given by:

$$\mathbf{u} = f_1^{-1} \circ f_2^{-1}(\mathbf{x}),$$

and the determinant of the Jacobian is given by

$$\det \frac{\partial \mathbf{x}}{\partial \mathbf{u}} = \det \frac{\partial f_2 \circ f_1(\mathbf{u})}{\partial \mathbf{u}} = \det \frac{\partial f_2(f_1(\mathbf{u}))}{\partial f_1(\mathbf{u})} \cdot \det \frac{\partial f_1(\mathbf{u})}{\partial \mathbf{u}}.$$

To motivate the necessity of applying several transformations, it is helpful to understand each transformation as a local expansion or contraction of the initial density that only affects a small region of the D-dimensional space. Applying more transformations allows us to transform more of the original space, and we can approximate increasingly complex densities by applying more transformations. To keep track of the different transformations in the flow, we overload the subscript of $f_i(\cdot)$. The subscript i now refers to the i-th transformation in the sequence, and we can write out the transformed variable as

$$\mathbf{x} := \mathbf{x}_K = f_K(\mathbf{x}_{K-1}) = f_K \circ \cdots \circ f_1(\mathbf{u}). \tag{3.4}$$

The equation above shows that we can generate a sample from the final iterate of the transformed density by generating a sample from the base density and simply passing it through the sequence

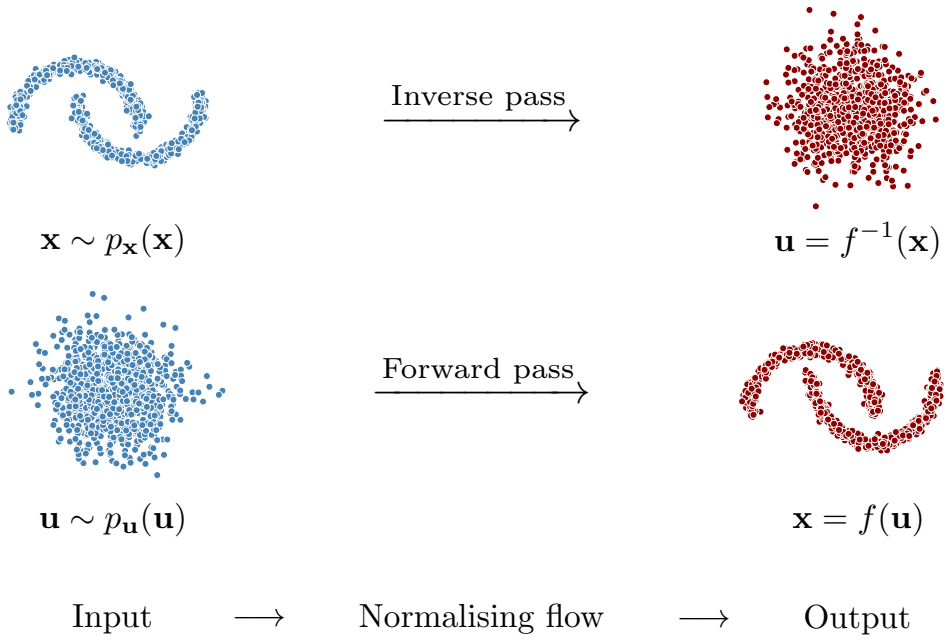


Figure 3.2: The forward pass and the inverse pass of a normalising flow.

of transformations. This is referred to as the *noise to data* transformation, or as the *forward pass*. Denoting the density after k transformations as $p_k(\mathbf{x})$, the transformed density is given by

$$p_K(\mathbf{x}_K) = p_{K-1}\left(\mathbf{x}_{K-1}\right) \cdot \left|\det \frac{\partial f_K}{\partial \mathbf{x}_{K-1}}\right|^{-1} = \cdots = p_{\mathbf{u}}(\mathbf{u}) \cdot \prod_{i=1}^{K} \left|\det \frac{\partial f_k}{\partial \mathbf{x}_{k-1}}\right|^{-1}, \tag{3.5}$$

where we let $\mathbf{x}_0 = \mathbf{u}$. To evaluate the likelihood of a sample under $p_K(\mathbf{x}_K)$ according to Equation (3.5), we first have to calculate its pre-image under the base density as

$$\mathbf{u} = f_1^{-1} \circ f_2^{-1} \circ \cdots \circ f_K^{-1}(\mathbf{x}_K),$$

using what we call the *data to noise* transformation, or the *inverse pass* through the flow. See Figure 3.2 for an illustration of the forward pass and the inverse pass. In practice, not all flows are invertible everywhere in the $\mathbf{x}_K$ space, meaning it is not always possible to compute the likelihood of external samples. We will illustrate this in Section 3.8 in our discussion on planar flows.

Finally, we can use *the law of the unconscious statistician* to compute the expectation of any function $h(\mathbf{x}_K)$ under $p_K(\mathbf{x})$ without knowing the density of the random variable $h(\mathbf{x}_K)$ explicitly. The expectation of any function $h(\mathbf{x}_K)$ can be taken as an expectation with respect to the base density, and be computed using only the forward pass through the sequence of transformations

$$\mathrm{E}_{q_K}\left(h(\mathbf{x}_K)\right) = \mathrm{E}_{p_{\mathbf{u}}(\mathbf{u})}(h(f_K \circ \cdots \circ f_1(\mathbf{u}))) \,.$$

3.4 Expressivity of normalising flows

In this section, we repeat the argument presented in (Papamakarios et al., 2019) showing that even when restricted to a simple base density, we are under some mild assumptions able to represent any density $p_X(\mathbf{x})$ using a normalising flow:

Suppose that $p_X(\mathbf{x}) > 0$ for all $\mathbf{x} \in \mathbb{R}^D$, and that all conditional probabilities $P(X_i \leq x_i \mid \mathbf{x}_{<i})$ are differentiable with respect to x_i and $\mathbf{x}_{<i}$. The notation $\mathbf{x}_{<i}$ refers to all elements of the vector $\mathbf{x}$ with

[2]Suppose $\mathbf{M}$ is an invertible matrix. Then, $\det(\mathbf{I}) = \det\left(\mathbf{M}\mathbf{M}^{-1}\right) = \det\left(\mathbf{M}\right)\det\left(\mathbf{M}^{-1}\right) = 1$, so $\det\left(\mathbf{M}^{-1}\right) = \frac{1}{\det(\mathbf{M})}$

index less than i. The density $p(\mathbf{x})$ can then be factorised using the probability product rule as

$$p_X(\mathbf{x}) = \prod_{i=1}^{D} p_X(x_i \mid \mathbf{x}_{<i}). \tag{3.6}$$

Define the cumulative distribution function of each conditional random variable as F_i such that

$$F_i(x_i \mid \mathbf{x}_{<i}) = P(X_i \leq x_i \mid \mathbf{x}_{<i}) = \int_{-\infty}^{x_i} p_X(x_i' \mid \mathbf{x}_{<i}) dx_i'.$$

Consider now the vectorised transformation $F : \mathbf{x} \mapsto \mathbf{z} \in (0,1)^D$ such that the i-th element of F is the transformation F_i acting on the i-th element of $\mathbf{x}$. Because each transformation F_i is differentiable with respect to x_i and $\mathbf{x}_{<i}$, F is differentiable with respect to $\mathbf{x}$. Because $\partial F_i / \partial x_j = 0$ for $i < j$, the Jacobian of the transformation will be lower triangular, and its determinant will be the product of its diagonal elements:

$$\det \frac{\partial F(\mathbf{x})}{\partial \mathbf{x}} = \prod_{i=1}^{D} \frac{\partial F_i}{\partial x_i} = \prod_{i=1}^{D} p(x_i \mid \mathbf{x}_{<i}) = p_X(\mathbf{x}).$$

Because the determinant of the transformation F is equal to $p_X(\mathbf{x}) > 0$, the transformation F is clearly invertible. The density function of the transformed random variable $\mathbf{z} = F(\mathbf{x})$ is according to the change of variables theorem given as:

$$p_Z(\mathbf{z}) = p_X\left(F^{-1}(\mathbf{z})\right) \cdot \left| \det \frac{\partial F^{-1}(\mathbf{z})}{\partial \mathbf{z}} \right| = p_X(\mathbf{x}) \cdot \left| \det \frac{\partial F(\mathbf{x})}{\partial \mathbf{x}} \right|^{-1} = p_X(\mathbf{x}) \cdot |p_X(\mathbf{x})|^{-1} = 1.$$

This implies that $\mathbf{z}$ has a uniform density in $(0,1)^D$, so that a flow-based model can express any density $p_X(\mathbf{x})$ in terms of a uniform base density on the unit cube. Note that because F is differentiable and invertible, the invertibility and differentiability of F^{-1} are both ensured by the inverse function theorem.

This argument can be extended further in a similar manner to include mappings to different base densities than the uniform. To do so, we use the mapping between $\mathbf{x}$ and $\mathbf{z}$ as defined above as an intermediate step. We write out the case of a Gaussian base density below:

Assume we have a random vector $\mathbf{y} \in \mathbb{R}^D$ following a multivariate Gaussian density $p_Y(\mathbf{y})$ so that its density is strictly greater than zero everywhere in $\mathbb{R}^D$. Because the conditional densities of a multivariate Gaussian density are also Gaussian (Härdle and Simar, 2015, Theorem 5.3), we can safely assume that all conditional probabilities are differentiable with respect to $(y_i, \mathbf{y}_{<i})$. Decomposing $p_Y(\mathbf{y})$ as we decomposed $p_X(\mathbf{x})$ above, we can define a map $G : \mathbf{y} \mapsto \mathbf{z}$ where G satisifies the same properties as F above. Hence, we have:

$$p_Z(\mathbf{z}) = p_Y\left(G^{-1}(\mathbf{z})\right) \cdot \left| \det \frac{\partial G^{-1}}{\partial \mathbf{z}} \right| = p_Y(\mathbf{y}) \cdot \left| \det \frac{\partial G}{\partial \mathbf{y}} \right|^{-1} = p_Y(\mathbf{y}) \cdot |p_Y(\mathbf{y})|^{-1} = 1$$

Thus, we have $\mathbf{x} = \left(F^{-1} \circ G\right)(\mathbf{y})$, showing that we can express $p_X(\mathbf{x})$ in terms of an arbitrary Gaussian base density, including the standard Gaussian. Note that we have only showed that such mappings between densities exist, not necessarily implying that we will be able to find them. In practice, we can not directly choose the particular transformations F and G when designing a flow, as these are unknown. Instead, we will approximate the flow $F^{-1} \circ G$ by composing multiple simpler transformations that each has a pre-specified parameterised form. In the following, we will elaborate on how to design such transformations.

3.5 Designing finite flows

To obtain a flow with the properties described in Section 3.3 that is also computationally feasible to use, we have to impose some restrictions on the transformations we use. Given that we have chosen some simple base distribution for which we can evaluate the likelihood and generate samples from, we ideally want differentiable and bijective transformations which satisfy:

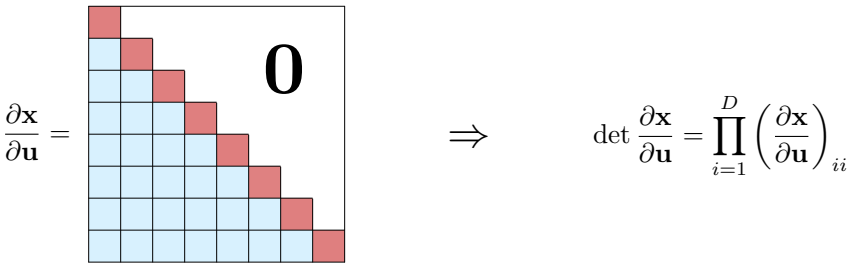


Figure 3.3: A lower triangular Jacobian has an easily computable determinant.

- efficient to compute the forward transformation (for sampling)
- efficient to compute the inverse transformation (for likelihood evaluation)
- efficient to compute the Jacobian and its determinant
- efficient sampling and likelihood evaluation under the base distribution
- differentiable with respect to its parameters to allow for gradient-based training,

while at the same time maintaining high expressive power in a reasonable number of transformations.

A central challenge in designing practical flows is to create transformations that have a tractable Jacobian, efficient forward pass, and efficient inverse pass at the same time. Though some notable exceptions exist (Dinh et al., 2014, 2017; Kingma and Dhariwal, 2018), the majority of flow-based models do not allow for both efficient forward pass and efficient inverse pass. Depending on the problem we want to solve using normalising flows, it is in many cases sufficient to have either efficient sampling, *or* to have efficient likelihood evaluation. We will explain this more in depth in Chapter 5, and provide examples of flows of both types.

To design a transformation with a tractable det-Jacobian is crucial for the computational complexity of the flow. Computing the determinant of a $D \times D$ matrix is generally cubic in D, which is prohibitively expensive considering the typical dimensionality of the input. One way to reduce the computational complexity of this computation, is to design the transformation such that the determinant of the Jacobian is equal to one, as is done in *non-linear independent component estimation (NICE)* (Dinh et al., 2014). This subclass of normalising flows is called *volume preserving* flows, since a det-Jacobian equal to one implies that the transformation does not lead to a change in volume.

However, the flexibility of volume preserving transformations is very limited, and consequently, more research has gone into designing transformations with *triangular* Jacobians. The computational complexity of the determinant of a triangular matrix is only linear with respect to its dimensionality, because the determinant of a triangular matrix is equal to the product of its diagonal elements. This is depicted in Figure 3.3 for a lower-triangular Jacobian. Transformations with lower-triangular Jacobians are usually achieved by using either an *autoregressive* transformation, or by using a *coupling layer*:

- An autoregressive flow (Kingma et al., 2016; Papamakarios et al., 2017; Huang et al., 2018) is composed by transformations $\mathbf{y} = f(\mathbf{x})$ where each each entry of the output is only conditioned on the previous entries of the input, such that $y_i = f(x_i; \theta(\mathbf{x}_{1:i-1}))$ for some function $\theta(\cdot)$ parameterising the transformation. Autoregressive flows is a main topic in Chapter 4 and 5, where the details of these flows will be presented.
- In a coupling layer (Dinh et al., 2014, 2017), the input vector $\mathbf{x} \in \mathrm{R}^D$ is partitioned into two disjoint parts $\mathbf{x}_{1:d}$ and $\mathbf{x}_{d+1:D}$. The transformation $\mathbf{y} = f(\mathbf{x})$ is defined such that d components remain the same, $\mathbf{y}_{1:d} = \mathbf{x}_{1:d}$, while the remaining components are transformed as $y_i = f(x_i; \theta(\mathbf{x}_{1:d}))$. The properties of the coupling layer depend on how we design this transformation. Possible choices for the coupling layer are additive and affine transformations, and a model based on the latter is presented in Chapter 5.

All the normalising flows we consider in this thesis are composed of a finite sequence of transformations, making them instances of *finite* flows. This is opposed to *infinitesimal*, or *continuous-time* flows (Rezende and Mohamed, 2015; Papamakarios et al., 2019), where the number of transformations in the flow tends to infinity. In this case, the transformation of the base density is no longer described by discrete transformations, but by a parameterised ordinary differential equation (ODE) $\frac{\partial \mathbf{z}_t}{\partial t} = g_{\boldsymbol{\phi}}(t, \mathbf{z}_t)$. To compute the transformed variable at t_1, we need to compute $\mathbf{z}_{t_1} = \mathbf{z}_{t_0} + \int_{t_0}^{t_1} g_{\boldsymbol{\phi}}(t, \mathbf{z}_t)dt$ using an ODE solver. The bulk of normalising flows literature is on finite flows, and we consider continuous-time flows to be beyond the scope of this thesis.

3.6 Normalising flows for density estimation

Normalising flows provide expressive transformations and exact likelihood evaluation, making them suitable for use in density estimation. Using normalising flows for density estimation was first proposed by Tabak and Turner (2013), whose work despite being less known[3] actually preceded, and laid the foundations of the work of Rezende and Mohamed (2015) and later publications on normalising flows.

Given a set of observations $\{\mathbf{x}_1, \dots, \mathbf{x}_N\}$ we want to approximate the density function of the unknown generative process $p^*_{\mathbf{x}}(\mathbf{x})$ using a normalising flow. To do so, we want to learn the inverse transformation $\mathbf{u} = f^{-1}(\mathbf{x})$ that goes in the direction from data to noise, as this is the transformation that defines the estimated density model $p_{\mathbf{x}}(\mathbf{x})$. The parameters of this inverse flow are trained using maximum likelihood estimation, i.e., by maximising the likelihood of the data:

$$\mathrm{E}_{p^*_{\mathbf{x}}(\mathbf{x})}\left(p_{\mathbf{x}}(\mathbf{x})\right) = \mathrm{E}_{p^*_{\mathbf{x}}(\mathbf{x})}\left(p_{\mathbf{u}}(f^{-1}(\mathbf{x})) \cdot \left|\det \frac{\partial f^{-1}}{\partial \mathbf{x}}\right|\right)$$

The above objective function is maximised by using Monte Carlo approximations over mini-batches to maximise the average log-likelihood during training:

$$\begin{aligned}
\mathrm{E}_{p^*_{\mathbf{x}}(\mathbf{x})}\left(\log p_{\mathbf{x}}(\mathbf{x})\right) &\approx \frac{1}{N}\sum_{i=1}^{N} \log\left(p_{\mathbf{u}}(f^{-1}(\mathbf{x}_i)) \cdot \left|\det \frac{\partial f^{-1}}{\partial \mathbf{x}_i}\right|\right) \\
&= \frac{1}{N}\sum_{i=1}^{N}\left(\log p_{\mathbf{u}}(\mathbf{u}_i) + \sum_{k=1}^{K} \log\left|\det \frac{\partial f_k^{-1}(\mathbf{x}_{i,k})}{\partial \mathbf{x}_{i,k}}\right|\right) \\
&= \frac{1}{N}\sum_{i=1}^{N}\left(\log p_{\mathbf{u}}(\mathbf{u}_i) - \sum_{k=1}^{K} \log\left|\det \frac{\partial f_k(\mathbf{x}_{i,k-1})}{\partial \mathbf{x}_{i,k-1}}\right|\right)
\end{aligned}$$

where $\mathbf{x}_{i,0} = \mathbf{u}$. The likelihood is evaluated repeatedly during training, so the computations of the inverse transformations and the determinant of the Jacobian have to be efficient in order for the training to be efficient. In fact, sampling from $p_{\mathbf{x}}(\mathbf{x})$, i.e., computing the forward pass $f(\mathbf{u})$, does not have to be practically feasible in order to fit a flow model using maximum likelihood and use it for density estimation. For density estimators, we are therefore willing to sacrifice efficient sampling for efficient likelihood evaluation, if such a trade-off has to be made (Papamakarios et al., 2017).

3.7 Normalising flows for variational inference

The flexibility and tractability of normalising flows also make them suitable for defining approximate posterior densities in variational inference. In this section, we will elaborate on how to train a normalising flow for variational inference (Rezende and Mohamed, 2015; Kingma et al., 2016; van den Berg et al., 2018) by maximising the evidence lower bound when using a normalising flow to parameterise the approximate posterior density over the latent variables. We again adopt the notation of the variational inference literature, with $q(\cdot)$ referring to the approximating density, and $\mathbf{z}$ referring to the (latent) variable of interest. If we model the posterior using a flow of length K, we then have $q_\phi(\mathbf{z}) = q_K(\mathbf{z}_K)$. Using the law of the unconscious statistician, the evidence lower bound can be

[3] 48 vs. 619 citations on Google Scholar as of 7 October 2019

rewritten as an expectation with respect to the initial density $q_0(\mathbf{z}_0)$, that is:

$$
\begin{aligned}
-\text{ELBO} &= -\int_{\mathbf{z}} \log\left(\frac{p(\mathbf{x} \mid \mathbf{z})p(\mathbf{z})}{q_\phi(\mathbf{z})}\right) q_\phi(\mathbf{z}) d\mathbf{z} \\
&= \mathrm{E}_{q_\phi(\mathbf{z})}\left(\log q_\phi(\mathbf{z})\right) - \mathrm{E}_{q_\phi(\mathbf{z})}\left(\log\left(p(\mathbf{x} \mid \mathbf{z})p(\mathbf{z})\right)\right) \\
&= \mathrm{E}_{q_K(\mathbf{z}_K)}\left(\log q_K(\mathbf{z}_K)\right) - \mathrm{E}_{q_K(\mathbf{z}_K)}\left(\log p(\mathbf{x}, \mathbf{z}_K)\right) \\
&= \mathrm{E}_{q_0(\mathbf{z}_0)}\left(\log q_0(\mathbf{z}_0) - \sum_{k=1}^{K} \log\det\left|\frac{\partial f_k}{\partial \mathbf{z}_{k-1}}\right|\right) - \mathrm{E}_{q_0(\mathbf{z}_0)}\left(\log p(\mathbf{x}, \mathbf{z}_0)\right) \\
&= \mathrm{E}_{q_0(\mathbf{z}_0)}\left(\log q_0(\mathbf{z}_0)\right) - \mathrm{E}_{q_0(\mathbf{z}_0)}\left(\sum_{k=1}^{K} \log\det\left|\frac{\partial f_k}{\partial \mathbf{z}_{k-1}}\right|\right) - \mathrm{E}_{q_0(\mathbf{z}_0)}\left(\log p(\mathbf{x}, \mathbf{z}_0)\right)
\end{aligned}
$$

This objective function is optimised using a Monte Carlo approximation as described in Chapter 2.3.3, using samples from the base density $q_0(\mathbf{z}_0)$. The parameters of each transformation in the flow are learned by minimising the negative ELBO using gradient-based optimisation.

