M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Density Estimation with Normalizing Flows

Mathias Niemann Tygesen
(s153583)

Kongens Lyngby 2020

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

In this thesis, we present the theory behind a family of generative models called Normalizing Flows. Normalizing flows work by transforming a simple, known distribution into a much more complex distribution through a series of simple, invertible transformations. Normalizing flows are an attractive family of models as they have efficient sampling and density estimation with exact log-likelihood. By conditioning the transformations in a flow on conditioning variables, we can create conditional normalizing flows that can be used for conditional density estimation. Regularization of conditional generative models is important since they are prone to overfitting. We experiment with different regularization methods and show that noise regularization methods are great at regularizing normalizing flows and have a nice theoretical interpretation.
Through experiments on synthetic data and two different geospatial data sets, we show that conditional normalizing flows can model low dimensional conditional distributions. The autoregressive structure in the affine coupling layers prohibits affine coupling flows from creating disjoint distribution while the transformations in planar flows struggle with modeling complex distributions. By combining both types of layers we show that a mixed flow is able to both create disjoint distributions and complex distributions.
Lastly, we demonstrate how normalizing flows can be used to solve the downstream task of optimizing bike rental locations in Copenhagen and present an alternative method for comparing the models based on the downstream task.

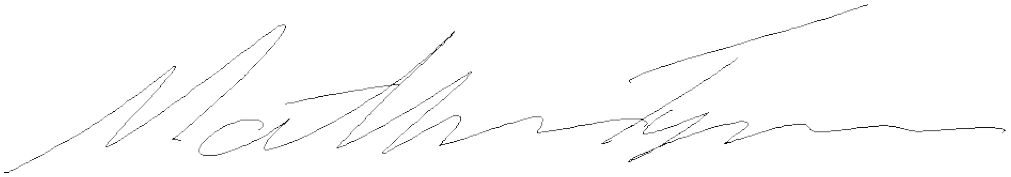
Preface

This 35 ECTS masters thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a masters degree in Mathematical Modelling and Computing.

This thesis assumes no prior knowledge of Normalizing Flows but some familiarity with Machine Learning methods.

I would like to thank Filipe Rodrigues, Sergio Garrido and Daniele Gammelli for excellent guidance during this thesis.

Kongens Lyngby, June 16, 2020



Mathias Niemann Tygesen (s153583)

Contents

CHAPTER 1

Introduction

Density estimation is the problem of approximating the underlying distribution of observed data. This problem is prevalent in a multitude of areas and applications such as synthetic speech, image generation, population analysis and geospatial demand prediction. Often, we wish to construct a generative model that can approximate the underlying distribution. A generative model allows for density estimates of observed data and sampling from the approximated distribution. Both of these operations can be valuable as is but can also be used to do downstream tasks in a larger data science process.

Normalizing flows are a recent family of generative models that have seen a lot of research in recent years. Normalizing flows are an especially interesting approach to generative modelling as they allow for both feasible sampling and density estimation in the same model while attaining exact likelihood. This is something other generative models like GANs and VAEs cannot do. Due to their exact likelihood, normalizing flows can be trained using common machine learning techniques but it also makes them usable within more complex Bayesian methods, that rely on flexible distributions, such as VAEs or Markow Chain Monte Carlo.

One of the first proposed normalizing flows were the planar flow presented in [RM15]. Since then there have been proposed a number of different architectures for normalizing flows, each with their own strengths and weaknesses. As for many other families of generative models, the context of images has seen a lot of research. Therefore many flows, optimized for the high-dimensional space of images, have been proposed. One such flow is the affine coupling flow that stands out by being computationally very efficient and having seen widespread application on images e.g. in [KD18] and [Ho+19]. However, the affine coupling flow has been largely untested in low-dimensional settings. In this thesis, we will apply normalizing flows to the problem of modelling the distribution of geospatial data. Geospatial density estimation problems often exhibit complex and highly multimodal structures. As such they are a great way to test normalizing flows expressiveness in low dimensions. Through experiments with affine coupling flows on geospatial demand data we will investigate the performance of the flows in low-dimensional settings. We compare the strengths and weaknesses of the affine coupling flow to that of the planar flow, which is not optimized for high-dimension problems. Furthermore, we experiment with combining the two types of flows into a single mixed normalizing flow showing how the flows complement each other.