The complexity of the true posterior is usually not known a priori, but a more flexible variational approximation will in any case bring us closer to the true posterior behaviour because it makes the ELBO a tighter bound on the likelihood. Normalising flows can be used to parameterise more complex densities in this regard, giving more accurate and reliable predictions. The chances of capturing the true behaviour increase if we choose a more flexible family for the approximating distribution. However, there will still be an implicit bias-variance trade-off in the choice of approximating family, because increased expressivity (more layers) comes at the cost of an increased computational cost.

3.8 Experimenting with planar flows

So far, we have written about normalising flows in general terms without explicitly specifying the transformations in the flow. In this section, we present the *planar flow*, a flow proposed by Rezende and Mohamed (2015) for variational inference, and reproduce the results from Figure 3 in (Rezende and Mohamed, 2015). We proceed to investigate the properties of the planar flow more in-depth than what was done in the original paper, by dissecting the model to look at each of the planar transformations in the flow individually. These experiments provide insights and intuitions about the inner workings of normalising flows, and we clearly motivate why longer flows are needed to approximate complex densities. These intuitions are valid not only for planar flows, but apply also to other classes of normalising flows.

3.8.1 The planar flow

Rezende and Mohamed (2015) proposed two normalising flows in a variational inference setting, called *planar* and *radial flows*. Both transformations have known det-Jacobians that are computable in linear time, but the transformations are also fairly limited and not widely used in modern normalising flows models. We present the planar flow here nevertheless, because it serves as an illustrative example that can help the reader gain intuition about what normalising flows are, and what they do.

Planar flows are compositions of transformations of the form:

$$
f(\mathbf{z}) = \mathbf{z} + \mathbf{u}h(\mathbf{w}^T\mathbf{z} + b) \tag{3.7}
$$

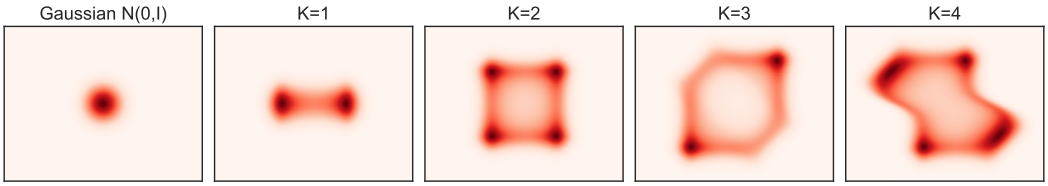


Figure 3.4: Transformed densities after $K = 1, 2, 3, 4$ planar transformations of a standard Gaussian.

Table 3.1: Potential functions with $w_1(\mathbf{z}) = \sin\left(\frac{2\pi z_1}{4}\right)$, $w_2(\mathbf{z}) = 3\exp\left(-\frac{1}{2}\left(\frac{z_1-1}{0.6}\right)\right)$, and $w_3(\mathbf{z}) = 3\sigma\left(\frac{z_1-1}{0.3}\right)$ where $\sigma(\cdot)$ is the sigmoid function and $\mathbf{z}$ is a sample from a standard bivariate Gaussian.

Potential functions $U(\mathbf{z})$
$U_1(\mathbf{z}) = \frac{1}{2}\left(\frac{\lVert\mathbf{z}\rVert_2 - 2}{0.4}\right)^2 - \log\left(e^{-\frac{1}{2}\left[\frac{z_1-2}{0.6}\right]^2} + e^{-\frac{1}{2}\left[\frac{z_1+2}{0.6}\right]^2}\right)$
$U_2(\mathbf{z}) = \frac{1}{2}\left[\frac{z_2 - w_1(\mathbf{z})}{0.4}\right]^2$
$U_3(\mathbf{z}) = -\log\left(e^{-\frac{1}{2}\left[\frac{z_2 - w_1(\mathbf{z})}{0.35}\right]^2} + e^{-\frac{1}{2}\left[\frac{z_2 - w_1(\mathbf{z}) + w_2(\mathbf{z})}{0.35}\right]^2}\right)$
$U_4(\mathbf{z}) = -\log\left(e^{-\frac{1}{2}\left[\frac{z_2 - w_1(\mathbf{z})}{0.4}\right]^2} + e^{-\frac{1}{2}\left[\frac{\mathbf{z}_2 - w_1(\mathbf{z}) + w_3(\mathbf{z})}{0.35}\right]^2}\right)$
$U_5(\mathbf{z}) = \frac{1}{2}\left(\frac{\lVert\mathbf{z}\rVert_2 - 4}{0.4}\right)^2 - \log\left(e^{-\frac{1}{2}(\frac{z_1-2}{0.8})^2} + e^{-\frac{1}{2}(\frac{z_1+2}{0.8})^2}\right)$

where $\mathbf{u}, \mathbf{w} \in \mathbb{R}^D$ and the scalar b are the learnable parameters of the transformation, and $h(\cdot)$ is some smooth and differentiable element-wise non-linearity. The planar flow belongs to the class of *residual flows* because the transformation in Equation (3.7) bears similarities to the skip-connections used in residual neural networks (He et al., 2015). Each planar transformation contracts or expands the density of $\mathbf{z}$ perpendicularly to the hyperplane defined by $\mathbf{w}^T\mathbf{z} + b = 0$, hence the name *planar flow*. This nature of the planar transformations is displayed in Figure 3.4 for $K = 4$ manually (and somewhat arbitrarily) specified transformations of a standard Gaussian density.

The Jacobian of a planar transformation is given by

$$\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = \mathbf{I} + \mathbf{u}h'(\mathbf{w}^T\mathbf{z} + b)\mathbf{w}^T$$

Using the *matrix determinant lemma:* $\det\left(\mathbf{A} + \mathbf{u}\mathbf{v}^T\right) = \left(1 + \mathbf{v}^T\mathbf{A}^{-1}\mathbf{u}\right)\det(\mathbf{A})$, with $\mathbf{A} = \mathbf{I}$, $\mathbf{u} = \mathbf{u}$, and $\mathbf{v} = h'(\mathbf{w}^T\mathbf{z} + b)\mathbf{w}$, we get the determinant of the Jacobian

$$\begin{aligned}\left|\det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}}\right| &= \left|(1 + h'(\mathbf{w}^T\mathbf{z} + b)\mathbf{w}^T\mathbf{I}^{-1}\mathbf{u})\det\mathbf{I}\right| \\ &= \left|1 + h'(\mathbf{w}^T\mathbf{z} + b)\mathbf{w}^T\mathbf{u}\right|\end{aligned}$$

which can be computed in linear time $\mathcal{O}(D)$, where D is the dimensionality of $\mathbf{z}$.

3.8.2 Showcasing the flexibility of normalising flows

In the first experiment, we follow the approach of Rezende and Mohamed (2015) and approximate four two-dimensional non-Gaussian densities with a planar flow using a bivariate standard Gaussian as the base density $q_0(\mathbf{z})$. A model of length K is made by stacking K planar transformations as defined in Equation (3.7) in sequence. Each transformation has learnable parameters, and the model is trained by updating the parameters to minimise the negative evidence lower bound:

$$-\text{ ELBO} = \mathrm{E}_{q_0(\mathbf{z}_0)}\left(\log q_0(\mathbf{z}_0) - \sum_{k=1}^{K}\log\left|1 + h'(\mathbf{w}^T\mathbf{z}_{k-1} + b)\mathbf{w}^T\mathbf{u}\right| - \log p(\mathbf{x}, \mathbf{z}_K)\right)$$

The potential functions U_1 to U_4 in Table 3.1 are the same as the ones used by Rezende and Mohamed (2015), and represent a set of unnormalised densities $p(\mathbf{z}) \propto -\exp(U(\mathbf{z}))$. The densities exhibit different characteristics like multimodality, periodicity, and bifurcations that are difficult to approximate using the typical mean-field approximation.

The flows are trained using mini-batches of 128 samples from the respective densities and the Adam optimiser with learning rates as given in Table 3.2. In this particular experiment, we have access to the true generating functions for each density, so we can train the model on unlimited amounts of data. Hence, we cheaply generate each mini-batch on the fly at each step in the optimisation, abandoning the conventional train/test split used in settings where data is limited. Note that there is

Table 3.2: Details about each flow model.

Potential $U(\mathbf{z})$	**Flow length**	**Learning rate**	**Number of iterations**
$U_1(\mathbf{z})$	32	6.0×10^{-4}	25000
$U_2(\mathbf{z})$	32	6.0×10^{-4}	25000
$U_3(\mathbf{z})$	32	7.5×10^{-4}	1000000
$U_4(\mathbf{z})$	32	6.0×10^{-4}	125000

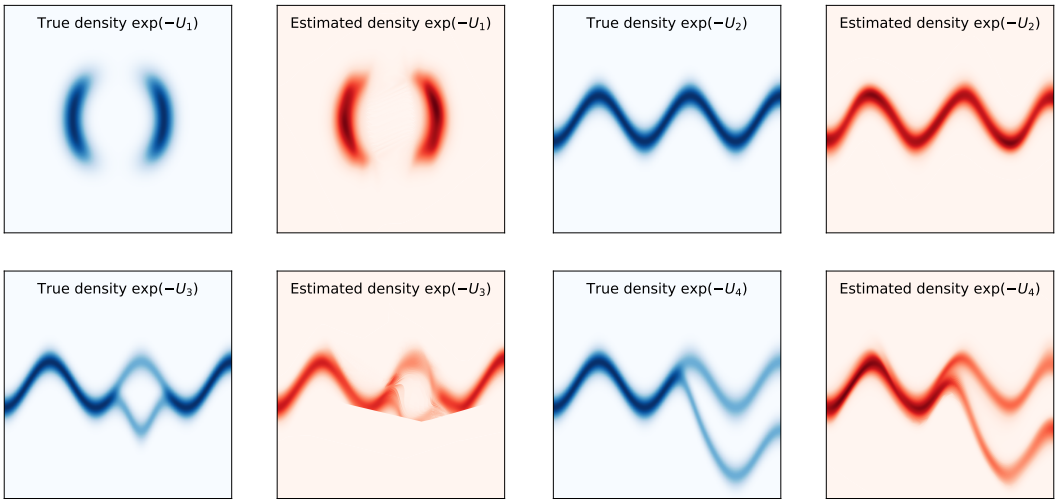


Figure 3.5: Approximations of the densities $p(\mathbf{z}) \propto -\exp(U(\mathbf{z}))$ using planar flows of length $K = 32$.

no risk of overfitting the model, because each data point used during training is unique and exactly represents the true generative process. In the following, we refer to the training on one mini-batch as one *iteration*.

We initially trained each flow model for 25000 iterations, which was sufficient to approximate the densities defined by U_1 and U_2 very well. The model learning the density defined by U_4 required training for 125000 iterations to achieve a qualitatively similar performance, while we trained the model approximating the density defined by U_3 for 1000000 iterations to achieve a sufficiently good approximation. The results are presented in Figure 3.5 alongside the true densities.

We observe that a planar flow of length 32 is able to approximate all of the densities very well. The approximation of the density defined by U_3 exhibits some irregularities, but is qualitatively more similar to the target density than the corresponding result presented in (Rezende and Mohamed, 2015). The observed differences are likely due to the fact that we have used the Adam optimiser rather than the RMSprop optimiser, and trained the model for twice as many iterations as they did in the original paper. The authors of the original paper trained each flow for 500000 iterations on all four densities.

3.8.3 The impact of increased flow length

The preceding results in Figure 3.5 are generated by planar flows of length 32. Looking at the output of normalising flow of this size does not reveal much about what is going on under the hood. We investigate the impact of varying the flow length, and show empirically that increased flow length leads to better approximations of the target density. The target density is defined by potential function U_5 from Table 3.1, which is different from the potential functions presented in (Rezende and Mohamed, 2015), and can be viewed as an even more multi-modal extension of U_1.

We use this new density to illustrate the effect of adding more transformations to a planar flow model. We train planar flows of length $K = 2, 4, 8, 16$ in a similar manner, again using the Adam optimiser with a learning rate of 6×10^{-4}, and training all the flows for the same number of 25000 iterations.

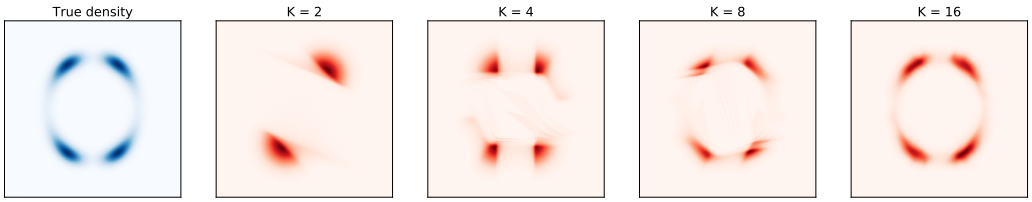


Figure 3.6: Approximations of the density $p(\mathbf{z}) \propto -\exp(U_5(\mathbf{z}))$ using planar flows of length $K = 2, 4, 8, 16$.

The results are presented in Figure 3.6, and we observe that the quality of the approximation is strictly improving with increasing flow length in this experiment.

For all models but the longest one, we can see clear traces of the planar transformations in the shape of symmetries and sharp edges. The non-smooth behaviour in the approximations with depth $K = 4$ and $K = 8$ bear resemblances to the non-smooth behaviour in the estimate of U_3 in Figure 3.5. The sharp edges disappear when we double the length of the flow to 32 transformations. This indicates that increasing the flow length beyond 32 layers in the previous experiment may give us an even better approximation of U_3. Stacking more transformations in sequence can be viewed as analogous to stacking multiple layers in a deep feed-forward neural network, where each layer contributes with a relatively simple non-linear transformation of its input, and the power of the model lies in the complexity of the composition of many simple non-linear transformations.

3.8.4 Dissecting a planar flow

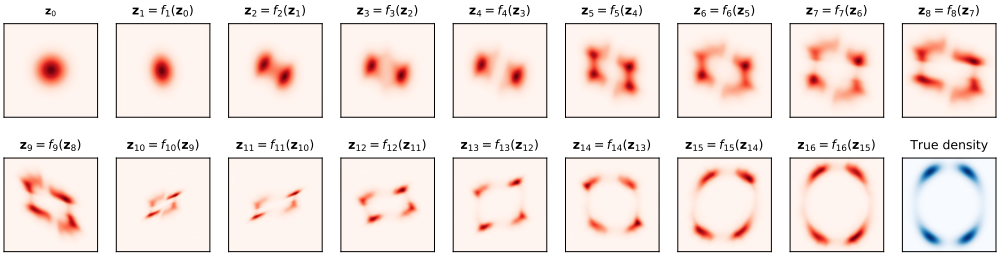
When looking at only the output of a sufficiently long flow, it is not evident that the final transformation is the result of many simple planar transformations. To understand what is happening under the hood, we dissemble the planar flow model and investigate the transformation learned by each layer individually. Remember that each layer in a planar flow is simply a planar transformation as defined in Equation (3.7). Figure 3.7a shows the intermediate outputs after each layer in the planar flow, starting from $\mathbf{z}_0 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, and all the way through to the final transformation $\mathbf{z}_{16} = f_{16} \circ \cdots \circ f_1(\mathbf{z}_0)$.

The views of a normalising flow presented in Figure 3.7 are illuminating because they reveal how the flow model stretches and compresses the initial Gaussian density into something that gradually becomes similar to the target density. Most notably, Figure 3.7a shows that the output of each layer in the model is a relatively simple transformation of its input, and that the power of the planar flow to a large degree lies in the sheer number of transformations that we apply in sequence. The simplicity of each transformation is perhaps even more evident when looking at Figure 3.7b where the transformation of each layer in the flow is applied to a standard bivariate Gaussian.

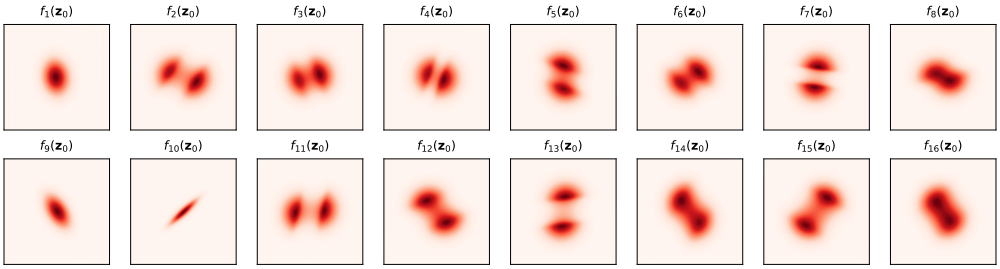
3.8.5 Discussing the properties of planar flows

Based on the experiments above, it is not clear what the theoretical limits of which densities a planar flow can learn are, if any. By extrapolating the observations we have made for the 2D densities above, it is tempting to argue that planar flows, given enough layers, are universal density estimators in two dimensions. Rezende and Mohamed (2015) claim that planar flows in the asymptotic regime are flexible enough to learn any true posterior density. However, this is a claim that lacks a rigorous proof. A different class of normalising flows called *neural autoregressive flows (NAF)* (Huang et al., 2018) has later been proposed alongside a formal proof that NAFs are universal approximators for continuous probability distributions.

At the same time as planar flows have proven to be very flexible density estimators, they are also very limited in practice because each transformation returns a very simple transformation of its input that only affects a small area or volume of the space that it is applied to. Because of this, planar flows are in practice suitable for learning low-dimensional densities only. To transform a high dimensional density into something meaningful and complex, we either need a very long flow with many transformations, or we have to go beyond planar flows and use more powerful transformations.



(a) Visualisations of the output of each layer of a planar flow of length 16.



(b) Visualisations of the learned transformation of each layer of a planar flow of length 16. Each transformation is applied to a standard Gaussian for visualisation.

Figure 3.7: A look at the internal workings of a planar flow.

Another way do gain intuition about why each planar transformation has a limited capacity, is to think of each transformation as a feed-forward neural network with the transformation as a bottleneck layer with one neuron and a skip-connection, as pointed out by Kingma et al. (2016). van den Berg et al. (2018) made improvements on this end by introducing the *Sylvester normalising flow*. The Sylvester normalising flow is very similar to the planar flow, but replaces the vectors $\mathbf{u}, \mathbf{w} \in \mathrm{R}^D$ by matrices $\mathbf{U}, \mathbf{W} \in \mathrm{R}^{D \times M}$, and the scalar bias by an M-dimensional bias vector. The planar flow is a special case of the Sylvester flow with $M = 1$, and the hyperparameter $M \leq D$ defines the number of neurons in the bottleneck layer. The determinant of the Jacobian of this transformation can be efficiently computed using Sylvester's determinant identity.

Chapter 4

A Precursor to Autoregressive Flows

This chapter lays out the foundations for the class of *autoregressive normalising flows*. As mentioned in Section 3.5, a large body of the literature on normalising flows makes use autoregressive transformations with lower-triangular Jacobians to obtain flexible and tractable flows for use in variational inference (Kingma et al., 2016) and density estimation (Papamakarios et al., 2017).

Before we proceed to present autoregressive flows in detail in Chapter 5, we first take a brief detour to explain the essentials of autoregressive models in general. We will also present the *Masked Autoencoder for Distribution Estimation (MADE)* (Germain et al., 2015), a neural autoregressive distribution estimator. MADE is an interesting model worthy of its own chapter, but it is presented here because it is the key building block in the autoregressive flows *Masked Autoregressive Flow (MAF)* (Papamakarios et al., 2017) and *Inverse Autoregressive Flow (IAF)* (Kingma et al., 2016).

Lastly, we write out the details on how to turn MADE into a general density estimator for continuous densities, by using Gaussian conditionals instead of Bernoulli conditionals. The Gaussian MADE is already used in IAF and MAF, but the details on how to make the MADE output Gaussian densities is left out of both papers. To our knowledge, this is the first time these details are explicitly written out in the literature.

4.1 Autoregressive models for density estimation

One general approach in density modelling is to make use of what is called *autoregressive models*, where we factorise the joint distribution over a random vector $\mathbf{x}$ using the probability product rule

$$p(\mathbf{x}) = p(x_1) \prod_{i=2}^{D} p(x_i \mid x_1, \ldots, x_{i-1}) = \prod_{i=1}^{D} p(x_i \mid \mathbf{x}_{<i}), \tag{4.1}$$

and model each scalar conditional distribution individually. In autoregressive models, the i-th element in $\mathbf{x}$ is only dependent on the elements that come before it given the chosen ordering of the random vector. We refer to this as *the autoregressive property*, and the autoregressive structure is illustrated in Figure 4.1. Some random vectors, such as time series data, have a natural (temporal) ordering of their elements, but most random vectors do not have an innate ordering of their variables, in which case we have to *choose* the ordering to impose on their elements.

Having to choose an ordering is a weakness with the autoregressive approach, as there are factorially many orderings to choose from, and it is not possible to a priori know which ordering that will work best in practice. As illustrated in (Papamakarios et al., 2017), the performance of an autoregressive density estimator is not invariant under the ordering of the variables, because each factorisation of $p(\mathbf{x})$ captures different dependencies between them. Autoregressive neural density estimators usually deal with this problem by using different orderings of the variables in different layers of the model, exactly to capture different inter-variable relationships. Another way to deal with this issue is to train the density estimator on different orderings of the input, and combine them into an ensemble.