One common problem with generative models is that their flexibility makes them prone to overfitting. Normal regularization methods can be applied to normalizing flows but with limited success. We try out a new proposed noise regularization that has been

shown to work on generative models with the added benefit of a theoretical interpretation of how the regularization affects the modelled distribution. We compare the noise regularization to other common regularization methods through experiments on a synthetic data set.
We do experiments on two different geospatial problems. Through modelling taxis in New York, we show how the expressiveness of the different normalizing flows differ and how they can be combined into a single normalizing flow. Through modelling bike rentals in Copenhagen, we show how a normalizing flow can be used to solve a downstream task based on the geospatial density estimation and how to create an alternative model comparison method based on the downstream task.
In Chapters 2-4 we present the theory behind normalizing flows in general and Affine Coupling and Planar flows in particular. In Chapter 6 we present Noise Regularization and describe how it can be used to regularize generative models. In Chapter 7 we experiment with conditional normalizing flows and different regularization methods on synthetic data. In Chapter 8 we experiment with conditional normalizing flows on a well known geospatial data set. We experiment with different depths and types of normalizing flows to see how their expressiveness differ. In Chapter 9 we apply conditional normalizing flows to our novel problem of geospatial demand prediction of rental bikes in Copenhagen with the downstream task of optimizing placements rental locations around Copenhagen.

Implementations

All the code used in thesis can be found at `https://github.com/MathiasNT/Thesis_Density_Estimation_w_Normalizing_Flows`. A short discussion of the implementation can be found in Appendix A.1.

Notation

We will use a notation based on [Pap+19].

- Vectors are indicated by lowercase bold i.e. $\boldsymbol{x}$.
- Matrices are indicated by uppercase bold i.e. $\boldsymbol{X}$.
- Probabilities are indicated by $\Pr(\cdot)$.
- Probability densities are indicated by $p(\boldsymbol{x})$. Often we add subscript to clarify which random variable the density refers to i.e. $p_x(\boldsymbol{x})$ or in the case of a modelled distribution which parameters gave rise to the distribution, i.e., $p_{\boldsymbol{\phi}}(\boldsymbol{x})$.
- We label the true probability distribution with an asterisk $p^*_x(\boldsymbol{x})$

CHAPTER 2
Density Estimation

This introductory chapter is based on [Bis06], [Rot+19a] and [TT18]

2.1 Density Estimation

One of the core tasks in machine learning is *density estimation*. Density estimation is the unsupervised task of approximating the distribution of observed data within an input space. Given a data set $\mathcal{D} = \{\boldsymbol{x_n}\}_{n=1}^{N}$ we want to approximate the underlying probability distribution that gave rise to the data $p^*(\boldsymbol{x})$.
Note that given a finite data set, any distribution $p(\boldsymbol{x})$, that is non-zero at all the observations, could have given rise to the data. As there are infinitely many possible such probability distributions, the problem of density estimation is by definition ill-posed. Nevertheless, there are many different approaches to do density estimation such as *kernel density estimators*, *nearest-neighbour methods* or fitting a Gaussian using maximum likelihood estimation. The different approaches can be grouped together in two general types of methods: *parametric* and *non-parametric* methods.

Non-parametric methods are quite diverse but common for all of them is that we do not assume that the distribution follows any specific form. We approximate the distribution through other means commonly defining the distribution using the observed data. A common example of a non-parametric model is the Kernel Density Estimator (KDE). A KDE can be seen as a more advanced, smoothed histogram. How the KDE can be derived from the histogram can be seen in [Bis06, p.122-124][1]. KDE works by placing a kernel $K(\cdot)$, with a bandwidth hyperparameter h at each point in a data set. For a new observation, called a query point, $\boldsymbol{x}_q$ the modelled probability $p(\boldsymbol{x}_q)$ is then the sum of all probabilities from the kernels scaled with the data set size and bandwidth

$$p(\boldsymbol{x}_q) = \frac{1}{N}\sum_{t=1}^{N}\frac{1}{V_{Dh}}K\left(\frac{\|\boldsymbol{x}_q - \boldsymbol{x}_t\|}{h}\right), \tag{2.1}$$

where $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N)$ is observed data and V_{Dh} is the volume encompassed by the kernel. A common choice of kernel function is a Gaussian. If we plug a Gaussian into (2.1) we

[1] Alternatively see `https://mglerner.github.io/posts/histograms-and-kernel-density-estimation-kde-2.html?p=28` for a more visual explanation.

get

$$p(\mathbf{x}_q) = \frac{1}{N}\sum_{n=1}^{N} \frac{1}{(2\pi h^2)^{1/2}} \exp\left\{-\frac{\|\mathbf{x}_q - \mathbf{x}_n\|^2}{2h^2}\right\}. \tag{2.2}$$

In parametric density estimation we assume that the distribution comes from a parametric family of distributions $\mathcal{F} = \{p_{\boldsymbol{\theta}}(\boldsymbol{x})|\boldsymbol{\theta} \in \Theta\}$, where Θ is the space of possible parameters. We then wish to choose the parameters $\boldsymbol{\theta}$ such that the approximated distribution $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ comes as close to the true distribution $p^*(\boldsymbol{x})$ as possible. It is important that the chosen family of distributions is expressive enough to model the true density. Otherwise, even for the models optimal parameters $\boldsymbol{\theta}^*$ the difference between the modelled distribution and the true distribution might be significant simply due to limited expressiveness. As normalizing flows can be seen as a parametric model the focus will mainly be on parametric methods.

2.2 Conditional Density Estimation

The task of conditional density estimation (CDE) is closely related to the problem of regression. Given observations of dependent variables[2] $\boldsymbol{y}$ and independent variables[3] $\boldsymbol{x}$, both CDE and regression analysis aim to predict the dependent variables given unseen conditioning variables. That is, given a data set $\mathcal{D} = \{\boldsymbol{x}_n, \boldsymbol{y}_n\}_{n=1}^N$, we fit the model $p(\boldsymbol{y}|\boldsymbol{x})$ such that we can infer information about $\boldsymbol{y}^*$ given some input $\boldsymbol{x}^*$, where $\boldsymbol{x}^*$ is some unseen data. Regression analysis tries to answer this by modelling the mean of the dependent variables, given the conditioning variables, i.e, $\mathbb{E}[\boldsymbol{y}^*|\boldsymbol{x}^*]$, commonly with the assumption that the residuals resemble white noise. This is a decent model if the conditional distribution $p(\boldsymbol{y}^*|\boldsymbol{x}^*)$ is symmetric and uni-modal. However this is not always the case and if, for example, $p(\boldsymbol{y}^*|\boldsymbol{x}^*)$ is bi-modal, the mean will be a pretty bad prediction . Even if $\mathbb{E}[\boldsymbol{y}^*|\boldsymbol{x}^*]$ is an unbiased prediction we can still miss out on a lot of information. For example, the distribution could be skewed, have fat tails or be close to uniform. It is also possible that for a specific task, we do not want to predict a single $\boldsymbol{y}$ value but want to approximate the distribution of $\boldsymbol{y}$.
CDE aims to do exactly that. Instead of modelling the mean, CDE models the full conditional distribution $p(\boldsymbol{y}^*|\boldsymbol{x}^*)$. As in unconditional density estimation, this can be done in a parametric and a non-parametric way, both of which can be seen as extensions of their unconditional cases.

One method to do Non-parametric CDE is to extend the KDE explained earlier. The extension builds on the definition of the conditional distribution:

$$p(\boldsymbol{y}|\boldsymbol{x}) = \frac{p(\boldsymbol{x}, \boldsymbol{y})}{p(\boldsymbol{x})}. \tag{2.3}$$

[2] Also called explained variables
[3] Also called explanatory variables, conditioning variables or covariates.