Using autoregressive models for neural density estimation has been an active line of research since before normalising flows were popularised, starting with the *Neural Autoregressive Distribution*

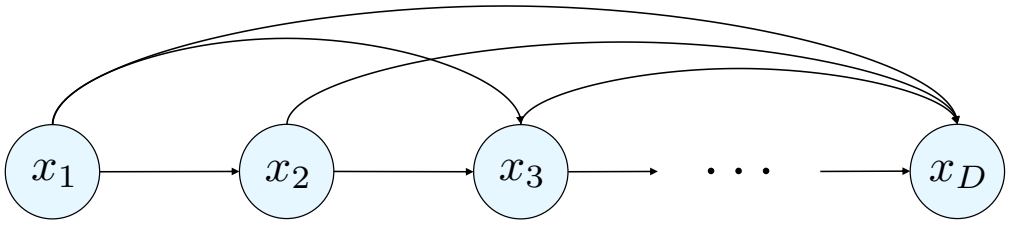


Figure 4.1: Graph showing the dependencies in an autoregressive model.

Estimator (NADE) (Larochelle and Murray, 2011) which extended on previous work by Frey et al. (1996) on *Fully Visible Sigmoid Belief Networks (FVSBN)* for density estimation. Uria et al. (2014) proposed a deeper version of NADE and a training procedure using randomised orderings of the variables and extensive parameter sharing between the models trained for each ordering. Finally, we mention the *Masked Autoregressive Distribution Estimator (MADE)* (Germain et al., 2015) which extended on the deep NADE by offering evaluation of probabilities at test time that is an order of magnitude faster than using deep NADE. MADE is present in Section 4.3 because it plays a central role in several autoregressive normalising flows.

4.2 Autoregressive models as flows

The argument for normalising flows being universal density approximators laid out in Chapter 3.4 builds upon decomposing a density into a product of conditionals using the product rule of probabilities and transform it into a uniform base density. This transformation is an instance of what we call *autoregressive* transformations, and the argument for normalising flows being universal density approximators (presented in Section 3.4) is thus simultaneously an argument about autoregressive flows in particular being universal density approximators, motivating the search for models within this class of flows.

Autoregressive flows are simply compositions of autoregressive transformations, and the connection between autoregressive models and normalising flows was pointed out in (Kingma et al., 2016) and exploited by (Kingma et al., 2016; Papamakarios et al., 2017; Huang et al., 2018) to create a new class of normalising flows called *autoregressive flows*, achieving state of the art results on a variety of benchmarks for variational inference and density estimation.

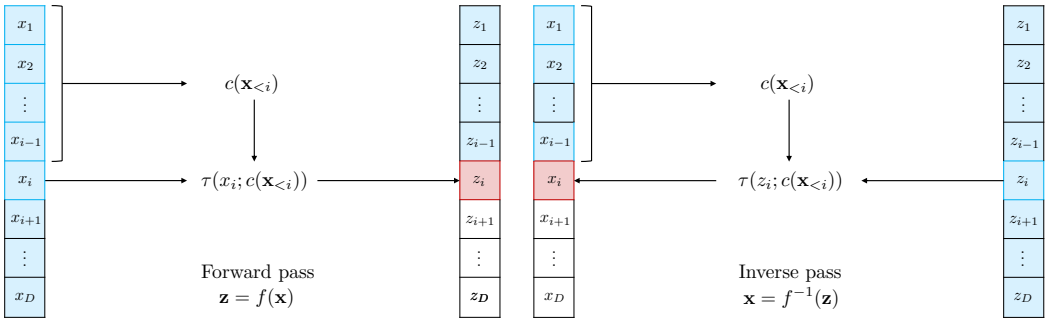
In an autoregressive flow, we model each conditional density in Equation (4.1) as a density whose parameters only depend on the previous elements of the random vector. To transform a random vector $\mathbf{x}$ using an autoregressive transformation, we specify each transformation as:

$$z_i = f(\mathbf{x}_{<i}) = \tau(x_i; c(\mathbf{x}_{<i})) \tag{4.2}$$

where τ is the *transformer*, and c the autoregressive *conditioner* (Huang et al., 2018). The transformer has to be invertible as a function of its input, x_i. The conditioner outputs the parameters of the transformer as a function of the previous variables $\mathbf{x}_{<i}$, but does not have to be invertible. When we have the vector $\mathbf{x}$ at hand, we can readily compute the parameters given by the conditioner $c(\mathbf{x}_{<i})$ for all elements x_i, and compute the forward pass $\mathbf{z} = f(\mathbf{x})$ in parallel. As long as the criterion of the transformer being invertible is satisfied, we can also compute $\mathbf{x}$ given $\mathbf{z}$ as:

$$x_i = \tau^{-1}(z_i; c(\mathbf{x}_{<i})) \tag{4.3}$$

This computation is not parallelisable, because we need to compute all the previous elements in $\mathbf{x}_{<i}$ in order to evaluate the conditioner for x_i. That makes the inverse pass inherently sequential. The forward and the backward pass are illustrated in Figure 4.2.



(a) Parallelisable forward pass.

(b) Sequential inverse pass.

Figure 4.2: The forward pass and the inverse pass of an autoregressive transformation.

The transformation defined in Equation (4.2) has a lower-triangular Jacobian because $\partial z_i / \partial x_j = 0$ for $j > i$. The logarithm of the magnitude of the det-Jacobian is given by:

$$\log \left| \det \frac{\partial \mathbf{z}}{\partial \mathbf{x}} \right| = \sum_{i=1}^{D} \log \left| \det \frac{\partial \tau}{\partial x_i}(x_i; c(\mathbf{x}_{<i})) \right| \tag{4.4}$$

Because of the aforementioned properties; invertibility and a tractable Jacobian, the autoregressive transformation in Equation (4.2) is suitable for making a normalising flow. By chaining several such transformations we obtain a flow, and this lays the groundwork for the class of autoregressive flows. The main drawback of autoregressive flows is that the inverse pass is sequential and hence not parallelisable. The consequence of this is that we can not utilise the benefits that the use of multiple GPUs has offered to deep learning in general when doing the inverse pass in autoregressive flows.

Autoregressive flows can hence not provide both efficient sampling and efficient density evaluation in the same flow. One has to make a trade-off when designing the model. For density estimation, one should model the flow in the direction $\mathbf{u} = f(\mathbf{x})$ (Papamakarios et al., 2017) for efficient density evaluation, while for models where efficient sampling is of importance, one should model the flow in the direction $\mathbf{x} = f(\mathbf{u})$ (Kingma et al., 2016) with $\mathbf{u}$ being a sample from the base density $p_{\mathbf{u}}(\mathbf{u})$.

4.3 Masked autoencoder for distribution estimaton

It follows from the discussion in the previous section, that while the transformer has to be invertible, there are no such constraints on the conditioner. The only requirement is that the i-th conditioner should only be dependent on the $i-1$ first inputs. Hence, the parameters specified by the conditioner can be computed by an arbitrarily complex and not necessarily invertible function. In particular, the conditioner is commonly modelled using a neural network. One could in principle use different networks to output the parameters of the transformer of each scalar x_i, but this becomes very computationally expensive for larger models.

This is why the *masked autoencoder for distribution estimation (MADE)* (Germain et al., 2015) has become the most popular choice of conditioner in autoregressive flows. A MADE network is attractive for this purpose because it is able to output the parameters of all the transformers in one single and parallelisable forward-pass. A MADE network is a modified autoencoder designed such that each output is only dependent on the previous inputs, satisfying the autoregressive property. Before we proceed to how these outputs can be used in autoregressive flows, we first present the original MADE that was used for distribution estimation for binary inputs. In the original MADE, the outputs define a set of conditional Bernoulli distributions $p(x_i \mid \mathbf{x}_{<i})$ for a particular choice of loss function. MADE thus offers one-pass density evaluation of an arbitrary sample.

4.3.1 Background

The idea of an autoencoder is to encode the input $\mathbf{x} \in \mathbb{R}^D$ to some (usually lower-dimensional) latent representation $\mathbf{z} = \text{enc}(\mathbf{x})$, and to reconstruct the input from this latent representation. The

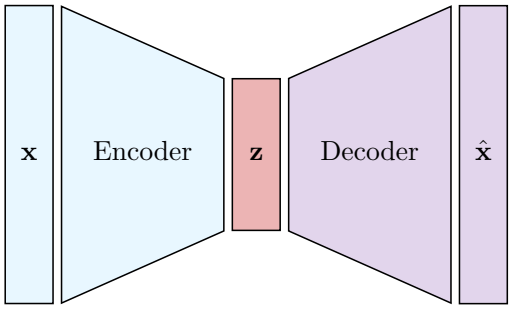


Figure 4.3: Sketch of an autoencoder. Rectangular shapes represent vectors. The input $\mathbf{x}$ and the reconstructed input $\hat{\mathbf{x}}$ have the same dimensionality, while $\mathbf{z}$ is typically of a lower dimensionality.

latter process is referred to as *decoding*, and the reconstructed input is denoted by $\hat{\mathbf{x}} = \text{dec}(\text{enc}(\mathbf{x}))$. See Figure 4.3 for an illustration of a typical autoencoder model. The output and the input of an autoencoder have the same dimensionality, and both the encoder and the decoder are usually modelled as neural networks. In the following we assume both to be feed-forward neural networks.

In the case of binary observations where each x_i takes on either 0 or 1, we can measure the quality of the autoencoder (the quality of the reconstructions) using the cross-entropy loss:

$$\mathcal{L}(\mathbf{x}) = \sum_{i=1}^{D} -x_i \log \hat{x}_i - (1 - x_i) \log(1 - \hat{x}_i) \tag{4.5}$$

which has the same shape as the negative log-likelihood of a Bernoulli distribution. However, if this actually was a proper negative log-likelihood of a Bernoulli, the implied data distribution would be $p(\mathbf{x}) = \prod_{i=1}^{D} \hat{x}_i^{x_i}(1 - \hat{x}_i)^{1-x_i}$. This would not be guaranteed to be a normalised distribution, i.e, the sum over all possible input vectors $\mathbf{x}$ would not sum to one[1]. This problem is addressed by Germain et al. (2015) by defining the joint probability of the binary input variables x_i as a product of Bernoulli distributions using the probability chain rule:

$$p(\mathbf{x}) = \prod_{i=1}^{D} p(x_i \mid \mathbf{x}_{<i}) = \prod_{i=1}^{D} p^{x_i}(1 - p)^{1-x_i} \tag{4.6}$$

with $p = p(x_i = 1 \mid \mathbf{x}_{<i}) = \hat{x}_i$ and $1 - p = p(x_i = 0 \mid \mathbf{x}_{<i}) = 1 - \hat{x}_i$. By defining the joint distribution of the input this way, each output of the autoencoder, $\hat{x}_i$, parameterises one conditional distribution in the product in Equation (4.6). The negative log-likelihood of the joint distribution corresponds exactly to the loss function in Equation (4.5).

This observation allows us to define a modified autoencoder that outputs a valid probability distribution simply by minimising the regular cross-entropy loss in Equation (4.5), and use the autoencoder as a distribution estimator for an arbitrary binary input vector. However, for this to be the case, we need the i-th output of the autoencoder to only be dependent on the $i - 1$ first elements of the input so that the autoregressive property is conserved, and Equation (4.6) is satisfied. We describe how to achieve this in the following.

4.3.2 Modifying the autoencoder

In a fully-connected neural network, all neurons in each layer will be connected to every neuron in the previous layer, and hence all outputs of the network will depend on all dimensions of the input $\mathbf{x}$. Germain et al. (2015) propose an elegant solution to this problem by using binary masks to zero out connections between neurons in the fully-connected networks. These masks have to be carefully designed in order to satisfy the autoregressive property:

[1] Consider the case where the autoencoder learns the identity mapping between the input and output. Then, the cross-entropy loss would be 0 for all inputs $\mathbf{x}$, with an implied distribution that has $p(\mathbf{x}) = 1$ for all $\mathbf{x}$.

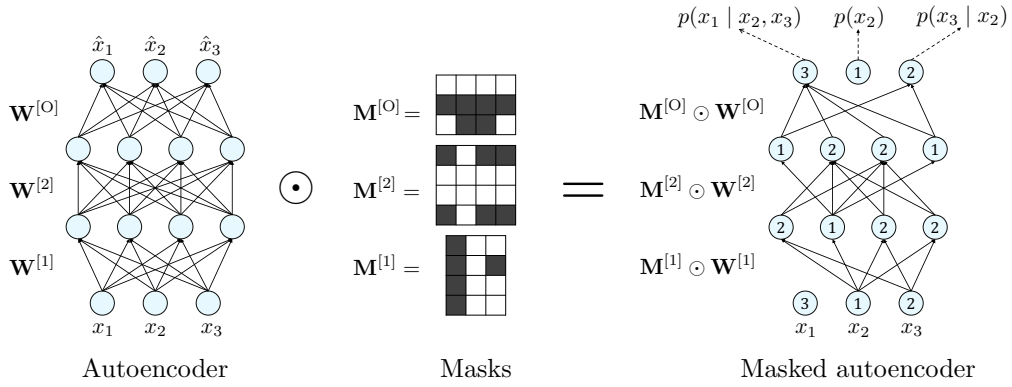


Figure 4.4: The binary masks drop connections between neurons. The numbers inside the neurons in the masked autoencoder on the left correspond to the numbers $m^{[l]}(k)$, defining how many of the input neurons the k-th neuron of the l-th layer can be connected to. We have used a randomised input ordering. This figure is heavily inspired by Figure 1 in (Germain et al., 2015).

We assign each neuron in a hidden layer of the autoencoder an integer m that determines the maximum number of connected inputs the k-th neuron of layer l is allowed to have:

$$m^{[l]}(k) \sim \text{DiscreteUniform}[m_0^{[l]}, D-1]$$

where $m_0^{[l]} = \min_{k'} m^{[l-1]}(k')$. This lower limit on the uniform interval ensures that there are no unconnected units in the masked out network by avoiding to create hidden units in layer l that are not allowed to be connected to any of the hidden units in the previous layer. By setting the upper limit on the uniform interval to $D-1$, we have also excluded the possibility of a hidden unit being connected to all D inputs, as such a hidden unit would be useless in modelling conditionals $p(x_i \mid \mathbf{x}_{<i})$. An illustration of a masked autoencoder is given in Figure 4.4.

We denote the weight matrix of the l-th layer by $\mathbf{W}^{[l]}$. In a standard feed-forward network, the activation of layer l is given by $\mathbf{a}^{[l]} = g^{[l]}(\mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]})$ where we define $\mathbf{a}^{[0]}$ to be the input $\mathbf{x}$. The dropped connections due to masking translates into a binary mask matrix $\mathbf{M}^{[l]}$ where we set the entries of the connections we wish to drop to zero. The activation of the l-th layer is then given by:

$$\mathbf{a}^{[l]} = g^{[l]}(\mathbf{M}^{[l]} \odot \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}) \tag{4.7}$$

We define the entries of the mask matrix using that hidden unit k in layer l can only be connected to the hidden units k' in layer $l-1$ that satisty $m^{[l-1]}(k') \leq m^{[l]}(k)$. The elements of the resulting mask $\mathbf{M}^{[l]}$ for the l-th hidden layer then becomes:

$$M_{k,k'}^{[l]} = \begin{cases} 1, & \text{if } m^{[l]}(k) \geq m^{[l-1]}(k') \\ 0, & \text{otherwise} \end{cases} \qquad \text{for } 1 \leq l \leq L \tag{4.8}$$

By defining $l = 0$ to refer to the input layer, and by defining $m^{[0]}(d) = d$ in order to keep the natural ordering of the inputs, this definition is valid for all layers in the autoencoder. We have to use a slightly different mask between the last hidden layer and the output layer to conserve the autoregressive property of the output, i.e., that the i-th output is only connected to the input units in $\mathbf{x}_{<i}$. This gives:

$$M_{d,k'}^{[O]} = \begin{cases} 1, & \text{if } d \geq m^{[L]}(k') \\ 0, & \text{otherwise} \end{cases} \tag{4.9}$$

The output of the network is:

$$\hat{\mathbf{x}} = \sigma\left((\mathbf{M}^{[O]} \odot \mathbf{W}^{[O]})\mathbf{a}^{[L]} + \mathbf{b}^{[O]}\right) \tag{4.10}$$

where $\sigma(\cdot)$ denotes the sigmoid function, and ensures that each output of the network can be interpreted as a probability by mapping each output to the range $(0, 1)$.

Each output of the MADE corresponds to the parameter of a Bernoulli distribution, so MADE gives single-pass direct estimates of high-dimensional joint distributions by explicitly parameterising each conditional distribution. MADE achieved state-of-the-art performance with respect to average negative log-likelihood on the test sets of a variety of regression benchmark datasets at the time of publication. The details about the experiments and the results are reported in (Germain et al., 2015).

4.3.3 Order-agnostic training

The authors show empirically that training the model on more than one ordering of the inputs can be beneficial, and refer to this as *order-agnostic training*. Order-agnostic training is achieved by sampling and ordering of the inputs before each mini-batch gradient update.In a MADE network, the input ordering is defined by the vector $\mathbf{m}^0 = [m^{[0]}(1), \ldots, m^{[0]}(D)]$ where $m^{[0]}(d)$ is the position of the original d'th dimension. A natural ordering corresponds to the case where $m^{[0]}(d) = d$ and $\mathbf{m}^0 = [1, \ldots, D]$. A random permutation of the input ordering corresponds to randomly permuting $[1, \ldots, D]$ before assigning the permuted vector to $\mathbf{m}^0$. This is also shown in Figure 4.4 where we do not use the natural order of the inputs (x_1, x_2, x_3), but the permutation $\mathbf{m}^0 = [3, 1, 2]$.

Order-agnostic training allows us to create an ensemble of autoregressive models, because models trained on different orderings of the input will correspond to different models. By reordering the vector $[1, \ldots, D]$ for each mini-batch during training, we are able to train as many models as there are orderings of D elements, i.e., factorially many models, using only one set of parameters. Hence, we can create an ensemble at test time by sampling a set of different orderings, compute $p(\mathbf{x})$ under each model, and average the results. This is more computationally expensive than computing $p(\mathbf{x})$ from a single forward-pass using *one* ordering, but using order-agnostic training was found to lower the negative log-likelihood test results on several density estimation benchmarks.

Germain et al. (2015) also proposed what they called *connectivity-agnostic training*, where they essentially resampled *all* masks for each mini-batch during training. This strategy lead to underfitting in many of the experiments, and a more successful strategy was to sample a finite set of masks prior to training, and alternate through them for each mini-batch. To obtain predictions at test time, we make predictions for each set of masks and average the probabilities.

4.3.4 Sampling

After training the model, it will in many cases be of interest to generate new samples from the learned density. As MADE is an autoregressive model, the sampling procedure is sequential. To sample from a MADE model, the steps are as follows:

1. Initialise an empty vector $\mathbf{x} \in \mathbb{R}^D$.
2. Sample x_1 from an arbitrary unconditional Bernoulli distribution to get a realisation for the first element of the vector $\mathbf{x}$.
3. **For** $i = 2, \ldots, D$**:**
   - Feed $\mathbf{x}$ into the network to obtain $\hat{x}_i$. Recall that $\hat{x}_i = p(x_i = 1 \mid \mathbf{x}_{<i})$.
   - Sample $x_i \sim \text{Bernoulli}(\hat{x}_i)$. Update $\mathbf{x}$.

Sampling using MADE is hence not very efficient, as it requires D forward passes to fill the vector $\mathbf{x}$.

4.4 MADE with Gaussian conditionals

The MADE presented above is the same version that is presented in the original paper (Germain et al., 2015), only allowing for input vectors taking on discrete binary values. The version of MADE that is used in autoregressive flows uses Gaussian conditionals, allowing the models to estimate real-valued, continuous densities. The details on how to implement a MADE with Gaussian conditionals are not explicitly stated in the papers by Kingma et al. (2016) and Papamakarios et al. (2017), nor in the Ph.D. thesis by the same author (Papamakarios, 2019). We make an attempt to fill in the details below:

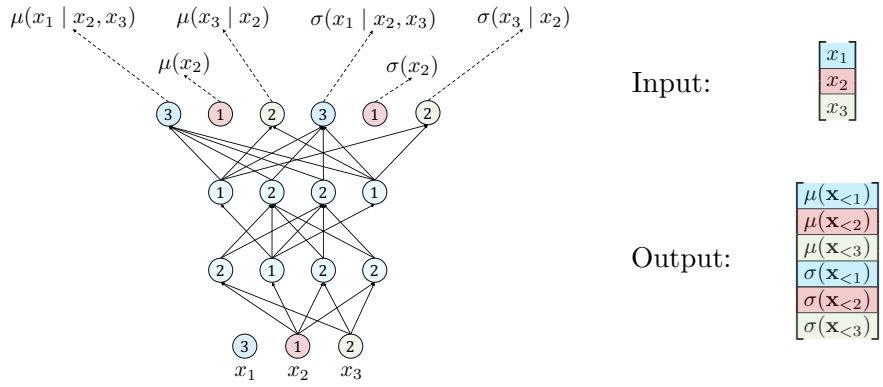


Figure 4.5: Gaussian MADE with colour encoded input and outputs. $\mathbf{x}_{<i}$ refers to the elements of $\mathbf{x}$ assigned a lower ordering than the order of element x_i.

4.4.1 Increasing the size of the output layer

Assume we want to estimate the joint density of a real-valued random vector $\mathbf{x} \in \mathbb{R}^D$ using a MADE network. We want to model each conditional $p(x_i \mid \mathbf{x}_{<i})$ as a one-dimensional Gaussian density, rather than a Bernoulli distribution. A Gaussian density is defined by its mean and variance, so we need the MADE to output two scalar parameters for each dimension of the input vector:

$$\mu(\mathbf{x}_{<i}) :== \mu(x_i \mid \mathbf{x}_{<i})$$
$$\sigma(\mathbf{x}_{<i}) := \sigma(x_i \mid \mathbf{x}_{<i}).$$

The parameters $\mu(\mathbf{x}_{<i})$ and $\sigma(\mathbf{x}_{<i})$ define the i-th conditional, and must depend on the same input variables to ensure that the autoregressive property is retained for each conditional density. This is done by creating the MADE network using the same procedure as before, up until last hidden layer. Because we have twice as many outputs in a Gaussian MADE compared to in a Bernoulli MADE, we must modify the output layer as follows:

In the MADE with Bernoulli conditionals, the weight matrix between the last hidden layer and the output layer has dimensions $(D \times n_L)$, where we let n_L denote the number of units in the last hidden layer. In the MADE with Gaussian conditionals, the output layer is twice as big, so the the weight matrix between the last hidden layer and the output layer has dimensions $(2D \times n_L)$, and we have to increase the size of the mask matrix accordingly.

There are many ways to create the mask matrix for this last hidden layer. We want the outputs to have pairwise identical dependencies on the inputs. One straightforward way to implement this, is to create a mask for the D first outputs in the same way as for the Bernoulli MADE, and stack two such matrices vertically. By doing so, we get a mask matrix of the appropriate dimensionality that gives the output elements $1, ..., D$ pairwise the same dependencies on the inputs as output elements $D + 1, ..., 2D$. As shown in Figure 4.5, the conditional density of the i-th element is then given by the i-th output, defining the mean, and the $(D + i)$-th output, defining the standard deviation.

4.4.2 Gaussian likelihood

We leave all outputs of the network to be unconstrained, real values. For numerical stability and to ensure that the standard deviations returned by the MADE network are non-negative, we define the second half of the output to be the log-standard deviations $\alpha(\mathbf{x}_{<i})$ such that $\sigma(\mathbf{x}_{<i}) = \exp \alpha(\mathbf{x}_{<i})$. The elements of the mean vector are given directly by the D first outputs of the network, as the mean

of a Gaussian is allowed to take on any real value. The log-likelihood of the joint density is then

$$\begin{aligned}
\log p(\mathbf{x}) &= \log \left(\prod_{i=1}^{D} p(x_i \mid \mathbf{x}_{<i}) \right) \\
&= \log \left(\prod_{i=1}^{D} \mathcal{N}\left(x_i; \mu(\mathbf{x}_{<i}), \sigma(\mathbf{x}_{<i})\right) \right) \\
&= \log \left(\prod_{i=1}^{D} \frac{1}{\sqrt{2\pi}\sigma(\mathbf{x}_{<i})} \exp\left(-\frac{1}{2} \left(\frac{x_i - \mu(\mathbf{x}_{<i})}{\sigma(\mathbf{x}_{<i})} \right)^2 \right) \right) \\
&= -\frac{1}{2} D \log(2\pi) + \sum_{i=1}^{D} \log \sigma(\mathbf{x}_{<i}) - \frac{1}{2} \sum_{i=1}^{D} \left(\frac{x_i - \mu(\mathbf{x}_{<i})}{\sigma(\mathbf{x}_{<i})} \right)^2 . \qquad (4.11)
\end{aligned}$$

The computation of the log-likelihood given in Equation (4.11) can be computed efficiently by defining the vector $\mathbf{u} \in \mathbb{R}^D$ with elements:

$$u_i = \frac{x_i - \mu(\mathbf{x}_{<i})}{\sigma(\mathbf{x}_{<i})}. \qquad (4.12)$$

The negative log-likelihood then becomes:

$$\mathcal{L}(\mathbf{x}) = -\log p(\mathbf{x}) = \frac{1}{2} \left(\|\mathbf{u}\|_2^2 + D \log(2\pi) \right) - \sum_{i=1}^{D} \log \sigma(\mathbf{x}_{<i}).$$

which will be the training objective for the Gaussian MADE, replacing the cross-entropy function used for estimating discrete, binary distributions in the original MADE.

Another viable way to implement the Gaussian MADE, is to use two separate MADEs with identical masks to compute the means and the standard deviations. This would abandon the extensive parameter sharing that is used in the current approach, but could potentially lead to better learning, because the means and standard deviations are then computed using two disjoint sets of weights. As of now, they share weights up until the last hidden layer, and the distinction between means and standard deviations is made solely in the output layer. The training objective for the alternative approach is the same as described above, and the two networks can be trained simultaneously with a gradient-based optimiser. The forward pass would still be fully parallelisable. This idea is not tested here, but it would be interesting to compare the performance of the two approaches in future work.

The flexibility of MADE can be increased further by either adding more Gaussian components per conditional such that each conditional is represented as a mixture of Gaussians. This can be implemented in the natural way, extending on the procedure described above. The flexibility of MADE can also be increased by stacking multiple MADEs in sequence to get the *masked autoregressive flow*. Both approaches were proposed (and combined) by Papamakarios et al. (2017), and the masked autoregressive flow is presented in Chapter 5.2.

Chapter 5

Autoregressive Flows

The main body of literature on normalising flows can be roughly divided into three: (*i*) normalising flows for variational inference, and (*ii*) normalising flows for density estimation, including (*iii*) normalising flows used in generative modelling.

We have already presented and experimented the planar flow for variational inference (Rezende and Mohamed, 2015) in Chapter 3. In this chapter, we present a selection of flow-based models covering all of the areas (*i*)-(*iii*) mentioned above, followed by a discussion of their relative strengths and weaknesses. The flows are:

- The *inverse autoregressive flow (IAF)* (Kingma et al., 2016) is the first flow based on interpreting autoregressive models as normalising flows. It allows for more flexible transformations of the base density and more efficient sampling in variational inference than previously proposed methods like planar and radial flows (Rezende and Mohamed, 2015). IAF offers fast sampling, but slow density evaluation.
- The *masked autoregressive flow (MAF)* is a very similar model to IAF, but with complementary strengths and weaknesses. Fast density evaluation at the cost of slow sampling makes MAF suitable for density estimation.
- The *real-valued non-volume preserving (Real NVP) flow* is the first successful flow-based deep generative model. Unlike MAF and IAF, Real NVP makes use of a *coupling layer* to achieve the elementwise invertible transformation. Offers fast density evaluation and fast sampling at the cost of limited flexibility in each transformation.

In addtion, the neural autoregressive flow (Huang et al., 2018) is briefly presented as an example of a non-affine flow:

- The *neural autoregressive flow (NAF)* offers more flexible transformations than the other flows by using an invertible neural network to model the transformer in each autoregressive transformation.

Finally, we present a batch normalisation layer that is suitable for use in normalising flows, as this is perhaps the most critical implementational detail when building a deep flow. We present the batch norm layer from the appendix of (Papamakarios et al., 2017), which in turn builds on the batch norm layer for normalising flows by Dinh et al. (2017).

5.1 Inverse autoregressive flows for variational inference

In the context of variational inference, efficient density evaluation and efficient sampling are both important, as both operations are performed repeatedly during training when evaluating the ELBO. As discussed in the previous chapter, the sampling procedure for an autoregressive model is inherently sequential and slow for high-dimensional variables because each element x_i depends on all elements $x_1, \ldots, x_{i-1}$. An autoregressive model with a sequential forward pass is hence not practical for variational inference, because efficient sampling is crucial for the model to be deemed practical.

Kingma et al. (2016) consider a vector-valued random variable $\mathbf{x} \in \mathbb{R}^D$ modelled by a MADE network with Gaussian conditionals. The network outputs the D-dimensional vectors $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ where the i-th element of each vector is the mean $\mu(\mathbf{x}_{<i})$ and the standard deviation $\sigma(\mathbf{x}_{<i})$ of the i-th conditional density, respectively. The vectors $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ satisfy the autoregressive property with respect to the elements in $\mathbf{x}$. Sampling from such model is a transformation of a random vector $\mathbf{u} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, where the sampled vector $\mathbf{x} = f(\mathbf{u})$ is given by:

$$x_i = \mu(\mathbf{x}_{<i}) + \sigma(\mathbf{x}_{<i}) \cdot u_i.$$

where the first element is sampled randomly. This transformation is sequential and slow, but the crucial observations made by Kingma et al. (2016) are about the inverse transformation $\mathbf{u} = f^{-1}(\mathbf{x})$, which exists long as $\sigma(\mathbf{x}_{<i}) > 0$ for all i. These two observations are:

i The inverse transformation is parallelisable because the elements in $\mathbf{u}$ can be computed independently of each other, and are directly computable as long as the mean and standard deviation vectors are available. The computation can be vectorised as:

$$\mathbf{u} = f^{-1}(\mathbf{x}) = \frac{\mathbf{x} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \tag{5.1}$$

where all operations are understood to be elementwise.

ii The inverse transformation has a lower triangular determinant because $\frac{\partial \mu(\mathbf{x}_{<i})}{\partial x_j} = 0$ and $\frac{\partial \sigma(\mathbf{x}_{<i})}{\partial x_j} = 0$ for $j \geq i$. The det-Jacobian is thus the product of the diagonal elements, which are simply given by $\frac{\partial u_i}{\partial x_i} = \frac{1}{\sigma(\mathbf{x}_{<i})}$. The log-det-Jacobian is given by:

$$\log \det \left| \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \right| = -\sum_{i=1}^{D} \log \sigma(\mathbf{x}_{<i}) \tag{5.2}$$

In other words, the inverse transformation has a tractable det-Jacobian, is parallelisable, and is a fairly flexible affine transformation of a base density. Combined, this makes the transformation a good building block for a normalising flow.

5.1.1 Inverse autoregressive flow

We want to use the parallelisable computation in Equation (5.1) as the forward pass in our flow, i.e., as the *noise to data* transformation, to get a flow that allows for efficient sampling. To achieve this, we reparameterise the expression in Equation (5.1) by swapping the places of $\mathbf{x}$ and $\mathbf{u}$.

$$\mathbf{x} = \frac{\mathbf{u} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} = \frac{1}{\boldsymbol{\sigma}}\mathbf{u} - \frac{\boldsymbol{\mu}}{\boldsymbol{\sigma}} \tag{5.3}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ now are computed by a MADE using $\mathbf{u}$ as the input. To avoid the division, we can rewrite this further as a more numerically stable affine transformation

$$\mathbf{x} = f(\mathbf{u}) = \hat{\boldsymbol{\mu}} + \hat{\boldsymbol{\sigma}}\mathbf{u} \tag{5.4}$$

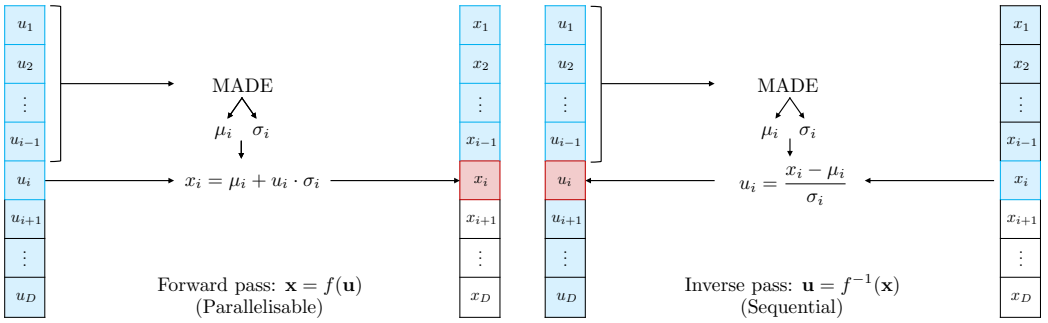
with $\hat{\boldsymbol{\sigma}} = \frac{1}{\boldsymbol{\sigma}}$ and $\hat{\boldsymbol{\mu}} = -\frac{\boldsymbol{\mu}}{\boldsymbol{\sigma}}$. We stack several transformations as the one described in Equation (5.4) in sequence to obtain what is known as an inverse autoregressive flow. We drop the hats from the parameter vectors, and write each transformation in the flow as

$$\mathbf{x}_k = \tau(\mathbf{x}_{k-1}; \boldsymbol{\mu}_k, \boldsymbol{\sigma}_k) = \boldsymbol{\mu}_k + \boldsymbol{\sigma}_k \odot \mathbf{x}_{k-1} \tag{5.5}$$

Here, we have adopted the notation from Chapter 4.2 where $\tau(\cdot)$ denotes the transformer, and $(\boldsymbol{\mu}_k, \boldsymbol{\sigma}_k)$ denote the parameters generated by the conditioner, in this case a Gaussian MADE using the output of the previous step in the flow as its input. The chain of transformations is initialised using a data point that is fed into an encoder network that outputs two initial vectors $\boldsymbol{\mu}_0$ and $\boldsymbol{\sigma}_0$. This encoder network does not have to be autoregressive. The first iterate in the chain is then computed as

$$\mathbf{x}_0 = \boldsymbol{\mu}_0 + \boldsymbol{\sigma}_0 \odot \mathbf{u} \tag{5.6}$$

where $\mathbf{u}$ is an initial sample drawn from a standard Gaussian density.



(a) Parallelisable forward pass.

(b) Sequential inverse pass.

Figure 5.1: The forward pass and the inverse pass of an inverse autoregressive transformation.

Note that the log-determinant of the Jacobian in Equation (5.2) does not change under the reparameterisation, but because $\mathbf{x}$ and $\mathbf{u}$ have swapped places, the log-det-Jacobian of each transformation in the forward pass is now given by

$$\log\det\left|\frac{\partial\mathbf{x}}{\partial\mathbf{u}}\right| = \sum_{i=1}^{D}\log\hat{\sigma}(\mathbf{u}_{<i}) = \sum_{i=1}^{D}\log\frac{1}{\sigma(\mathbf{u}_{<i})} = -\sum_{i=1}^{D}\log\sigma(\mathbf{u}_{<i}) \tag{5.7}$$

The flow is constructed by chaining multiple transformations as the one defined in Equation (5.5), and we emphasise that each transformation is modelled using a different MADE network. The forward and inverse pass of one IAF layer is illustrated in Figure 5.1.

5.1.2 Sampling and density evaluation

Sampling from an inverse autoregressive flow amounts to a simple forward pass through the flow $\mathbf{x} = f(\mathbf{u})$. Combining Equation (5.7) with Equation (5.6) using a Gaussian base density, the density under the final iterate of the transformation is given by:

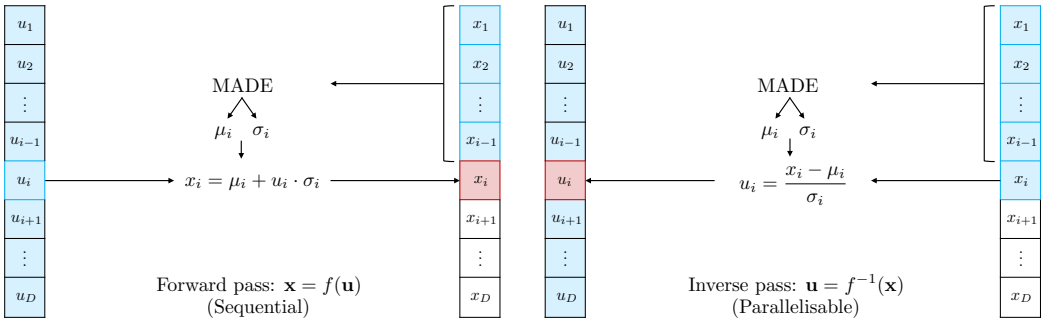
$$\begin{aligned}
\log p\left(\mathbf{x}_K\right) &= \log p_{\mathbf{u}}\left(\mathbf{u}\right) + \sum_{k=1}^{K}\log\det\left|\frac{\partial\mathbf{x}_k}{\partial\mathbf{x}_{k-1}}\right| \\
&= -\sum_{i=1}^{D}\left(\frac{1}{2}u_i^2 + \frac{1}{2}\log(2\pi) + \log(\boldsymbol{\sigma}_0)_i\right) - \sum_{k=1}^{K}\sum_{i=1}^{D}\log(\boldsymbol{\sigma}_k)_i \\
&= -\sum_{i=1}^{D}\left(\frac{1}{2}u_i^2 + \frac{1}{2}\log(2\pi) + \sum_{k=0}^{K}\log(\boldsymbol{\sigma}_k)_i\right)
\end{aligned}$$

In order to evaluate a sample under the final transformed density $p(\mathbf{x}_K)$, we need access to its corresponding random numbers u_i and the standard deviation vectors for all transformations in the flow. For samples generated by the model, these random numbers and vectors are readily available because they are computed during the forward pass and can be kept to compute the density of the generated sample at the end of the forward pass without any additional computational cost.

To evaluate the density of externally provided samples, we first need to compute the random numbers and standard deviations recursively. This requires D backward passes through the flow, and makes the IAF is a bad (slow) choice as a density estimator. Because of one-pass sampling and density evaluation of its own samples, it is instead well-suited for variational inference because we are able to use the samples and their corresponding log-likelihoods to evaluate the evidence lower bound.

5.2 Masked autoregressive flows for density estimation

The masked autoregressive flow (MAF) is closely related to the inverse autoregressive flow, and builds upon the observations made by Kingma et al. (2016), interpreting an autoregressive model as



(a) Sequential forward pass.

(b) Parallelisable inverse pass.

Figure 5.2: The forward pass and the inverse pass of a masked autoregressive transformation.

a normalising flow. A comparison of the two algorithms is given in Section 5.4. MAF is also very closely related to the Gaussian MADE, in fact, a one layer MAF is just a Gaussian MADE.

Papamakarios et al. (2017), like Kingma et al. (2016), consider an autoregressive model with Gaussian conditionals modelled by a Gaussian MADE. In the inverse autoregressive flow, the i-th conditional is parameterised by the mean $\mu(\mathbf{u}_{<i})$ and the standard deviation $\sigma(\mathbf{u}_{<i}) = \exp \alpha(\mathbf{u}_{<i})$, i.e., as a function of the random numbers $\mathbf{u}$. In the masked autoregressive flow, the i-th conditional is parameterised by $\mu(\mathbf{x}_{<i})$ and $\alpha(\mathbf{x}_{<i})$, i.e., as a function of the data $\mathbf{x}$. In a MAF, new data $\mathbf{x} = f(\mathbf{u})$ can be generated sequentially as

$$x_i = \mu(\mathbf{x}_{<i}) + u_i \exp \alpha(\mathbf{x}_{<i}) \tag{5.8}$$

with $u_i \sim \mathcal{N}(0, 1)$. Equation (5.8) is a bijective transformation from noise to data, and the inverse transformation can recover the random numbers $\mathbf{u}$ that was used to generate a data point $\mathbf{x}$ by:

$$u_i = \tau(x_i; \mu(\mathbf{x}_{<i}), \alpha(\mathbf{x}_{<i})) = \frac{x_i - \mu(\mathbf{x}_{<i})}{\exp \alpha(\mathbf{x}_{<i})} \tag{5.9}$$

This transformation is used to evaluate the density of a known sample $\mathbf{x}$, and is parallelisable because the mean and standard deviations are computed simultaneously by forward-passing the sample $\mathbf{x}$ through a Gaussian MADE.

With the elementwise transformation being as described above, we observe that $\partial u_i / \partial x_i = \exp(-\alpha(\mathbf{x}_{<i}))$ and $\partial u_i / \partial x_i = 0$ for $j > i$. The Jacobian of this inverse transformation is thus lower-triangular, and the log-det-Jacobian is given as

$$\log \left| \det \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \right| = \log \left| \prod_{i=1}^{D} \frac{1}{\exp \alpha(\mathbf{x}_{<i})} \right| = -\sum_{i=1}^{D} \alpha(\mathbf{x}_{<i}).$$

Following to the change-of-variables theorem, the likelihood of a data point $\mathbf{x}$ is thus given by $p(\mathbf{x}) = p_{\mathbf{u}}(\mathbf{u}) \exp(-\sum_i \alpha(\mathbf{x}_{<i}))$.

Equation (5.9) makes up one step in a masked autoregressive flow. One such transformation is likely not enough to get a sufficiently good data to noise mapping, starting from an arbitrarily complex data distribution. A more powerful transformation is obtained by stacking multiple such transformations in sequence, using the output of the previous transformation as the input to the next one.

The quality of the data to noise mapping can be used to assess how well the model fits the data. If the model is good, the resulting noise $\boldsymbol{\epsilon}_K$ should follow a D-dimensional Gaussian. If the data is one or two-dimensional, this can be inspected visually. If the dimensionality is higher, this can be tested formally using some normality test. A poor fit may indicate that the model needs more layers, or that further tuning of the hyperparameters is required.

5.2.1 Sampling and density evaluation

Sampling from a MAF with one layer is the same as sampling from a MADE with Gaussian conditionals, so sampling from a MAF with K layers is equivalent to sequentially sampling from a

MADE with Gaussian conditionals K times. This is a very slow process, and MAF is therefore, like MADE, not very well suited for sampling from high-dimensional densities.

Using the change of variables formula, the density of a variable $\mathbf{x} = f(\mathbf{u})$ is given by:

$$\begin{aligned}\log p(\mathbf{x}) &= \log p_{\mathbf{u}}(f^{-1}(\mathbf{x})) + \log \left| \det \frac{\partial f^{-1}}{\partial \mathbf{x}} \right| \\ &= \log p_{\mathbf{u}}(\mathbf{u}) + \log \left| \det \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \right| \\ &= \log p_{\mathbf{u}}(\mathbf{u}) - \sum_{i=1}^{D} \alpha(\mathbf{x}_{<i})\end{aligned}$$

where $p_{\mathbf{u}}(\mathbf{u})$ is some base density for which we are able to evaluate the density. For a masked autoregressive flow with several layers, the formula simply becomes:

$$\log p(\mathbf{x}) = \log p_{\mathbf{u}}(\mathbf{u}) - \sum_{k=1}^{K} \sum_{i=1}^{D} \alpha_k(\mathbf{x}_{<i})$$

To evaluate this density, we only need to evaluate the inverse transformation $\mathbf{u} = f^{-1}(\mathbf{x})$.