If we fit two KDE models, one for $\boldsymbol{x}$ and one for $\boldsymbol{y}$, we can use the definition of the conditional distribution to weigh the contribution of each kernel at a new query point, based on how similar the conditioning variables are. This gives us a *Kernel Conditional Density Estimator* (KCDE), specifically the Nadarya-Watson conditional density estimator ([DZ03], [HGI07])

$$p(\boldsymbol{y}|\boldsymbol{x}) = \frac{\sum_i K_{h_1}\left(\|\boldsymbol{y} - \boldsymbol{y}_i\|\right) K_{h_2}\left(\|\boldsymbol{x} - \boldsymbol{x}_i\|\right)}{\sum_i K_{h_2}\left(\|\boldsymbol{x} - \boldsymbol{x}_i\|\right)}, \tag{2.4}$$

where $K_h(\boldsymbol{t}) = \frac{1}{h^d} K(\frac{t}{h})$ where d is the dimension of the kernel.

In parametric conditional density estimation we still propose a family of distributions, $\mathcal{F} = \{p_{\boldsymbol{\theta}}(\boldsymbol{x})|\boldsymbol{\theta} \in \Theta\}$, but now we also propose a family of functions, h, indexed by $\omega \in \Omega$. We then select ω s.t.

$$h_\omega : X \to \Theta,\ \boldsymbol{x} \mapsto \theta. \tag{2.5}$$

θ is then used to model the distribution $p(\boldsymbol{y_i}|\boldsymbol{x_i})$ as $p(\boldsymbol{y_i}|\theta_i = h_\omega(\boldsymbol{x_i}))$. In other words, we introduce a function h_ω that, given the conditioning variables $\boldsymbol{x}$ finds the correct parameters, θ, for our chosen family of distributions to model $p(\boldsymbol{y}|\boldsymbol{x})$.
Hence, the chosen family of distributions $\mathcal{F}$ determines how complex distributions we can model, while the chosen family of functions h determines how complex conditional dependencies we can model. Parametric conditional density estimations methods often use highly expressive families of distributions for $\mathcal{F}$ and highly expressive families of functions for h. Therefore the methods can suffer from overfitting. This is especially a problem when combining the methods with maximum likelihood estimation and noisy and scarce data. As normalizing flows are parametric models, we will return to the problem of regularizing our models later.

CHAPTER 3
Normalizing Flows

This introductory chapter is based on [Pap+19] and [KPB19]

Normalizing flows are a relatively new method for probabilistic and generative modelling. As such it serves a similar purpose as other popular methods like *Generative Adversarial Networks* (GANs) and *Variational Auto Encoders* (VAEs). However, one of the main attractions of normalizing flows compared to VAEs and GANS is that normalizing flows allow for both generative sampling and density estimation while achieving exact likelihoods. This means that given a data point $\boldsymbol{x}$, we can calculate the exact likelihood $p(\boldsymbol{x})$ under the model. Therefore normalizing flows can be trained directly with maximum likelihood estimation (MLE). If we have a fitted normalizing flow the generative sampling can be used to generate samples $\tilde{\boldsymbol{x}}$ that closely follows the distribution of the observed data.
The main idea behind normalizing flows is to transform a simple known distribution, called the base distribution[1], into some complex distribution, called the target distribution, that approximates the distribution of the data. The base distribution is often chosen to be a standard Gaussian but can also be a parameterized distribution. To transform the base distribution they use highly expressive invertible transformations. Since the transformations are invertible we can transform externally observed data back to the base distribution for likelihood estimation or we can send samples from the base distribution through the transformations to get samples from the target distribution.
Normalizing flows can be constructed in different ways. In this chapter, we will go over the general theory of normalizing flows and different methods to fit them to data.

3.1 General structure

There are many different ways to make normalizing flows, but common for them all is that they have the same general structure. Here we will present this structure, describe how it works and explain how this dictates implementations of the different normalizing flow models.

All normalizing flows are built upon the property that if we do a change of variable in a probability distribution, we can change the probabilities along with it. This property is so important that we will set it up as a theorem and motivate a proof for it:

[1] Sometimes the base distribution is called the prior distribution.