5.3 Real-valued non-volume preserving flow

One of the most popular and general models within unsupervised probabilistic modelling is based on *real-valued non-volume preserving (Real NVP)* transformations (Dinh et al., 2017). This model extends on earlier work by introducing a scale term to the previously proposed additive coupling layer used by the *Non-linear Independent Components Estimation (NICE)* (Dinh et al., 2014). The coupling layer used in Real NVP model is an affine coupling layer, which is a simple and tractable bijective transformation $f : \mathbf{u} \to \mathbf{x}$. In each affine coupling layer, the D-dimensional input is split into two parts where the first $d < D$ dimensions stay the same, and the last $D - d$ dimensions are transformed using an affine transformation. The output of each coupling layer is thus defined as

$$\begin{aligned}\mathbf{x}_{1:d} &= \mathbf{u}_{1:d} \\ \mathbf{x}_{d+1:D} &= \mathbf{u}_{d+1:D} \odot \exp \alpha(\mathbf{u}_{1:d}) + \mu(\mathbf{x}_{1:d}),\end{aligned} \tag{5.10}$$

where $\alpha, \mu : \mathbb{R}^d \to \mathbb{R}^{D-d}$ are the scale and translation functions, respectively. The coupling layer used in NICE is similar, but with $\alpha = 0$. The inverse of Equation (5.10) has the same computational complexity as the forward transformation, and can therefore be computed in one pass through the flow as well. Using that the d first dimensions remain the same after the transformation, we find that the inverse of Equation (5.10) is given as

$$\begin{aligned}\mathbf{u}_{1:d} &= \mathbf{x}_{1:d} \\ \mathbf{u}_{d+1:D} &= (\mathbf{x}_{d+1:D} - \mu(\mathbf{x}_{1:d})) \odot \exp(-\alpha(\mathbf{x}_{1:d})).\end{aligned} \tag{5.11}$$

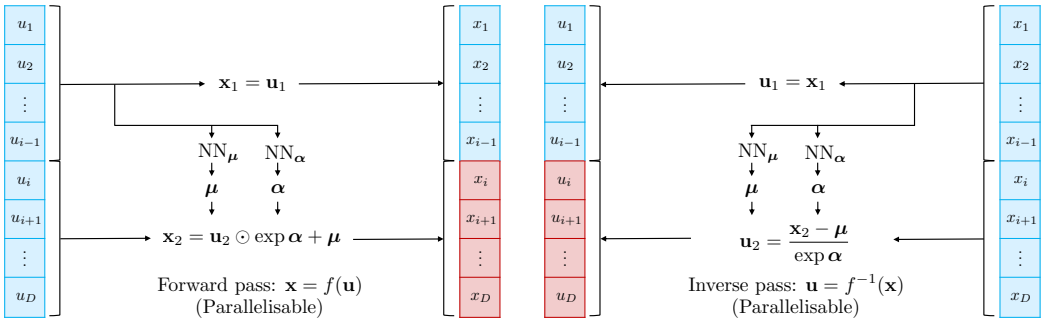
The power of the seemingly simple affine coupling layer lies in the flexibility of the scale and translation functions. Since these functions according to Equation (5.11) do not have to be invertible, they can be arbitrarily complex. In practice, we will choose these functions to be deep neural networks. The Jacobian of this the affine coupling layer is the triangular matrix:

$$\frac{\partial \mathbf{x}}{\partial \mathbf{u}} = \begin{pmatrix} \mathbf{I}_d & 0 \\ \frac{\partial \mathbf{x}_{d+1:D}}{\partial \mathbf{u}_{1:d}} & \operatorname{diag} \exp(\alpha(\mathbf{u}_{1:d})) \end{pmatrix} \tag{5.12}$$

Since the determinant of a triangular matrix is equal to the product of its diagonal elements, the determinant is:

$$\det \frac{\partial \mathbf{x}}{\partial \mathbf{u}} = \exp \left(\sum_i \alpha(\mathbf{u}_{1:d})_i \right) \tag{5.13}$$

where the index i runs over all $D - d$ elements of the vector $\alpha(\mathbf{u}_{1:d})$.



(a) Parallelisable forward pass.

(b) Parallelisable inverse pass.

Figure 5.3: The forward pass and the inverse pass of a Real NVP coupling layer.

Since each coupling layer leaves d components of its input unchanged, it is necessary to apply more than one coupling layer to transform the entire input vector. By composing multiple coupling layers with alternating ordering, we ensure that all components of the input vector are transformed. The generation of new samples is parallelisable since the computation of each element of the output vector of the coupling layer given the input can be computed independently of the other output elements. This is true for both the forward pass and the inverse pass, which makes Real NVP one of the few flows that offers both efficient sampling and efficient inference.

5.3.1 Sampling and density evaluation

Because the forward pass and the inverse pass of the Real NVP have the same computational cost and are parallelisable, it does not matter which way we define the flow. Defining the forward direction of the flow as in Equation (5.10) so that the coupling layer is as shown in Figure 5.3, sampling from the model amounts a single forward pass of a random vector $\mathbf{u}$ through the flow.

To evaluate the density of a sample or observation $\mathbf{x}$, we need to compute the corresponding random number $\mathbf{u} = f^{-1}(\mathbf{x})$ following Equation (5.11), and use the change of variables formula to compute.

$$\begin{aligned}\log p(\mathbf{x}) &= \log p_{\mathbf{u}}(f^{-1}(\mathbf{x})) + \log \left| \det \frac{\partial \mathbf{x}}{\partial \mathbf{u}} \right|^{-1} \\ &= \log p_{\mathbf{u}}(\mathbf{u}) - \sum_{k=1}^{K} \sum_{i} (\boldsymbol{\alpha}_k)_i\end{aligned}$$

This log-likelihood is computed during a single inverse pass. At each transformation of the flow, the vector $\boldsymbol{\alpha}_k$ is computed from the d first elements of the output of the previous layer.

5.4 Comparing autoregressive flows

The masked autoregressive flow makes use of the same transformation as the inverse autoregressive flow, but inverted. This can be seen by writing out Equation (5.5) and Equation (5.8) as:

$$\begin{aligned}\text{IAF}: \quad & x_i = \mu(\mathbf{u}_{<i}) + u_i \exp \alpha(\mathbf{u}_{<i}) \\ \text{MAF}: \quad & x_i = \mu(\mathbf{x}_{<i}) + u_i \exp \alpha(\mathbf{x}_{<i})\end{aligned}$$

The difference between them is how the scale and shift parameters are computed; IAF computes them from the previous random numbers $\mathbf{u}$, while MAF computes them from previous data variables $\mathbf{x}$. Because of this, IAF and MAF have complementary strengths and weaknesses. Which flow to choose has to be decided with the task at hand in mind. For variational inference, IAF offers fast sampling and density evaluation of its own samples, while MAF is not suitable because MAF sampling is sequential and slow. For density estimation, MAF is the better choice because it offers one-pass density evaluation of any sample, while IAF density evaluation is sequential for external samples (i.e., samples "in the wild", not generated by the model itself).

If we need a model that offers both efficient sampling and efficient density evaluation, we should consider using Real NVP, which offers one-pass sampling and one-pass density evaluation (of any sample). This computational convenience comes at the cost of using a less flexible transformation at each step in the flow. The Real NVP paper also proposes the use of masked convolutions and a multi-scale architecture specialised for dealing with image data, so for efficient sampling and exact log-likelihood computation of high-dimensional image data, this implementation of Real NVP is the better choice of the presented flows. *Glow* (Kingma and Dhariwal, 2018) is a more recent flow that has extended Real NVP further with additional invertible convolutional layers, improving both density estimation and generation of images compared to Real NVP. Glow is another example of a flow utilising affine transformations.

Papamakarios et al. (2017) point out that Real NVP can be viewed as a MAF with only one autoregressive step instead of D autoregressive steps. Real NVP is a special case of the autoregressive transformation used in IAF and MAF, which instead of using an individual affine transform that is a function of all previous elements for all x_i, uses a blockwise transformation. The first block of d elements is simply transformed using an identity mapping. The second block with elements $d+1, \ldots, D$ is transformed using a function that is a function of the d first elements. That corresponds to an affine transformation where the scale and shift parameters are defined as follows:

$$
\begin{aligned}
\text{Real NVP}: \qquad x_i &= \mu(\mathbf{u}_{<i})_i + u_i \exp \alpha(\mathbf{u}_{<i})_i \\
\mu(\mathbf{u}_{<i}) &= \begin{cases} 0, & \text{for } i = 1, \ldots, d \\ \mu(\mathbf{u}_{1:d}), & \text{for } i = d+1, \ldots, D \end{cases} \\
\alpha(\mathbf{u}_{<i}) &= \begin{cases} 0, & \text{for } i = 1, \ldots, d \\ \alpha(\mathbf{u}_{1:d}), & \text{for } i = d+1, \ldots, D \end{cases}
\end{aligned}
$$

We observe that each element x_i is only conditioned on any elements with index $j < i$, so the autoregressive property still holds.

The affine transformations utilised in these autoregressive flows presented thus far are all fairly simple, and the expressivity of these flows come mainly from the expressivity of the conditioners, i.e., from the the MADE networks used in IAF and MAF, and in the neural networks used to model α and μ in Real NVP. Affine transformations are popular because they are differentiable and trivially invertible. Recently, models have been proposed that go beyond affine transformations, allowing for more expressive transformations in each step of the flow. Examples of such models are *neural autoregressive flows (NAF)* (Huang et al., 2018), and *Flow++* (Ho et al., 2019). Huang et al. (2018) formally prove that NAF is an universal density estimator, meaning it in the limit can approximate any well-behaved density arbitrarily well. We outline the idea of this flow in the next section.

5.5 A brief intro to neural autoregressive flows

Whereas MAF, Real NVP, and IAF all use affine (scale-and-shift) transformers, Huang et al. (2018) propose an autoregressive flow that uses an invertible neural network as the transformer function. The model is aptly named a *neural autoregressive flow (NAF)*, and defines an example of a non-affine autoregressive flow. Neural networks can model a richer family of output densities than an affine transformation can. In particular, a NAF can transform a unimodal input density into a multimodal density using only one transformation, whereas an IAF needs more than one transformation to do so. In a NAF layer, the transformer is a neural network, such that

$$
\tau(x_i; c(\mathbf{x}_{<i})) = \mathrm{NN}(x_i; c(\mathbf{x}_{<i})),
$$

where the parameters of the neural network $c(\mathbf{x}_{<i})$ are outputs from an autoregressive conditioner.

For a neural network $\mathrm{NN}(\cdot) : \mathbb{R} \to \mathbb{R}$ to be strictly monotonic (and thus invertible, as is required for the transformer), it is sufficient that the network uses strictly positive weights and strictly monotonic activation functions. Examples of strictly monotonic activation functions are the sigmoid, the leaky ReLU, and $\tanh$.

Based on this, Huang et al. (2018) propose a transformer architecture called **deep sigmoidal flows (DSF)**. This is a neural network with scalar input and output, with only one hidden layer of sigmoid units and strictly positive weights. The output layer is a logits layer. Logit is an inverse sigmoid

function, and is defined as:

$$\text{logit}(p) = \log \frac{p}{1-p},$$

mapping its input from the domain $(0, 1)$ to the unconstrained domain $(-\infty, \infty)$. To ensure that the input to the logits layers is in $(0, 1)$, the outputs of the sigmoid layer are combined as a softmax-weighted sum:

$$x'_i = \text{logit}(\mathbf{s}^T \cdot \sigma(\mathbf{w} \cdot x_i + b)),$$

where $\mathbf{s}, \mathbf{w} \in \mathbb{R}^{n \times 1}$, where n denotes the number of hidden units. The elements of $\mathbf{s}$ are such that $0 < s_{i,j} < 1$ and $\sum_i s_{i,j} = 1$. Remember that we need all weights of the network to be positive to have an invertible network, so we also require $a_{s,t} > 0$.

The argument for using sigmoid units, is that the sigmoid function (in addition to being strictly monotonic) has an inflection point, which in turn induces inflection points in the transformer. It is these inflection points that allow the transformer to readily output multimodal densities, because they enable the transformer to expand and contract different regions of the output density in only one transformation.

If we stack multiple DSF transformations in sequence, we get a feed-forward neural network with bottleneck layers. Another way to improve the flexibility of the neural autoregressive flow is to generalise the DSF transformation to several fully-connected sigmoid layers between the input and the output while preserving the positivity of the weights. This gives the **deep dense sigmoidal flow**.

The authors use small transformer networks with one to two hidden layers with only 8 or 16 hidden units in each layer. These choices were found to yield good results without increasing the computational cost too much compared to IAF and MAF. Remarkably, Huang et al. (2018) proved formally that a deep sigmoidal flow can approximate any strictly monotonic univariate function, so the NAF using DSF is a universal density approximator. See the original paper for details.

5.6 Batch normalisation

When training deep models in general, it is often advantageous, and in many cases necessary, to normalise the activations of the hidden layers at each forward pass to improve the training of the model. This is known as *batch normalisation*, or simply *batch norm* (Ioffe and Szegedy, 2015), and is one of the most common and most crucial implementational tricks when working with deep neural networks. Precisely *why* batch norm works so well is still debated (Bjorck et al., 2018), but applying batch norm has widely demonstrated to speed up and stabilise the training of neural networks.

The technique works by normalising all dimensions of the activations of each hidden layer during the forward pass by using statistics computed per mini-batch. Intuitively, because the weights of each hidden layer changes during training, so does the input distribution of each layer. By normalising the input to each layer to have zero mean and unit variance, we ensure that no activations get really big or really small, avoiding saturated activation functions that could hamper learning.

Batch norm for normalising flows was first proposed and used in Real NVP (Dinh et al., 2017). We choose to implement and use a slightly modified version of batch normalisation that is presented in Appendix B of (Papamakarios et al., 2017). Batch normalisation is a non-volume preserving transformation, so to include it in a flow, we need to be able to compute its Jacobian determinant efficiently. Luckily, batch norm can be viewed as a composition of affine transformation, and is straightforward to differentiate.

We consider a general autoregressive flow $\mathbf{x} = f(\mathbf{u}) = f_K \circ \cdots \circ f_1(\mathbf{u})$, and add batch norm between all autoregressive layers and between the last autoregressive layer and the base density, such that the flow becomes

$$\mathbf{x} = f(\mathbf{u}) = f_K \circ \text{BN}_K \circ f_{K-1} \circ \cdots \circ f_1 \circ \text{BN}_1(\mathbf{u}).$$

To simplify notation, we consider one batch norm layer and let $\mathbf{x}$ represent the vector closer to the data, and we let $\mathbf{u}$ represent the side of the transformation that is closer to the base density. Then,

$$\mathbf{x} = \text{BN}(\mathbf{u}) = (\mathbf{u} - \boldsymbol{\beta}) \odot (\mathbf{v} + \boldsymbol{\epsilon})^{1/2} \odot \exp -\boldsymbol{\gamma} + \mathbf{m},$$

The inverse transformation is given by

$$\mathbf{u} = \mathrm{BN}^{-1}(\mathbf{x}) = (\mathbf{x} - \mathbf{m}) \odot (\mathbf{v} + \boldsymbol{\epsilon})^{-1/2} \odot \exp \boldsymbol{\gamma} + \boldsymbol{\beta}$$

with determinant

$$\det \frac{\partial \mathrm{BN}^{-1}(\mathbf{x})}{\partial \mathbf{x}} = \exp\left(\sum_{i=1}^{D}\left(\gamma_i - \frac{1}{2}\log(v_i + \epsilon_i)\right)\right)$$

The vectors $\mathbf{m}$ and $\mathbf{v}$ represent the sample mean and variance, respectively. During training these are computed per mini-batch. The vectors $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ are trainable parameters, and a small hyperparameter $\boldsymbol{\epsilon}$ is added to the variance ensure numerical stability in case any element of $\mathbf{v}$ is close to zero.

At validation and test time, we set $\mathbf{m}$ and $\mathbf{v}$ equal to sample mean and variance computed using the entire training set. We also experiment with setting the mean and the variance equal to running averages $\tilde{\mathbf{m}}$ and $\tilde{\mathbf{v}}$ computed during training as:

$$\tilde{\mathbf{m}}_{i+1} = \beta\tilde{\mathbf{m}}_i + (1-\beta)\hat{\mathbf{m}}_i$$
$$\tilde{\mathbf{v}}_{i+1} = \beta\tilde{\mathbf{v}}_i + (1-\beta)\hat{\mathbf{v}}_i,$$

where the subscript refers to the mini-batch index, and $\beta \in (0, 1)$ is a hyperparameter, typically close to 1. This yielded very similar results to computing the parameters using the training set. It will become clear in the next chapter which version of batch norm we have used for which experiments.

Chapter 6

A Novel Hybrid Flow for Density Estimation

In this chapter, we combine two of the previously presented layers – the Gaussian MADE, as used in the masked autoregressive flow, and coupling layer used in Real NVP – into a new layer suitable for use in a normalising flow. The proposed layer, which we name the *hybrid autoregressive flow (HAF)*, makes the same trade-offs as a MAF layer, and is thus best suited for use in density estimators. The model can be viewed as a Real NVP that is specialised for density estimation. We present the idea behind HAF in Section 6.1, and formalise it Section 6.2. This chapter is purely descriptive, but in Chapter 7.5 carry out some preliminary experiments, and compare the results with the performance of regular MAF and regular Real NVP.

6.1 Background

The starting point for the proposed flow is the observation that the expressiveness of the Real NVP coupling layer is compromised because the coupling layer leaves a large fraction, typically 50%, of the input elements unchanged in each transformation. As a result, we need two sequential coupling layers using opposite orderings to transform all elements of the input, when we ideally would like to transform every variable in every transformation in the flow, to make the transformation as expressive as possible. Increased expressivity in each layer can reduce the length of the flow needed to achieve the same performance, and it can allow for modelling inter-variable dependencies more efficiently.

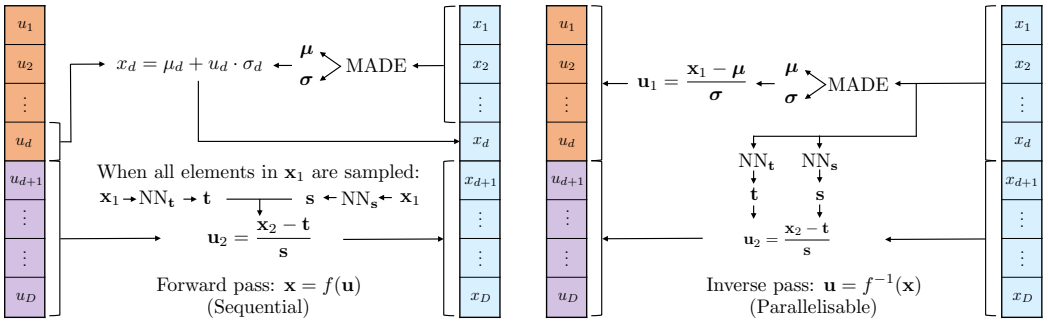
Recall that the (inverse) transformation in each coupling layer is given as:

$$\begin{aligned} \mathbf{u}_{1:d} &= \mathbf{x}_{1:d} \\ \mathbf{u}_{d+1:D} &= (\mathbf{x}_{d+1:D} - \mu(\mathbf{x}_{1:d})) \odot \exp\left(-\alpha(\mathbf{x}_{1:d})\right). \end{aligned} \tag{6.1}$$

The obvious way to increase the flexibility of this transformation, is to also transform the d elements that are currently not being transformed in the coupling layer. The transformations we can use for this purpose must of course be bijective and differentiable, and we also want it to have a lower-triangular Jacobian. A transformation that satisfy all the aforementioned properties can be defined by using a Gaussian MADE in the same way that it is used in the masked autoregressive flow, but instead of transforming the entire input to the layer, we only let the MADE transform the elements that are not already transformed by the coupling layer in Equation (6.1). The transformation defined by the MADE, used for the autoregressive layers in the masked autoregressive flow, is defined elementwise by the transformation

$$u_i = (x_i - \mu(\mathbf{x}_{<i})) \cdot \exp(-\alpha(\mathbf{x}_{<i})). \tag{6.2}$$

The means and log-standard deviations are defined by a Gaussian MADE network with $\mathbf{x}_{1:d}$ as its input, and the elements $u_1, ..., u_d$ can be computed in parallel. This is exactly what we will do in our proposed *hybrid autoregressive flow*, which is presented in the next section.



(a) Partly sequential forward pass.

(b) Parallelisable inverse pass.

Figure 6.1: The forward pass and the inverse pass of a hybrid autoregressive transformation.

6.2 Hybrid autoregressive flow

6.2.1 Extending the coupling layer

We combine the transformations from Equation (6.1) and Equation (6.2) into a modified coupling layer that defines the following forward transformation:

$$\begin{aligned} \mathbf{x}_{1:d} &= \boldsymbol{\mu}_{\text{MADE}} + \mathbf{u}_{1:d} \odot \exp \boldsymbol{\alpha}_{\text{MADE}} \\ \mathbf{x}_{d+1:D} &= t(\mathbf{x}_{1:d}) + \mathbf{u}_{d+1:D} \odot \exp(s(\mathbf{x}_{1:d})), \end{aligned} \tag{6.3}$$

where $\boldsymbol{\mu}_{\text{MADE}} := \mu(\mathbf{x}_{1:d})$ and $\boldsymbol{\alpha}_{\text{MADE}} := \alpha(\mathbf{x}_{1:d})$ are the mean vector and the log-standard deviation vector from a Gaussian MADE and have to be computed sequentially, and $t(\cdot)$ and $s(\cdot)$ as denote the shifting and scaling networks of the coupling layer. These networks have previously been denoted by μ and α, but we change the notation here to avoid ambiguity with the MADE vectors.

Note that the forward pass in Equation (6.3) is partly sequential, as the first d elements have to be sampled sequentially using a MADE. Only when all elements $x_1, ..., x_d$ have been sampled, can the scaling and shifting vectors be computed for the rest of the vector. We are rather interested in the inverse transformation from $\mathbf{x}$ to $\mathbf{u}$. This transformation is fully parallelisable, and defined by

$$\begin{aligned} \mathbf{u}_{1:d} &= (\mathbf{x}_{1:d} - \boldsymbol{\mu}_{\text{MADE}}) \odot \exp(-\boldsymbol{\alpha}_{\text{MADE}}) \\ \mathbf{u}_{d+1:D} &= (\mathbf{x}_{d+1:D} - t(\mathbf{x}_{1:d})) \odot \exp\left(-s(\mathbf{x}_{1:d})\right). \end{aligned} \tag{6.4}$$

When going in this direction, we have the vector $\mathbf{x}$ at hand, and can readily compute $\boldsymbol{\mu}_{\text{MADE}}$ and $\boldsymbol{\alpha}_{\text{MADE}}$ using a single forward pass through the Gaussian MADE, and the same applies to the scaling and shifting of $\mathbf{x}_{d+1:D}$. The final thing we need in order to use this transformation in a flow for density estimation, is a tractable Jacobian with an easy-to-compute determinant to use the change of variables formula:

$$\log p_{\mathbf{x}}(\mathbf{x}) = \log p_{\mathbf{u}}(\mathbf{u}) - \log \det \left| \frac{\partial \mathbf{x}}{\partial \mathbf{u}} \right|$$

The transformation defined in Equation (6.3) has a lower-triangular Jacobian with log-determinant given by

$$\log \det \frac{\partial \mathbf{x}}{\partial \mathbf{u}} = \sum_{i=1}^{d} (\boldsymbol{\alpha}_{\text{MADE}})_i + \sum_j s(\mathbf{x}_{1:d})_j \tag{6.5}$$

where j runs over the $D - d$ elements transformed by the Real NVP style affine transformation. A visualisation of the Jacobian of HAF compared to the Jacobian of the Real NVP is shown in Figure 6.2. This figure is aimed at helping to understand what the differences between the proposed flow and the Real NVP are. One HAF layer is theoretically equivalent to chaining a coupling layer and a MAF layer that is only applied to half of the inputs. However, the HAF layer is faster because the computation of the output when going the inverse direction can be done in parallel, motivating collecting both transformations in one layer.

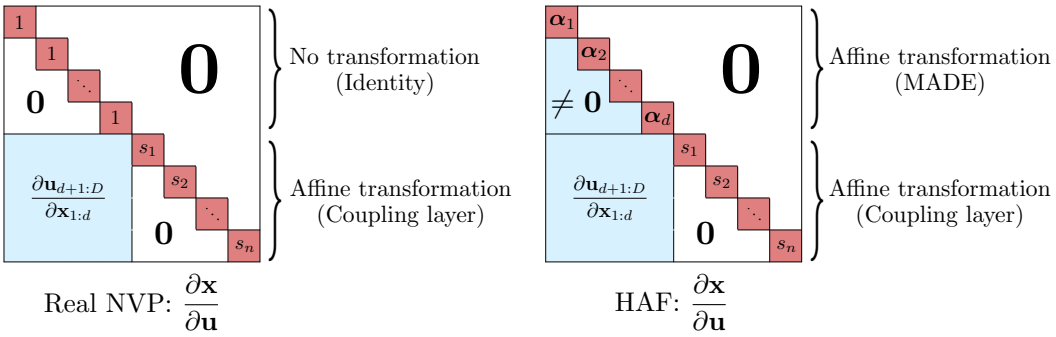


Figure 6.2: **Left:** The Jacobian of the original coupling layer in the Real NVP. **Right:** The Jacobian of the modified coupling layer in the HAF.

6.2.2 Discussion and properties

One of the advantages of the coupling layer in Real NVP is that the forward pass and the inverse pass come with the same computational cost. That convenience is thrown overboard here when introducing a sequential transformation of half of the elements in the forward pass. However, when using the Real NVP for general purpose density estimation, the cost of the forward pass is not a primary concern, as generating samples is not the application that the model is intended for. Trading off efficient sampling for a more flexible transformation is acceptable, as long as the density evaluation remains efficient. The same kind of trade-off is made in the inverse autoregressive flow and the masked autoregressive flow as well. We argue that it is reasonable to sacrifice cheap sampling for better density evaluation.

The hope is that by using a hybrid between masked autoregressive networks (MAF/MADE) and fully connected networks (Real NVP) to model the transformations, we will be able to capture more complex dependencies between the variables than when using either of the models separately. The MAF part of the transformation captures autoregressive dependencies if there are any, and in the Real NVP part of the transformation can an make use of arbitrarily complex neural network to transform the remaining elements using an affine transformation.

While the forward pass in HAF is less efficient than in a pure Real NVP, it is also more efficient than in a pure MAF model, as half of the elements can be computed in parallel, contrasting the fully sequential sampling procedure in MAF. Also, the inverse pass is fully parallelisable and offers a strictly more flexible transformation than the Real NVP does. Whether the proposed transformation is more flexible than the one used in MAF is not immediately clear. To better understand the properties of MAF vs. HAF, we investigate the number of parameters contributing to the each transformation in the competing models. We also include Real NVP in the comparison for completeness.

Number of parameters

We compare number of connection weights in one layer in each of the three models. We ignore the number of bias terms as it is negligible compared to the number of weights. For simplicity, we assume all neural networks used in either model to have the same size of one hidden layer with H hidden units. The input vector to each layer has D elements, and in the coupling layers we assume that the input is split in to two equally sized parts.

MAF: A MAF consists of multiple MADE networks in sequence. Each MADE has a D-dimensional input and $2D$-dimensional output. The total number of weights is thus $DH + 2DH = 3DH$. However, in a MADE, the sub-network modelling each conditional can be much smaller than the the full unmasked network. Assuming that 50% of the weights are masked out due to the uniform assignment of connections in MADE, there are $\frac{3}{2}DH$ "active" parameters in each MAF layer.

Real NVP: The coupling layer consists of a scaling and and shifting network, both of size $1 \times H$ with $\frac{D}{2}$ inputs and $\frac{D}{2}$ outputs. This gives a total of $2 \cdot (\frac{1}{2}DH + \frac{1}{2}DH) = 2DH$ weights per coupling layer. Because there is no masking in these networks, the number of active parameters is the same as the number of total parameters

Table 6.1: Number of weights in one layer of the different flows, given by the size of the input (D), and the size of the hidden layer (H).

Model	#parameters	#active parameters
MAF	$3DH$	$\frac{3}{2}DH$
Real NVP	$2DH$	$2DH$
HAF	$\frac{7}{2}DH$	$\frac{11}{4}DH$

HAF: The first $\frac{D}{2}$ elements are transformed using a MADE with $\frac{D}{2}$ inputs and D outputs, giving a total of $\frac{3}{2}DH$ weights. Considering that half of the weights do not contribute because they are masked out, the number of active weights becomes $\frac{3}{4}DH$. The second half is transformed in the same way as in the Real NVP described above. This adds $2DH$ weights to the transformation, making up a total of $\frac{3}{2}DH + 2DH = 3.5DH$ parameters, out of which $\frac{11}{4}DH$ are active

A summary of the above sections is presented in Table 6.1. We see that HAF has more parameters per layer than MAF and Real NVP. The fraction of active parameters in HAF lies by design between the 50% of MAF and the 100% of Real NVP, and is close to 80%. Papamakarios et al. (2017) argue that MAF makes better use of its available parameters than Real NVP. To make a fair comparison between the flows, the size of the hidden layer in HAF should be reduced accordingly to have the same number of active parameters as MAF and/or Real NVP.

Assuming a fixed input dimension D, we would have to reduce the size of the hidden layer in HAF by a factor $\frac{3}{2}\frac{11}{4} \approx 0.5$ and $2/\frac{11}{4} \approx 0.7$ to get the same number of active parameters as MAF and Real NVP, respectively. The computations above can be easily extended to deeper networks with more than one hidden layer, but we leave this for future work.

6.2.3 Implementation

Because our transformation is partly autoregressive and partly coupling based, we have to be extra careful with the ordering of the intermediate inputs in the flow to make sure that all variables benefit maximally from the autoregressive MAF transformation. As mentioned in Chapter 5, it is common to use alternating ordering in every second layer when using MAF, and to alternate between transforming $\mathbf{x}_{1:d}$ and $\mathbf{x}_{d+1:D}$ in every second layer when using Real NVP. If we naively use both approaches in each layer, we never get to model $\mathbf{x}_{1:d}$ and $\mathbf{x}_{d+1:D}$ in both directions. We need to combine the two strategies to ensure that the MAF part transforms all elements in both the natural and reverse order.

To achieve this, we divide the input $\mathbf{x}$ into two disjoint parts so that the even-indexed elements are in $\mathbf{x}_{1:d}$ and the odd-indexed elements are in $\mathbf{x}_{d+1:D}$. In the first layer of a hybrid autoregressive flow, the elements in $\mathbf{x}_{1:d}$ are transformed according to an autoregressive MAF style transform, while the elements in $\mathbf{x}_{d+1:D}$ are transformed according to a Real NVP style update, dependent on the untransformed variables in $\mathbf{x}_{1:d}$. In the next layer it is the other way around, such that the variables that were originally in $\mathbf{x}_{d+1:D}$ are transformed according to an autoregressive MAF transform, while the elements in $\mathbf{x}_{1:d}$ are transformed according to a Real NVP update, dependent on the variables in $\mathbf{x}_{d+1:D}$. This alternating pattern is repeated throughout the flow.

The elements that are transformed using the MAF transform are transformed using the natural input ordering in the two first layers, and and then the reverse input ordering for the two next layers, and so on. This is done to capture different dependencies between the dimensions of the input, and is the same strategy applied by Papamakarios et al. (2017) and Kingma et al. (2016) in their autoregressive flows. Note that our approach differs slightly from theirs, as we each time only change the ordering for half of the elements in the input vector. Permutation of the rows in a vector is a bijective transformation itself, and is hence allowed as a component of a normalising flow. In particular, it is a volume-preserving transformation, such that the absolute value of the Jacobian of the transformation is equal to one, so it does not show up in our log-det-Jacobian calculations.

Chapter 7

Experiments

In this chapter, we apply the presented algorithms from Chapter 4.2 and Chapter 5 on different density estimation tasks. First, we use the MADE and the Gaussian MADE on a selection of experiments using the MNIST dataset. These experiments are inspired by, but also extending on the experiments conducted by Germain et al. (2015). Then, we apply Real NVP on two two-dimensional toy datasets, and visualise and discuss its performance. The implementations of MADE and Real NVP are found to work satisfactory, which is crucial as these are the builiding blocks of MAF and HAF.

The preceding experiments lead up to the final set of experiments in Section 7.5, where we apply MADE, MAF, Real NVP, and our proposed flow, HAF, on a two different general purpose density estimation datasets. We follow the experimental setup from (Papamakarios et al., 2017), successfully reproduce their reported results for MADE, MAF, and Real NVP, and show that HAF gives promising initial results across the different datasets. The section is rounded off with a discussion of the results.

7.1 Datasets

The increased availability of labelled datasets has been vital to the development of machine learning techniques, and to the growth of the machine learning community as a whole. There are now larg datasets from a wide range of domains publicly available for anyone interested in building and training their own machine learning models. Collecting datasets can be a tedious and costly process, particularly if it involves labelling, so the machine learning community benefits from sharing their datasets. Sharing allows researchers and practitioners to spend their time on research and development of better methods, and also makes experiments in research papers reproducible, and enables researchers to compare the performance of their algorithms by benchmarking on the same datasets. Datasets such as *MNIST* (LeCun and Cortes, 2010) and *ImageNet* (Deng et al., 2009) have pushed forward the rapid development in computer vision in the deep learning paradigm.

There are many publicly available sources for datasets for machine learning, including some hosted by major research institutions, and governments around the world. Open, domain specific datasets can now be found easily by doing some quick digging online. Below, we have listed three of the most popular and general dataset finders:

- *Kaggle* is an online machine learning community. In addition to offer a platform for finding and publishing datasets, Kaggle also lets you connect with and learn from other users, and participate in machine learning competitions.
- *UCI Machine Learning Repository* is a repository by the University of California, Irvine, containing hundreds of high quality datasets for machine learning. There are datasets available for both classification, regression, and clustering tasks, coming from many different domains. The datasets vary vastly in number of observations, and in number of features.
- *Google Dataset Search* was launched by Google in late 2018, aiming to make it easier to find and discover new datasets by making datasets from thousands of different repositories available through a unified search engine. The search engine came out of beta in January, 2020.

Table 7.1: Size (N) and dimensionality (D) of all the pre-processed datasets.

			Split		
Dataset	D	N	N_{train}	N_{val}	N_{test}
HEPMASS	21	525 123	315 123	35 013	174 987
MNIST (and binarised MNIST)	784	70 000	50 000	10 000	10 000

Choice of datasets and pre-processing

For our density estimation experiments, we use one relatively low-dimensional dataset describing particle collisions in a high energy physics experiment, in addition to the MNIST dataset (LeCun and Cortes, 2010) of handwritten digits. We use the same pre-processing of these datasets that was used in the experiments in (Papamakarios et al., 2017). This pre-processed version of these datasets has since publication become the de facto evaluation suite for benchmarking neural density estimators (Grathwohl et al., 2018; Huang et al., 2018; Durkan et al., 2019; Kobyzev et al., 2019). Note that Papamakarios et al. (2017) used in total seven different datasets, and we work with only two of them here; one dataset with 21 features, and one high-dimensional dataset with 784 features.

The first dataset listed below is taken from the UCI Machine Learning Repository. Discrete-valued features were removed from, in addition to features with a Pearson correlation coefficient greater than 0.98. Each of the remaining features was normalised by subtracting the sample mean and dividing by its sample standard deviation. Below, we provide a short description of the datasets we have used, and provide additional details on the pre-processing, following Papamakarios et al. (2017):

- **HEPMASS** (Baldi et al., 2016) is a dataset describing the outcomes of particle collisions in a high energy physics experiment. Positive samples in the dataset describe particle generating collisions, while the negatives come from background noise. Five features are removed because of to many reoccurring values that can lead to spikes in the density, giving misleading results when performing density estimation.
- **MNIST** (LeCun and Cortes, 2010) is perhaps the most famous dataset in the machine learning literature. It consists of 70000 grayscale images of handwritten digits. The resolution of each image is 28×28 pixels, and they are represented as 784 dimensional vectors where each vector element is in the range $(0, 1)$. The images are dequantised by adding uniform noise in the interval $[0, \frac{1}{256}]$, and transformed to the logit space. More details on this process are given in Section 7.3.

The UCI dataset was subsampled by Papamakarios et al. (2017) so that the product of the dimensionality and number of samples is approximately 10^7. HEPMASS was then split into training, validation, and test sets with splits as shown in Table 7.1. The way the MNIST data is provided by LeCun and Cortes (2010), it is already divided into a 50000/10000/10000 split for the training, validation, and test sets. We use the splits provided by the dataset in our experiments. An overview of the dimensionality and size of each pre-processed dataset is given in Table 7.1.

7.2 Reproducing results from the MADE paper

Because MADE is an integral part of the masked autoregressive flow presented in Chapter 5, we need to be confident that our implementation of MADE is working well in order to implement the masked autoregressive flows later on. As a sanity check of our MADE implementation, we try to reproduce some of the results from the original paper (Germain et al., 2015) on the MNIST dataset (LeCun and Cortes, 2010) of handwritten digits. Because the original MADE assumes binary inputs, we use the *binarised* MNIST dataset with pixel values only taking on the exact values 0 and 1, as originally curated and used by Salakhutdinov and Murray (2008).

Dataset

The dataset consists of 70000 images of handwritten digits, and is used for benchmarking models within a wide range of machine learning applications, including generative modelling, density

Table 7.2: Negative log-likelihood (NLL) on the test set (lower is better). Results from (Germain et al., 2015) in parentheses.

Model	**Hidden units**	**Optimiser**	**Input ordering**	$-\log p(\mathbf{x})$
MADE, 1 hidden layer	500	Adam	Random	93.93 (94.70)
MADE, 1 hidden layer	8 000	Adagrad	Natural	88.40 (88.40)

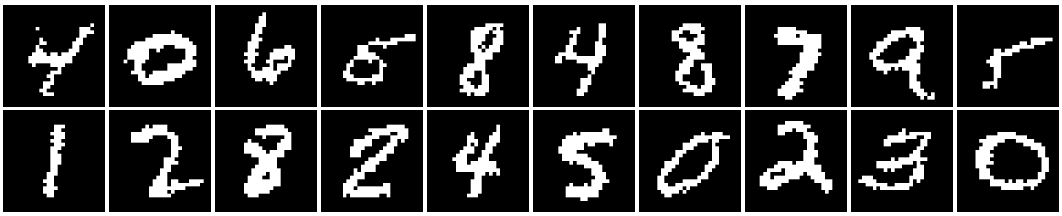


Figure 7.1: Random samples of handwritten digits from the binarised MNIST training set.

estimation, computer vision, and classification. The images are of size 28×28 pixels and are flattened and represented as 784-dimensional vectors. Like for the original MNIST, the dataset is randomly split into a training set of 50000 samples, a validation set of 10000 samples, and a test set of 10000 samples. Twenty random samples from the training set are shown in Figure 7.1.

Models and training

We reproduce the first result from Figure 2 in (Germain et al., 2015) using a MADE network with a single hidden layer of 500 units, and only one mask. In our experiments, we use an autoencoder with the same architecture, and train the network using the Adam optimiser with a learning rate of 0.001 that is decreased to 0.0001 after 50 epochs. Though not mentioned in the original paper, we found that using a random ordering of the inputs was necessary to achieve the reported results.

Further, we train a bigger MADE network with 8000 hidden units using the natural input ordering to reproduce results from Table 6 in (Germain et al., 2015). The model uses only one mask, i.e., we make no use of order-agnostic or connectivity-agnostic training. For this problem, we use the Adagrad optimiser with stepsize 0.001. Both models were trained using a batch size of 128, and early stopping with a patience of 30

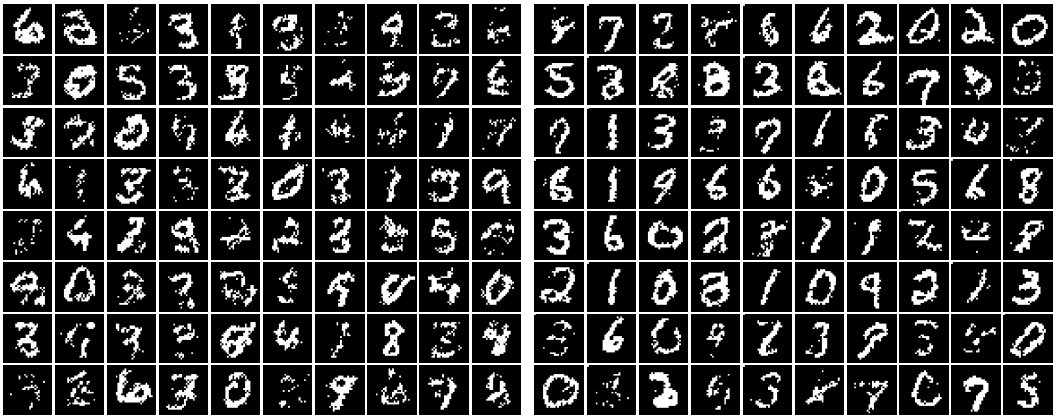
Results and discussion

The results of the experiments are shown in in Table 7.2, and measure up to the results reported by Germain et al. (2015). In fact, it seems like our implementation of the smaller model slightly outperforms the one from Germain et al. (2015) on this particular problem. This is likely because we use the Adam rather than the Adagrad optimiser, as we found Adam to be more efficient on the 500 hidden unit version of MADE. We deem the implementation to be successful.

Hyperparameters

The authors list the sets of values they tried for each hyperparameter in their experiments, but are not explicit about which hyperparameter settings they have used for each particular model. We could experiment with different combinations of the listed hyperparameter values to find the optimal combination for each model for each experiment (e.g. by using grid search or random search). Alas, the larger model with 8000 hidden units has $\sim 12 \times 10^6$ trainable parameters, making experimenting with different hyperparameters computationally expensive and slow using the computational hardware we had available.

Because of the aforementioned concerns, we did not prioritise searching for the optimal hyperparameter setting as soon as we had found one that yielded performance close to the reference values. It would also be informative to report the standard deviations of the reported test results as well, as is done in the supplementary materials of Germain et al. (2015). However, getting reasonable empirical standard deviations would require more runs than the five we used to compute the averages



(a) MADE, one hidden layer, 500 hidden units. (b) MADE, one hidden layer, 8000 hidden units

Figure 7.2: Samples from two different MADE models.

in Table 7.2, which would be very time consuming. All models exhibited a large degree of consistency between runs, so extrapolating from five runs, we suspect the standard deviations to be rather small.

7.2.1 Samples from the MADEs

Figure 7.2a and 7.2b show 60 random samples from the one hidden layer models with 500 hidden units and 8000 hidden units, respectively. Both models produce some samples that could just as well have been part of the original MNIST dataset. As we would expect from the test likelihoods, the quality of the samples improves with the size of the model. The bigger model produces a larger proportion realistic looking samples, and fewer samples that are just noise (like the ones seen in the upper right corner, row two and three).