Theorem 3.1.1. Let $p_u(\boldsymbol{u})$ be a probability distribution and T be an invertible transformation such that T and T^{-1} are both differentiable. Then if $\boldsymbol{u} = T^{-1}(\boldsymbol{x})$ the density of $\boldsymbol{x}$ is given by

$$p_x(\boldsymbol{x}) = p_u(\boldsymbol{u}) |\det \boldsymbol{J}_T(\boldsymbol{u})|^{-1} \tag{3.1}$$

or equivalently in terms of the inverse transformations

$$p_x(\boldsymbol{x}) = p_u(T^{-1}(\boldsymbol{x})) |\det \boldsymbol{J}_{T^{-1}}(\boldsymbol{x})|, \tag{3.2}$$

where

$$\boldsymbol{J}_T(\boldsymbol{x}) = \frac{\partial \boldsymbol{T}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial T_1}{\partial x_1} & \cdots & \frac{\partial T_1}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial T_D}{\partial x_1} & \cdots & \frac{\partial T_D}{\partial x_D} \end{bmatrix} \tag{3.3}$$

is the Jacobian of the transformation where D is the dimension of $\boldsymbol{x}$.

A complete proof of the theorem requires measure theory and is outside the scope of this thesis. The intuition is that an infinitesimal small neighbourhood $\boldsymbol{du}$ around a point $\boldsymbol{u}$ is mapped to an infinitesimal neighbourhood $\boldsymbol{dx}$ around another point $\boldsymbol{x} = T(\boldsymbol{u})$. The absolute Jacobian determinant $|\det \boldsymbol{J}_T(\boldsymbol{u})|$ quantifies the relative change of volume around the points going from $\boldsymbol{u}$ to $\boldsymbol{x}$ due to T. More specifically, $|\det \boldsymbol{J}_T(\boldsymbol{u})|$ is equal to the volume of $\boldsymbol{dx}$ divided by $\boldsymbol{du}$. Since the probability mass in $\boldsymbol{dx}$ must be equal to the probability mass in $\boldsymbol{du}$ the density at $\boldsymbol{x}$ must be changed proportionally. Hence we have $p_x(\boldsymbol{x})$ must be $p_u(\boldsymbol{u}) |\det \boldsymbol{J}_T(\boldsymbol{u})|^{-1}$.
Clearly, the above theorem can be used in a normalizing flow to transform the simple base distribution into more a complex distribution as long as T is chosen s.t. it satisfies the conditions. Suppose we wish to estimate a joint distribution, $p^*(\boldsymbol{x})$, with $\boldsymbol{x} \in \mathbb{R}^D$. Let $p_u(\boldsymbol{u})$ be the base distribution with $\boldsymbol{u} \in \mathbb{R}^D$. A common choice for the base distribution is a simple Gaussian $\mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$. Let T be an invertible transformation s.t. $\boldsymbol{x} = T(\boldsymbol{u})$ for $\boldsymbol{u} \sim p_u(\boldsymbol{u})$. Then by direct application of Theorem 3.1.1 we can calculate the modelled distribution $p_x(\boldsymbol{x})$ by

$$\begin{aligned} p_x(\boldsymbol{x}) &= p_u(\boldsymbol{u}) \left|\det \boldsymbol{J}_T(\boldsymbol{u})\right|^{-1} \\ &= p_u(\boldsymbol{T^{-1}}(\boldsymbol{x})) |\det \boldsymbol{J}_{T^{-1}}(\boldsymbol{x})| \end{aligned} \tag{3.4}$$

or in log terms

$$\begin{aligned} \log p_x(\boldsymbol{x}) &= \log p_u(\boldsymbol{T}(\boldsymbol{u})) - \log |\det \boldsymbol{J}_T(\boldsymbol{u})| \\ &= \log p_u(\boldsymbol{T}^{-1}(\boldsymbol{x})) + \log |\det \boldsymbol{J}_{T^{-1}}(\boldsymbol{x})|. \end{aligned} \tag{3.5}$$

This distribution over $\boldsymbol{x}$ is defined using only a simple known distribution $p_u(\boldsymbol{u})$ and a transformation T. The question is then how to choose the transformation T s.t. $p_x(\boldsymbol{x})$ actually approximates the true distribution $p^*(\boldsymbol{x})$.

As stated in Theorem 3.1.1 the transformation T has to have certain properties. In order for T to be a candidate for a normalizing flow, it must have

- an inverse T^{-1}
- both T and T^{-1} must be differentiable in order to calculate $|\det \boldsymbol{J}_T(\boldsymbol{x})|$ and $|\det \boldsymbol{J}_{T^{-1}}(\boldsymbol{x})|$

A transformation that adheres to these constraints is called a *diffeomorphism*. If T is a diffeomorphism then the normalizing flow is theoretically possible. However, it might still not be practically feasible. As the calculation of determinants requires $\mathcal{O}(D^3)$ operations, where D is the number of dimensions, this can quickly make training and using the normalizing flow infeasible for high dimensional problems. Hence for practical normalizing flows, the following restriction on T is necessary:

- It must be fast to calculate the log determinant of the transformation, $\log|\det \boldsymbol{J}_T(\boldsymbol{u})|$.

Making sure the log determinant is fast to calculate is a quite restrictive constraint. Luckily diffeomorphisms are composable. This means that if T_1 and T_2 are diffeomorphisms then $T_2 \circ T_1$ is as well. Then due to the algebra of compositions the inverse and Jacobian determinants of T are easily computable given that they are so for T_1 and T_2:

$$(T_2 \circ T_1)^{-1} = T_1^{-1} \circ T_2^{-1} \tag{3.6}$$

$$\det \boldsymbol{J}_{T_2 \circ T_1}(\boldsymbol{u}) = \det \boldsymbol{J}_{T_2}(T_1(\boldsymbol{u})) \cdot \det \boldsymbol{J}_{T_1}(\boldsymbol{u}). \tag{3.7}$$

We can use (3.4) or (3.5) along with (3.6) and (3.7) to create arbitrarily long flows as follows: Let T be a transformation composed of K sub-transformation, i.e. $T = T_K \circ \cdots \circ T_1$ and let $p_u(\boldsymbol{u})$ be a base distribution. Let $\boldsymbol{z}_k$ be the variable a sample $\boldsymbol{u} \sim p_u(\boldsymbol{u})$ takes after the k'th transformation, and denote $\boldsymbol{z}_0 = \boldsymbol{u}$ and $\boldsymbol{z}_K = \boldsymbol{x}$. Then the target distribution can be found by

$$\log p_x(\boldsymbol{x}) = \log p_u(\boldsymbol{u}) - \sum_{k=1}^{K} \log|\det \boldsymbol{J}_{T_k}(\boldsymbol{z}_{k-1})| \tag{3.8}$$

and again we can do it in terms of the inverse transformation

$$\log p_x(\boldsymbol{x}) = \log p_u(\boldsymbol{u}) + \sum_{k=1}^{K-1} \log|\det \boldsymbol{J}_{T_k^{-1}}(\boldsymbol{z}_k)| + \log|\det \boldsymbol{J}_{T_K^{-1}}(\boldsymbol{x})|. \tag{3.9}$$

The series $(\boldsymbol{z}_0, \ldots, \boldsymbol{z}_K)$ obtained from a sample $p_u(\boldsymbol{u})$ is called a *flow*. The transformation from base distribution to target distribution is called the *generative direction* and the inverse is called the *normalizing direction*. Some papers also denote the transformations by these directions: f for the generative direction (T in the above) and g for the normalizing direction (T^{-1} in the above). We will not adopt this notation but assume the generative direction unless otherwise stated. These directions also correspond to the two operations a flow-based model can do

- Sampling from the model: A sample from $p_u(\boldsymbol{u})$ is sent through the generative direction to get a sample from the target distribution $p_x(\boldsymbol{x})$.

- Evaluating the model's density: An observation $\boldsymbol{x} \sim p_x^*(\boldsymbol{x})$ is sent through the normalizing direction to get $\boldsymbol{u} \sim p_u(\boldsymbol{u})$ the density of which can be evaluated.