We want to emphasise that MADE does not take the structure of an image into account. MADE is a general purpose density estimator, and the samples are generated as a 784 column vector following the autoregressive procedure described in Section 4.3.4 and illustrated in Figure 7.3. It is a bit artificial to generate images pixel by pixel, starting from the top left corner. For more complex images than the handwritten grey scale digits in MNIST (higher-dimensional, multiple colour channels, more detailed motives), this naive approach fails to generate good samples. There are other autoregressive neural networks with convolutional layers that are more suitable for generating images specifically, such as PixelCNN (van den Oord et al., 2016), and PixelCNN++ (Salimans et al., 2017).

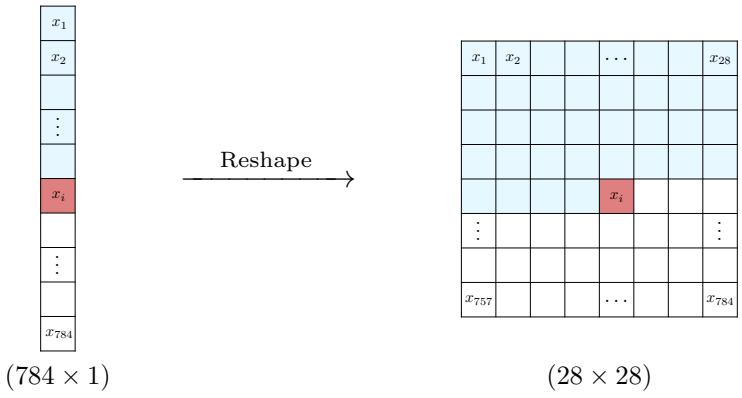


Figure 7.3: Sampling procedure using MADE.

Figure 7.4: Reconstructed digits.

7.2.2 MADE as an autoencoder

We carry out an additional experiment that is not conducted in the original paper. We use the masked autoregressive autoencoder as an autoencoder in the sense that we investigate how well the MADE is able to reconstruct its inputs. A reconstructed sample is obtained by using a sample from the test set as input to the model, and sampling a new image from the resulting set of conditional distributions. Strictly speaking, the outputs $\hat{x}_i$ of the MADE are the reconstructed inputs. But while the input to the model is binary, each output of the model is real on the interval $(0, 1)$, as they represent probabilities. To get a binary valued reconstruction, we thus sample from the corresponding conditional Bernoulli distributions defined by each $\hat{x}_i$.

Figure 7.4 shows ten reconstructed samples from a one hidden layer model with 8000 hidden units using natural ordering of the inputs. The top row shows ten samples from the test set, the middle row shows the corresponding set of conditional distributions from the MADE (dark blue indicates a probability close to one), and the bottom row shows the sampled reconstructed inputs. The second row is strikingly similar to the first row, as we would expect, as this is what we optimised for during training of the model. The samples in the third row are a bit more noisy, but it is in all cases easy see which digit each sample resembles. A comparison with existing work is not possible because this experiment was not done in the original paper, but qualitatively the model performs well on this task too, and we deem our implementation successful.

An artefact of the MADE model that becomes visible in this experiment is that the top left pixels in the second row of Figure 7.4 seem to be learning slower than the other pixels. This is seen by noticing that each top left pixel (corresponding to x_1 given the natural ordering) has a stronger shade of blue than its neighbouring pixels. This is easier seen in Figure 7.5. Revisiting Equation (4.10) and Figure 4.4, we observe that all connections going into this neuron are masked out to, meaning that the only trainable parameter in this particular output neuron is a scalar bias term. Because this bias term is the only parameter contributing to estimate the output distribution $p(x_1)$, the training of this particular unit slows down compared to the other units, which have several trainable weights.

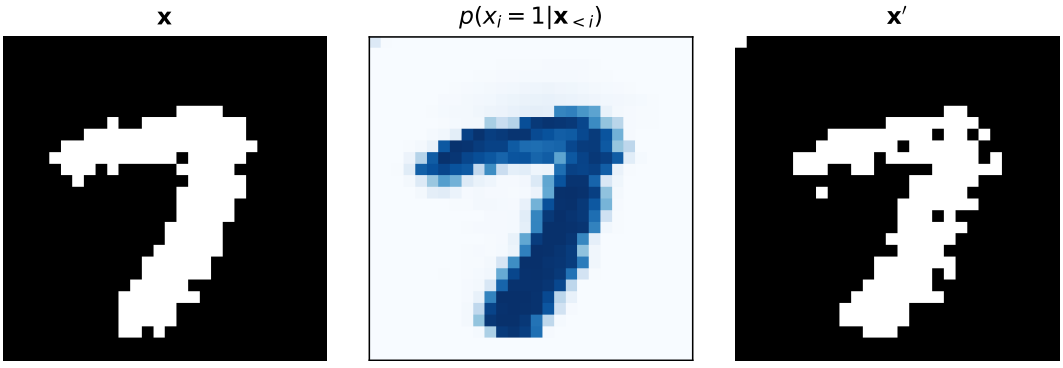


Figure 7.5: Closeup of a reconstructed digit. Note the colour of the top left pixel in the reconstruction.

Table 7.3: Details about the MADE model, and test log-likelihood in logit space. Inside the parenthesis is the result from (Papamakarios et al., 2017) with error bars corresponding to two standard deviations.

Model	Hidden units	Learning rate	Test log-likelihood
Gaussian MADE	1024	0.001	-1379.97 ± 1.79 (-1380.8 ± 4.8)

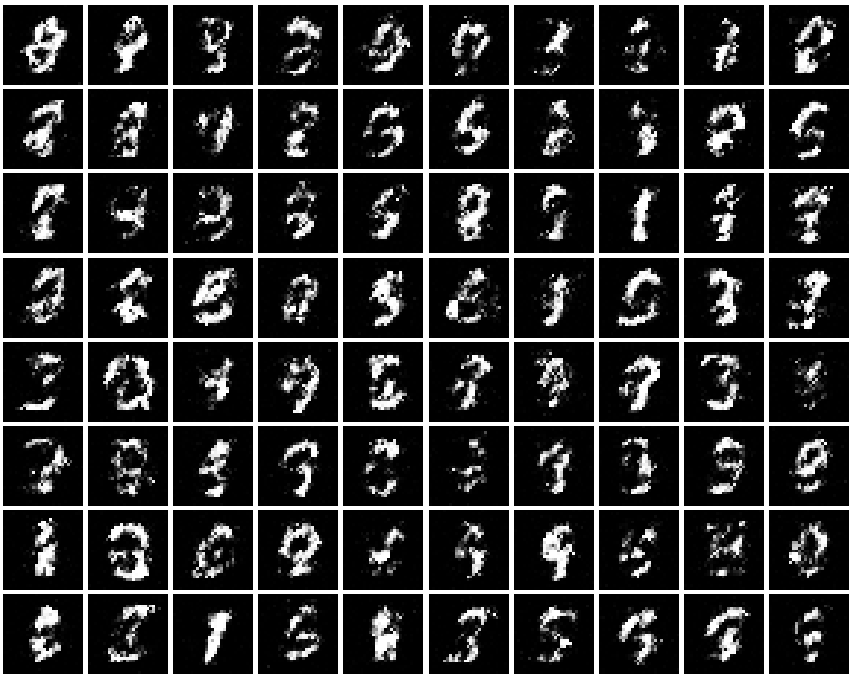


Figure 7.6: Samples from the Gaussian MADE.

As $p(x_1)$ is not dependent on any variables in the input $\mathbf{x}$, the parameter of $p(x_1)$ is only a property of the trained model itself, and is therefore the same across all ten samples in Figure 7.4. This was confirmed through inspection of the weights and biases of the model, and the particular value of the bias was found to be $b = -0.868$ (corresponding to $\sigma(b) = 0.1338$ being the parameter of the Bernoulli distribution $p(x_1)$). This non-zero probability of the pixel in the top left corner not being black is the reason why we can see a white pixel in the top left corner of the reconstruction in Figure 7.5. This pixel clearly should have been black, but this error will occur in approximately 13% of the reconstructed images from this model.

This observation speaks in favour of using training the model using multiple different orderings of the inputs and averaging the outputs at test time. Because all other orderings would assign a close-to-zero probability for the top left pixel of being white, the resulting average probability $p(x_1)$ would also be very close to zero.

7.3 Gaussian MADE

We carry out a simple and similar experiment with the Gaussian MADE described in Section 4.4. There are no experiments using Gaussian conditionals in the original MADE paper, as the Gaussian MADE is not described there at all. To validate the performance of our model, we instead follow the experimental setup from (Papamakarios et al., 2017) where a one-layer Gaussian MADE with 1024 hidden units is trained on the MNIST dataset.

Additional pre-processing

Here, the regular MNIST dataset is used instead of the binarised MNIST, because using Gaussian conditionals allows for modelling real-valued vectors. The pixel values in the original MNIST take on

values in the interval $[0, 1)$, but with most values being very close to 1 (white) or very close or exactly equal to 0 (black). These pixel values are actually discrete valued, because they are transformed from an 8 bit representation of the black and white images, which allows each pixel to take on values in the range $[0, 255]$. The transformed pixels have discrete values $0, \frac{1}{256}, \ldots, \frac{255}{256}$.

To transform this discrete distribution of pixel values into a continuous distribution that lends itself to being approximated by Gaussian densities, we first have to *dequantise* the pixel values by adding noise:

$$\hat{\mathbf{x}} = \mathbf{x}_Q + \frac{\mathbf{z}}{256}$$

where $\mathbf{z} \sim \text{Uniform}(0, 1)$, for each quantised image $\mathbf{x}_Q$. Following the approach in (Papamakarios et al., 2017), we transform the dequantised images to the logit space as:

$$\mathbf{x} = \text{logit}(\lambda + (1 - 2\lambda) \cdot \hat{\mathbf{x}})$$

with $\lambda = 10^{-6}$. We train the model on pixel values in the unconstrained logit space instead in the domain from 0 to 1, as the Gaussian conditionals have support on the entire real line. Consequently, we evaluate the log-likelihoods in the logit space as well, and this allows us to compare our results to the results in Table 2 in (Papamakarios et al., 2017).

Training and results

We use the Adam optimiser with a learning rate of 0.001. To prevent overfitting, we use l_2 regularisation with coefficient $\lambda = 10^{-6}$, such that the sum $\lambda \sum_i w_i^2$ is added to the training objective, penalising large weights w_i. We also use early stopping with a patience of 30, i.e., we stop training if the validation loss has not decreased for 30 consecutive training epochs. We train the model without batch normalisation, as this is a model with only one hidden layer.

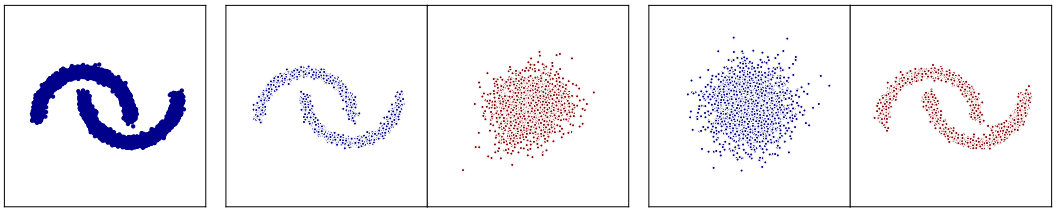
The result of the training is shown in Table 7.3. We achieve an average test log-likelihood similar to the one reported by Papamakarios et al. (2017), and well inside the error bars corresponding to two standard deviations. The standard deviations are computed from the sample variance across the test set, using one model only. This is the same way the standard deviations are computed in the original paper. Note that the log-likelihood in Table 7.3 can not be compared with the log-likelihood from the Bernoulli MADE on the binarised MNIST, as we have used different datasets, and different distributions to model the pixels of the images.

We can, however, compare the sample quality between the models. Figure 7.6 shows 80 samples from the Gaussian MADE. The samples are qualitatively more similar to the samples from the smaller Bernoulli MADE in Figure 7.2a with 500 hidden units, than to the samples from the larger Bernoulli MADE with 8000 hidden units in Figure 7.2b. This is as expected, considering the model size of 1024 hidden units, and that the MNIST dataset used here is more complex than the binarised MNIST used in the previous section.

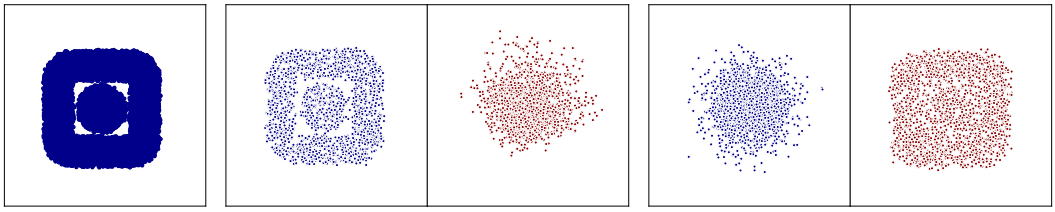
7.4 2D density estimation with Real NVP

We will proceed to use Real NVP for estimation of high-dimensional densities in the next section, but we first learn a two-dimensional density from samples to verify that the model works for low-dimensional cases. Working with two-dimensional densities is convenient because it allows for neat and intuitive visualisations, but this experiment serves more as a sanity check of the model and a stepping stone towards the large datasets we will use in the next section. In this section, we use the "two moons" dataset, a toy 2D dataset included in the Python library SciKit-learn (Pedregosa et al., 2011), and a more unorthodox "density", created from the NTNU logo.

We define each coupling layer using two small neural networks with one hidden layer each to model the scaling function $s(\cdot)$ and shifting function $t(\cdot)$ in each coupling layer. Both networks have 200 hidden units using the ReLU non-linearity, and these architecures are the same for the two moons model and the NTNU model. The partitioning of a two-dimensional input vector to each coupling layer is trivial, as there are only two possible ways to order the elements of a two-dimensional vector. Hence, one element is copied, and the other element is transformed in each layer. The ordering of the input is reversed for every successive layer. To model the two moons dataset, we use a flow with 32 coupling layers, and to model the NTNU dataset, we use a deeper flow with 48 coupling layers.



(a) The two moons density.



(b) The "NTNU density".

Figure 7.7: **Left:** 10000 samples from the target density. **Middle:** Inverse pass $\mathbf{x} \mapsto \mathbf{u}$ of 2000 samples from the target density. **Right:** Forward pass $\mathbf{u} \mapsto \mathbf{x}$ of 2000 samples from the Gaussian base density.

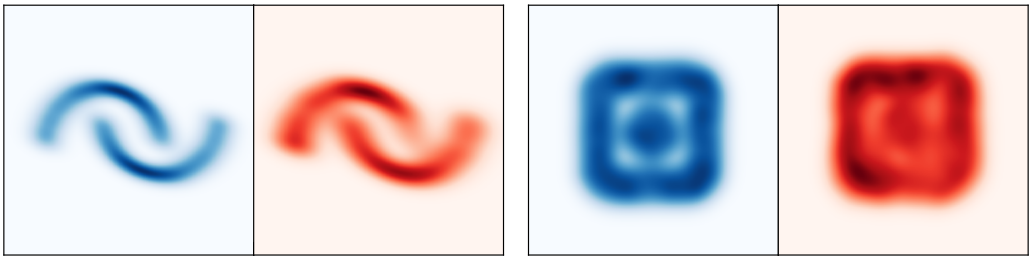


Figure 7.8: Kernel density plots of the two densities, estimated from 10000 random samples. Target density to the left, and estimated density to the right.

We sample 5000 samples from each density to use for training, and train each model for a fixed number of 300 epochs using a batch size of 256. Batch norm is added after each coupling layer, using running averages with $\beta = 0.9$. We use the Adam optimiser with fixed learning rates of 2×10^{-4} and 1×10^{-5} for the two-moons, and the NTNU datasets, respectively. The results are shown in Figure 7.7. The (very) astute reader will recognise the two-moons dataset from Figure 3.2 in Chapter 3, which was indeed created using the same Real NVP model that is described in this section. No likelihood values are reported, as there are no comparable results in the literature, and these likelihood values are also unlikely to be of value as a future reference for others.

The Real NVP models are able to learn a bijective mapping between the target density and the base density quite well. The performance is qualitatively better on the two moons dataset compared to the performance on the NTNU dataset. The output of the data to noise transformation in Figure 7.7b does not seem to be exactly Gaussian, and the generated samples do not reflect the white gaps between the "outer fram" and the inner circle in the target density very well. Scatterplots are, however, not very good at revealing areas of high or low densities due to overlapping plotted points. For this reason, we add an additional kernel density plot visualising the estimated densities.

These plots are shown in Figure 7.8, and the plots to the right reveal that the model has estimated the NTNU density better than what it appears like based on the scatterplot in Figure 7.7 alone. As desired, there is a clearly defined region with lower density around the middle circle, though the density is not as low as it should have been when comparing with the target density. The two moons dataset is estimated accurately, judging from both the scatterplot and the density plot, and we deem the performance of both models to be satisfactory.

Finally, we point out that the Real NVP is not particularly well-suited for two-dimensional density estimation, given the simplicity of each transformation. For two-dimensional densities, traditional approaches work just as well or better, and are likely to be more parameter-efficient than a normalising flow. As mentioned earlier, each transformation in the coupling layer uses two neural networks to scale and shift only one dimension of the input in each layer, which must be said to be a horribly inefficient use of computing power. Where normalising flows shine, is in the high-dimensional regime, i.e., when dealing with high-dimensional data, and this will be illustrated in the next section.

Remark: To create a density based on the NTNU logo, we start out with an image of the logo of size 339 by 318 pixels that we turn into a binary image with pixel values 1 and 0. We then create a grid of the same size as the image, and obtain the coordinates of the one-valued pixels. These coordinates are scaled and shifted to be centered around zero, and they describe the points in the plane coming from the "NTNU density", so that each pair of coordinates represent a sample. To avoid repetitions when drawing samples from the NTNU density, a small amount of uniform noise from the interval $(-0.5, 0, 5)$ is added to each coordinate at the time of sampling, ensuring unique draws.

7.5 Density estimation with HAF, MAF, and Real NVP

Through the initial experiments above, we have verified that our implementations of the Gaussian MADE and Real NVP perform as expected from the literature. We experiment with the masked autoregressive flow and Real NVP for density estimation in the following, reproducing some results from the masked autoregressive flow paper (Papamakarios et al., 2017) for the MNIST and HEPMASS datasets. The authors carry out a comparison of MADE, Real NVP, and MAF on a range of density estimation tasks. We extend two of these experiments to include our hybrid autoregressive flow, and compare our model to the existing and closely related models.

Papamakarios et al. (2017) also include MADEs and MAFs where each conditional is modelled as a mixture of ten Gaussians, and ten-layer MAF and Real NVPs in their comparison. Extending a five layer model to a ten layer models is straightforward, but training the deeper models is a matter of how much computational resources and time that is available. We restrict ourselves to the five layer models, and a single Gaussian MADE. The MADE using a mixture of Gaussians is a natural extension of the Gaussian MADE, which can be viewed as a special case of a Gaussian mixture with only one component.

7.5.1 Models

MADE: We implement a single layer Gaussian MADE which serves as a baseline in each experiment. The MADE uses the natural ordering of the inputs, and ReLU hidden units. The implementation is as described in Section 4.4, with no additional tricks like order-agnostic or connectivity-agnostic training.

Real NVP: A large part of the Real NVP paper (Dinh et al., 2017) is about generative modelling of images, incorporating knowledge about image structure into the model by using convolutional neural networks to model both $s(\cdot)$ and $t(\cdot)$. The layers of these networks are modified with spatial and channel-wise binary masks to generate the partitioning in each coupling layer. In this thesis, we abandon this specialised architecture used for modelling images, and rather use an implementation of Real NVP for general-purpose density estimation, as is done in (Papamakarios et al., 2017).

This version of Real NVP takes in, and returns a vector. We alternate between transforming the even-indexed and the odd-indexed elements of the input in every other coupling layer. In the first layer, the odd-indexed elements are copied and the even-indexed elements are transformed. In the next layer, the even-indexed elements are copied, and the odd-indexed elements are transformed. This pattern is repeated throughout the flow. Note that this approach requires a permutation of the rows before each coupling layer to preserve the autoregressive property. This is a bijective transformation with an abs-det-Jacobian equal to one, hence row permutation does not affect the likelihood computation according to the change of variables formula.

We model the scaling function $s(\cdot)$ as a fully-connected feed-forward network using the hyperbolic tangent activation function in the hidden layers, and we model the translation function $t(\cdot)$ as a fully-connected feed-forward network using ReLU activation function in the hidden layers. Both networks have unconstrained, linear outputs. This is the same choice of activation functions as used

Table 7.4: Average test log-likelihood in logit space (higher is better) with error bars corresponding to two standard deviations. The best performing model among the candidates is highlighted in bold.

MNIST			
Model	**L×H**	**Test log-likelihood**	**Papamakarios et al.**
MADE	1×1024	-1379.97 ± 1.79	$\mathit{-1380.8 \pm 4.8}$
Real NVP, 5 layer	1×1024	-1310.13 ± 4.54	$\mathit{-1323.2 \pm 6.6}$
MAF, 5 layer	1×1024	-1296.09 ± 3.62	$\mathit{-1300.5 \pm 1.7}$
HAF, 5 layer	1×550	$\mathbf{-1292.91 \pm 2.57}$	–
HAF, 5 layer	1×700	-1297.64 ± 4.14	–

Table 7.5: Average test log-likelihood (higher is better) with error bars corresponding to two standard deviations. The best performing model among the candidates is highlighted in bold.