These two operations are quite different, and different settings might require one or the other. If the setting requires a model with fast sampling, the normalizing flow needs a fast implementation of the generative direction. Conversely, if the setting requires density estimation, the normalizing flow needs a fast implementation of the normalizing direction. In practice, if the model only needs to do one of these operations, the other operation does need to be tractable. However for the theory to still apply, both directions need to be well defined.

3.2 Training of normalizing flows

Depending on which direction of the flow is efficient there are different ways of training normalizing flows. Normalizing flows are parametric models, i.e., the transformations in the flow are parameterised by parameters $\boldsymbol{\theta}$. We will denote the parameters for the flow as $\boldsymbol{\phi}$ and denote any parameters for the base distribution as $\boldsymbol{\psi}$, i.e, $\boldsymbol{\theta} = \{\boldsymbol{\phi}, \boldsymbol{\psi}\}$. As mentioned earlier, a standard Gaussian is often used for the base distribution in which case $\boldsymbol{\theta} = \boldsymbol{\phi}$. Given a distribution $p_x^*(\boldsymbol{x})$ the goal when fitting a normalizing flow is to choose parameters $\boldsymbol{\theta}$ such that the flows target distribution $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ is as close to the true distribution as possible.

Training finite flows with density estimation

Given, that a flow has efficient density estimation a way to fit it to a distribution can be derived by minimizing the KL-divergence between the target distribution and true distribution:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} D_{\mathrm{KL}}\left(p_x^*(\boldsymbol{x}) \| p_{\boldsymbol{\theta}}(\boldsymbol{x})\right) = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}). \tag{3.10}$$

Using the fact that a normalizing flow admits exact likelihood we can rewrite $\mathcal{L}(\theta)$ as follows

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\theta}) &= D_{\mathrm{KL}}(p_x^*(\boldsymbol{x}) \| p_{\boldsymbol{\theta}}(\boldsymbol{x})) && (3.11)\\
&= \mathbb{E}_{p_x^*(\boldsymbol{x})}\left[\log \frac{p_x^*(\boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{x})}\right] && (3.12)\\
&= \mathbb{E}_{p_x^*(\boldsymbol{x})}\left[\log p_x^*(\boldsymbol{x})\right] - \mathbb{E}_{p_x^*(\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x})\right] && (3.13)\\
&= -\mathbb{E}_{p_x^*(\boldsymbol{x})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x})] + c && (3.14)\\
&= -\mathbb{E}_{p_x^*(\boldsymbol{x})}\left[\log p_u(T^{-1}(\boldsymbol{x}; \boldsymbol{\phi}); \boldsymbol{\psi}) + \log \left|\det \boldsymbol{J}_{T^{-1}}(\boldsymbol{x}; \boldsymbol{\phi})\right|\right] + c. && (3.15)
\end{aligned}$$

Given that we have a data set of i.i.d samples from the target distribution, i.e. $\mathcal{D} = \{\boldsymbol{x}_i\}_{i=0}^N$ where $\boldsymbol{x}_n \sim p_x^*(\boldsymbol{x})$, then the optimization in equation 3.10 can be approximated

by minimizing the KL-divergence between the empirical data distribution $p_{\mathcal{D}}(\boldsymbol{x})$ and the model $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ where

$$p_{\mathcal{D}}(\boldsymbol{x}) = \frac{1}{N}\sum_{n=1}^{N}\delta(\|\boldsymbol{x}-\boldsymbol{x}_n\|) \quad \delta(0)=1 \text{ and } \delta(\cdot)=0 \text{ otherwise.} \tag{3.16}$$

This corresponds to doing the optimization in equation 3.10 while approximating the expectation in equation 3.15 with a Monte Carlo estimate, i.e. we approximate the loss with

$$\mathcal{L}(\boldsymbol{\theta}) \approx -\frac{1}{N}\sum_{n=1}^{N}\log p_u(T^{-1}(\boldsymbol{x_n},\boldsymbol{\phi});\boldsymbol{\psi}) + \log|\det \boldsymbol{J}_{T^{-1}}(\boldsymbol{x_n},\boldsymbol{\phi})| + c. \tag{3.17}$$