HEPMASS			
Model	**L×H**	**Test log-likelihood**	**Papamakarios et al.**
MADE	1×512	-22.40 ± 0.05	$\mathit{-20.98 \pm 0.02}^*$
Real NVP, 5 layer	1×512	$\mathbf{-17.80 \pm 0.05}$	$\mathit{-19.62 \pm 0.02}^*$
MAF, 5 layer	1×512	-18.80 ± 0.07	$\mathit{-17.70 \pm 0.02}^*$
HAF, 5 layer	1×256	-19.18 ± 0.04	–
HAF, 5 layer	1×350	-19.64 ± 0.05	–

by Papamakarios et al. (2017) in their implementation of Real NVP. Both networks are of the same size with number of hidden layers (L) and number of hidden units (H) as given in Table 7.4 and 7.5.

MAF: For the masked autoregressive flow, we reproduce the experimental setting from (Papamakarios et al., 2017). We use a five layer MAF, where each layer is a Gaussian MADE with ReLU activation functions. We use the natural ordering of the data for the first layer in the flow, and reverse the order before each of the following layers. The size $L \times H$ refers to the size of each MADE network in the model.

HAF: We implement each layer following the procedure described above. The MADE is a single layer MADE with ReLU hidden units, the translation network has ReLU hidden units, while the scaling network has hyberbolic tangent hidden units. For simplicity, all networks chosen to have the same size $L \times H$. The Real NVP part of the model behaves as described above, by alternating between transforming the odd-indexed and even-indexed elements, while the MADE part of the model reverses the order of inputs after every two layers.

On each dataset, we implement two HAFs; one with approximately 50% as many hidden units in each network as the other models, and one with approximately 70% as many hidden units as the other models. These choices follow from the discussion in Chapter 6, and are made to allow for a fair comparison between the competing models.

7.5.2 Training

All models are trained using the Adam optimiser and added l_2 regularisation with coefficient $\lambda = 10^{-6}$ to prevent overfitting. For the MAF, Real NVP, and HAF, we use a learning rate of 0.0001, whereas for MADE, we use a larger learning rate of 0.001. We use a batch size of 100 in all experiments, and add batch norm between all layers in the flow, and between the last layer and the base density, which is chosen to be a standard Gaussian for all models. In the batch norm layers, we choose the $\epsilon = 10^{-5}$ to ensure numerical stability, and use mean and variance computed over the entire training set at validation and test time.

The models are using early stopping with a patience of 30, so the training terminates after 30 consecutive training epochs without improving the performance on the validation set, and the best performing model on the validation set is used to compute the test loss. The test loss is reported as the

average loss over all test samples, with error bars corresponding to two empirical standard deviations. All models were trained on exactly the same datasets with the same train/validation/test splits.

7.5.3 Results and discussion

A summary of the test results is presented in Table 7.4 and Table 7.5. The MADE result from Section 7.3 is repeated here to allow for easy comparison.

For the **MNIST** dataset, we observe that the MADE result is on par with the performance of the equivalent model in the masked autoregressive flows paper, but that our versions of MAF and Real NVP seem to yield slightly better results than their counterparts in the literature. These differences do not change the order of the models in terms of performance, with MAF coming out on top, followed by Real NVP and MADE. We address the reasons why the results might differ in Section 7.5.4.

We also observe that the smallest HAF model with a layer size of 550 outperformed the other models, including the larger HAF with 700 hidden units in each layer. There is no linear relationship between the size of a model and its performance on a given dataset, and this was also seen in (Papamakarios et al., 2017), where the five layer MAF and Real NVP both outperformed their ten layer counterparts on MNIST. It is still interesting to see that HAF performs better than MAF and Real NVP on this problem, using a comparable number of parameters. This encourages further testing of the model.

On the **HEPMASS** dataset, both MAF and Real NVP perform better than the HAF, but the results are still comparable. Again, our Real NVP perform better than reported by (Papamakarios et al., 2017). An important thing to note is that the results from (Papamakarios et al., 2017) in Table 7.5 are marked with an asterix to show that their reported results may also come from models with network 2×512. In their paper, all models were given two options on the HEPMASS dataset; 1×512 and 2×512. The best model was picked based on validation performance, but the paper does not state which model that generated the reported results. We chose to only try out the shallower of the two architectures, but it is likely that the deeper choice would yield better performance for one or more of the models. We attribute the differences in performance for MAF and MADE largely to this fact.

Further experimentation is needed in order to be able to draw conclusions about the hybrid autoregressive flow, but the model shows promising performance in these initial experiments when compared to popular neural density estimators like MAF and Real NVP, given similar circumstances. A natural extension would be to compare the models on the remaining five datasets in the density estimation evaluation suite by (Papamakarios et al., 2017). The tendency in the experiments thus far, is that the number of hidden layers should not be much larger than the number of inputs, as the smaller HAF outperformed the larger one in both experiments.

If this is an actual effect that is valid for HAF across datasets, we suspect that it might also be valid for Real NVP. This could have implications for the experimental setup when comparing Real NVP to MAF. Though the setup is originally designed by Papamakarios et al. (2017) to make a fair comparison between the MAF and Real NVP, it might be that the experimental setup is disadvantageous for the Real NVP[1]. By using the same number of hidden units in each of the networks in Real NVP and MAF, it might be that the Real NVP is not set up to make the most out of its available parameters. Not only does an overparameterisation give an inefficient use of the parameters, but it could also directly deteriorate the performance of the model, as observed for the HAF.

Lastly, we want to emphasise that the error bars are computed from the test run of a single model, as is also done in (Papamakarios et al., 2017). While the average test loss was observed to be consistent across runs, the error bars fluctuated more, but remained on the same order of magnitude. Because of this, we do not attribute too much significance to the width of the error bands.

7.5.4 Remarks on the implementation of MAF and Real NVP

The source code for the experiments carried out in (Papamakarios et al., 2017) is openly available on GitHub. However, their implementation is written in the now-deprecated deep learning framework Theano, which has made a direct comparison of our and their code more difficult when debugging,

[1] E.g., the width of the scaling and shifting networks on MNIST is set to be 1024, but there are only 392 inputs to each network. A network width of 1024 makes more sense in the MAF, as each network has a full number of 784 inputs, resulting in what, based on our observations, might be a better relation between the size of the input and the size of the hidden layer.

as the syntax and structure of Theano code is very different from PyTorch code. While Theano was deprecated back in 2017, PyTorch is a recent framework updated with the current best practices. There might be subtle differences hidden within the frameworks (e.g., different default parameter settings or initialisations) that are difficult to pick up, so it is not uncommon to obtain slightly different results when implementing the same deep learning model in different frameworks.

Nevertheless, we made an attempt to trace down the differences between our implementations that cause the differences in test results. It is likely that the differences in performance between their and our **MAF** model on MNIST can be attributed to different weight initialisations. Weight initialisation is not mentioned in the paper, but by inspecting the code of (Papamakarios et al., 2017), we find that they have used a manually modified version of the LeCun normal initialisation (LeCun et al., 1998) in the MADE networks that make up the MAF, whereas we have used the uniform Kaiming initialisation (He et al., 2015) that is default for `Linear` (fully connected) layers in PyTorch. Considering that using Kaiming initialisation is considered best practice when using ReLU activation functions, and that our implementation of MAF actually perform slightly *better* than the ones in the paper, we stick to using the Kaiming initialisation in all our experiments.

We also investigated their **Real NVP** implementation to find out what could explain the differences in our results. In addition to using the same LeCun normal initialisation as above, Papamakarios et al. (2017) have also implemented the coupling layers in a different way than us. We illustrate the differences below by considering one coupling layer $f^{-1} : \mathbf{x} \to \mathbf{u}$ for a D-dimensional input $\mathbf{x}$. We copy the odd-indexed elements and transform the even-indexed elements. We achieve this partitioning by using an alternating binary mask $\mathbf{m} = (0, 1, 0, 1, ..., 0, 1)^T$ on the input.

Papamakarios et al. (2017) define $\mathbf{x}_1 = \mathbf{x} \odot \mathbf{m}$ and $\mathbf{x}_2 = \mathbf{x} \odot (1 - \mathbf{m})$, so that $\mathbf{x}_1$ and $\mathbf{x}_2$ are still D-dimensional, but with every second element zeroed out. They then use scaling and translation networks $s, t : \mathbb{R}^D \to \mathbb{R}^D$ with one hidden layer of 1024 hidden units. With $\mathbf{s} = s(\mathbf{x}_1)$ and $\mathbf{t} = t(\mathbf{x}_1)$, the output of the coupling layer is defined as:

$$\mathbf{u} = \mathbf{x}_1 + (1 - \mathbf{m}) \odot (\mathbf{x}_2 - \mathbf{t}) \odot \exp(-\mathbf{s}).$$

On the other hand, in our implementation we define $\mathbf{x}_1$ and $\mathbf{x}_2$ by considering $\mathbf{m}$ as a boolean vector. The following is written in pseudo-code:

$$\begin{aligned}\mathbf{x}_1 &= \mathbf{x}[\mathbf{m}] \\ \mathbf{x}_2 &= \mathbf{x}[\sim\mathbf{m}],\end{aligned}$$

where the tilde denotes logical negation, and the brackets are used for logical indexing of the vector. This way, we get vectors $\mathbf{x}_1, \mathbf{x}_2$ which are only $D/2$-dimensional. The scaling and translation networks are then defined as $s, t : \mathbb{R}^{D/2} \to \mathbb{R}^{D/2}$. With $\mathbf{s}$ and $\mathbf{t}$ defined as before, the output of the coupling layer is then defined as:

$$\begin{aligned}\mathbf{u} &\leftarrow \mathbf{x} \\ \mathbf{u}[\sim\mathbf{m}] &\leftarrow (\mathbf{x}_2 - \mathbf{t}) \odot \exp(-\mathbf{s}).\end{aligned}$$

This may seem like a subtle implementational difference compared to (Papamakarios et al., 2017), but it is in fact significant in terms of memory usage. To see this, consider the number of parameters in a one hidden layer neural network. Ignoring bias terms, the number of parameters is given by $n_{in} \times n_{hidden} + n_{hidden} \times n_{out}$. By reducing the size of the input and output from D to $D/2$, we effectively reduce the number of parameters in the network by 50%. Multiplying this by two neural network in each coupling layer, times the number of layers in the flow, the number of parameters saved in our particular model is approximately:

$$0.5 \times (2 \times K \times n_{hidden} \times (D + D)) = 0.5 \times 2 \times 5 \times 1024 \times (784 + 784) \approx 8 \times 10^6.$$

Note that we still transform the same number of elements with exactly the same dependencies as Papamakarios et al. (2017), but our implementation is much more memory and time efficient than the original. Judging from the results, it seems like this implementation also might result in more efficient training of the parameters, and better performance.

Chapter 8

Summary and Outlook

Summary

In this thesis, we have motivated the use of *normalising flows* for improving the model performance within a range of fields in statistics. We have built up the theory about normalising flows stone by stone, motivated by their usefulness in variational inference and high-dimensional density estimation in particular. We proceeded to present the highly influential class of *autoregressive normalising flows*, bridging the classical field of autoregressive density modelling and modern normalising flows. We presented the MADE, and three different, but still closely related, flows with different strengths and weaknesses, followed by a comparison of the three, and a brief presentation of a more flexible flow, pointing out a direction for normalising flows research going forwards.

Next, we proposed a new autoregressive flow, the *hybrid autoregressive flow*, heavily inspired by Real NVP and MAF, and presented the fundamental theory concerning the proposed transformation. The presented models for density estimation were put to the test in the following chapter, including the proposed flow. Its performance was compared to the performance of existing flows in a standardised experimental setting. The HAF showed competitive performance with the existing autoregressive flows on the two datasets we have benchmarked them on. In one case, the proposed flow even outperformed MAF and Real NVP. The hybrid autoregressive flow can be added to the toolkit of the practitioner that seeks to use a neural density estimator for evaluation of high-dimensional densities.

Future research

Further testing of the hybrid autoregressive flow is required to draw conclusions about its capabilities relative to the other autoregressive flows. Particularly, more experimentation with number and sizes of hidden layers in each of the three neural networks making up the HAF layer is needed to understand the relationships between them. We used the same number of hidden units in all networks in the hybrid coupling layer, but this might not be the most efficient use of parameters for this model.

A deeper theoretical understanding of what the limitations of the different flows is lacking for most models, and so also for the hybrid autoregressive flow. Whether flows such as Real NVP, MAF, and HAF are universal density estimators remain open questions. Getting general insights about how get the most out of each flow by tuning the hyperparameters and architectures is also challenging, as such design choices are highly dataset specific.

The "holy grail" of research on normalising flows, is a flow that is able to efficiently evaluate densities, efficiently generate new samples, and do both tasks with the performance of the best alternative models out there. Current models typically sacrifice one of the three aforementioned properties to succeed on the two others. Designing new flows is a thus still a natural avenue of future research. Autoregressive flows has up until recently been the most successful class of models, but going forward, it is likely that other classes will be more influential in pushing the field further. The Neural ODE paper (Chen et al., 2018) won the prestigious best paper award at NeurIPS in December 2018 using *continuous-time* normalising flows, and could be an omen of what the future will bring.

As new and better flows are proposed, the applications of flows to other fields in statistics can be improved concurrently, also outside pure density estimation and variational inference. A range of such applications was mentioned in Chapter 1. Considering that the interest in the field of normalising flows is still rising and the relative youth of the field, there is reason to believe that further advancements towards even more expressive tractable flows will be made in the near future.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Baldi, P., Cranmer, K., Faucett, T., Sadowski, P., and Whiteson, D. (2016). Parameterized neural networks for high-energy physics. *The European Physical Journal C*, 76(5):235.

Bauer, M. and Mnih, A. (2019). Resampled priors for variational autoencoders. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 89 of *Proceedings of Machine Learning Research*, pages 66–75. PMLR.

Bjorck, N., Gomes, C. P., Selman, B., and Weinberger, K. Q. (2018). Understanding batch normalization. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 7694–7705. Curran Associates, Inc.

Blei, D. M., Kucukelbir, A., and McAuliffe, J. D. (2016). Variational Inference: A Review for Statisticians. *arXiv e-prints*, page arXiv:1601.00670.

Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*.

Chen, T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. K. (2018). Neural ordinary differential equations. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 6571–6583. Curran Associates, Inc.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

Dinh, L., Krueger, D., and Bengio, Y. (2014). Nice: Non-linear independent components estimation. *CoRR*, abs/1410.8516.

Dinh, L., Sohl-Dickstein, J., and Bengio, S. (2017). Density estimation using real nvp.

Durkan, C., Bekasov, A., Murray, I., and Papamakarios, G. (2019). Neural spline flows. In Wallach, H., Larochelle, H., Beygelzimer, A., dAlché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 7511–7522. Curran Associates, Inc.

Evans, M. and Rosenthal, J. (2009). *Probability and Statistics: The Science of Uncertainty*. W. H. Freeman.

Frey, B. J., Hinton, G. E., and Dayan, P. (1996). Does the wake-sleep algorithm produce good density estimators? In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems 8*, pages 661–667. MIT Press.

Germain, M., Gregor, K., Murray, I., and Larochelle, H. (2015). Made: Masked autoencoder for distribution estimation. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 881–889, Lille, France. PMLR.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Grathwohl, W., Chen, R. T. Q., Bettencourt, J., Sutskever, I., and Duvenaud, D. (2018). FFJORD: free-form continuous dynamics for scalable reversible generative models. *CoRR*, abs/1810.01367.

Graves, A. (2011). Practical variational inference for neural networks. In Shawe-Taylor, J., Zemel, R. S., Bartlett, P. L., Pereira, F., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 24*, pages 2348–2356. Curran Associates, Inc.

Härdle, W. K. and Simar, L. (2015). *Theory of the Multinormal*, pages 183–199. Springer Berlin Heidelberg, Berlin, Heidelberg.

He, H. (2019). The state of machine learning frameworks in 2019. https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow\-dominates-industry/. Visited: 1 February 2020.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852.

Hernandez-Lobato, J., Li, Y., Rowland, M., Bui, T., Hernandez-Lobato, D., and Turner, R. (2016). Black-box alpha divergence minimization. In Balcan, M. F. and Weinberger, K. Q., editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1511–1520, New York, New York, USA. PMLR.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. arXiv: 1207.0580.

Ho, J., Chen, X., Srinivas, A., Duan, Y., and Abbeel, P. (2019). Flow++: Improving flow-based generative models with variational dequantization and architecture design. *CoRR*, abs/1902.00275.

Hoffman, M., Sountsov, P., Dillon, J. V., Langmore, I., Tran, D., and Vasudevan, S. (2019). Neutralizing bad geometry in hamiltonian monte carlo using neural transport. arXiv: 1903.03704.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366.

Huang, C.-W., Krueger, D., Lacoste, A., and Courville, A. (2018). Neural autoregressive flows. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2078–2087, Stockholmsmässan, Stockholm Sweden. PMLR.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167.

Jordan, M. I., Ghahramani, Z., Jaakkola, T. S., and Saul, L. K. (1999). An introduction to variational methods for graphical models. *Machine learning*, 37(2):183–233.

Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. *arXiv e-prints*, page arXiv:1412.6980.

Kingma, D. P. and Dhariwal, P. (2018). Glow: Generative flow with invertible 1x1 convolutions. arXiv: 1807.03039.

Kingma, D. P., Salimans, T., Jozefowicz, R., Chen, X., Sutskever, I., and Welling, M. (2016). Improving Variational Inference with Inverse Autoregressive Flow. *arXiv e-prints*, page arXiv:1606.04934.

Kingma, D. P. and Welling, M. (2013). Auto-Encoding Variational Bayes. *arXiv e-prints*, page arXiv:1312.6114.

Klenke, A. (2013). *Probability Theory: A Comprehensive Course*. Universitext. Springer London.

Kobyzev, I., Prince, S., and Brubaker, M. A. (2019). Normalizing flows: An introduction and review of current methods.

Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86.

Larochelle, H. and Murray, I. (2011). The neural autoregressive distribution estimator. In Gordon, G., Dunson, D., and Dudík, M., editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 29–37, Fort Lauderdale, FL, USA. PMLR.

LeCun, Y. (2018). The Next AI Revolution Will Not Be Supervised. https://www.nsf.gov/events/event_summ.jsp?cntn_id=245179&org=NSF. Visited: 26 January 2020.

LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998). Efficient backprop. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, page 9–50, Berlin, Heidelberg. Springer-Verlag.

LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. Visited: 10 November 2019.

Li, Y. and Turner, R. E. (2016). Rènyi Divergence Variational Inference. *arXiv e-prints*, page arXiv:1602.02311.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

McLachlan, G. J. and Basford, K. E. (1988). *Mixture models: Inference and applications to clustering.* Marcel Dekker, New York.

Mohamed, S., Rosca, M., Figurnov, M., and Mnih, A. (2019). Monte Carlo Gradient Estimation in Machine Learning. arXiv: 1906.10652.

Müller, T., McWilliams, B., Rousselle, F., Gross, M., and Novák, J. (2018). Neural importance sampling. *CoRR*, abs/1808.03856.

Papamakarios, G. (2019). *Neural Density Estimation and Likelihood-free Inference*. PhD thesis, University of Edinburgh.

Papamakarios, G., Nalisnick, E., Rezende, D. J., Mohamed, S., and Lakshminarayanan, B. (2019). Normalizing flows for probabilistic modeling and inference.

Papamakarios, G., Pavlakou, T., and Murray, I. (2017). Masked autoregressive flow for density estimation. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 2335–2344, Red Hook, NY, USA. Curran Associates Inc.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library.

Pati, D., Bhattacharya, A., and Yang, Y. (2017). On Statistical Optimality of Variational Bayes. *arXiv e-prints*, page arXiv:1712.08983.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Rezende, D. J. and Mohamed, S. (2015). Variational Inference with Normalizing Flows. *arXiv e-prints*, page arXiv:1505.05770.

Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page II–1278–II–1286. JMLR.org.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252.

Salakhutdinov, R. and Murray, I. (2008). On the quantitative analysis of deep belief networks. In *Proceedings of the 25th international conference on Machine learning*, pages 872–879. ACM.

Salimans, T., Karpathy, A., Chen, X., and Kingma, D. P. (2017). Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. *CoRR*, abs/1701.05517.

Tabak, E. G. and Turner, C. V. (2013). A family of nonparametric density estimation algorithms. *Communications on Pure and Applied Mathematics*, 66(2):145–164.

Uria, B., Murray, I., and Larochelle, H. (2014). A deep and tractable density estimator. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page I–467–I–475. JMLR.org.

van den Berg, R., Hasenclever, L., Tomczak, J. M., and Welling, M. (2018). Sylvester Normalizing Flows for Variational Inference. *arXiv e-prints*, page arXiv:1803.05649.

van den Oord, A., Kalchbrenner, N., and Kavukcuoglu, K. (2016). Pixel recurrent neural networks. *CoRR*, abs/1601.06759.

van den Oord, A., Li, Y., Babuschkin, I., Simonyan, K., Vinyals, O., Kavukcuoglu, K., van den Driessche, G., Lockhart, E., Cobo, L. C., Stimberg, F., Casagrande, N., Grewe, D., Noury, S., Dieleman, S., Elsen, E., Kalchbrenner, N., Zen, H., Graves, A., King, H., Walters, T., Belov, D., and Hassabis, D. (2017). Parallel wavenet: Fast high-fidelity speech synthesis. *CoRR*, abs/1711.10433.

Wainwright, M. J. and Jordan, M. I. (2008). Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1–2):1–305.