Note that the above is exactly equal to fitting the model to the data using maximum likelihood estimation. This means that we can choose the optimal parameters for a flow model by optimising the likelihood

$$\theta^* = \arg\min_{\boldsymbol{\theta}} D_{\mathrm{KL}}(p_x^*(\boldsymbol{x})\|p_{\boldsymbol{\theta}}(\boldsymbol{x})) \approx \arg\max_{\boldsymbol{\theta}}\sum_{n=1}^{N}\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_n). \tag{3.18}$$

So in order to do the maximum likelihood estimation we need the flow to be able to efficiently transform an external sample in the normalizing direction, i.e. use T^{-1} and evaluate the density at the base distribution $p_u(\boldsymbol{u};\boldsymbol{\psi})$. As this is the most direct way to fit a normalizing flow, many flows are optimized for this. This means that the flows are created such that the transformation in the normalizing direction T^{-1} is fast, with fast calculation of the log determinant and density estimation at the base distribution. In fact it is possible to fit a normalizing flow using the above method if just these properties are tractable. However, if the model has to be able to generate samples after training, then the transformation in the generative direction and the corresponding log determinant also has to be tractable.

Training flows with sampling

If we have an external method to give likelihoods to generated samples, we can also use the flows ability to generate samples to fit the flow. A method to do this can be derived by flipping the distributions in the KL-divergence:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= D_{\mathrm{KL}}\left[p_{\boldsymbol{\theta}}(\boldsymbol{x})\|p_{\boldsymbol{x}}^*(\boldsymbol{x})\right] \\ &= \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) - \log p_{\boldsymbol{x}}^*(\boldsymbol{x})\right] \\ &= \mathbb{E}_{p_{\boldsymbol{u}}(\boldsymbol{u};\boldsymbol{\psi})}\left[\log p_{\boldsymbol{u}}(\boldsymbol{u};\boldsymbol{\psi}) - \log|\det \boldsymbol{J}_T(\boldsymbol{u};\boldsymbol{\phi})| - \log p_{\boldsymbol{x}}^*(T(\boldsymbol{u};\boldsymbol{\phi}))\right].\end{aligned} \tag{3.19}$$

Note we do a change of variable so that the expectation is over the base distribution $p_u(\boldsymbol{u};\boldsymbol{\psi})$. Furthermore notice that we have to be able to evaluate the density $p_x^*(T(\boldsymbol{u};\boldsymbol{\phi}))$ which means we have to be able to evaluate the true density but not to sample from it. The evaluation of the true density does need to be exact as optimization on $\mathcal{L}(\boldsymbol{\theta})$ above

is the same even if we do not know $p_x^*(\boldsymbol{x})$ exactly but within a multiplicative normalizing constant, i.e. we can replace $p_x^*(\boldsymbol{x})$ with $\tilde{p}_x(\boldsymbol{x})$ if $p_x^* = \tilde{p}_x(\boldsymbol{x})/C$ where $C = \int \tilde{p}_x(\boldsymbol{x})d\boldsymbol{x}$. Since the expectation is over the base distribution, we can make as many samples as we want and do a Monte Carlo estimate of the expectation. And after that, we can optimize using SGD methods as before. Note that this method has the opposite requirements from before, and the flow now does not need the inverse operator T^{-1}. This formulation is perfect for adding normalizing flows into variational inference (VI) or doing *model distillation*. *Model distillation* is where one has a trained model with good density evaluation but slow sampling, and one wishes to "replicate" the model with a normalizing flow. This is in essence how the Parallel WaveNet ([Oor+17]) in Google Assistant is trained. An autoregressive model with great training performance is fitted and then distilled into a normalizing flow model. This allows the model to synthesize speech in real-time, allowing you to have a conversation with your phone.

CHAPTER 4
Finite Normalizing Flows

As mentioned in Chapter 3 a way to construct Normalizing Flows is to exploit the fact that diffeomorphisms are composable. The idea is to chain a sequence of simple transformations to construct a much more complex transformation

$$T = T_K \circ \cdots \circ T_1 \tag{4.1}$$

and then use (3.8) and (3.9) to calculate the probabilities of samples and external observations. This requires that the transformations are diffeomorphisms, which we will show two different methods of ensuring. However in order for the flow to be usable in practice and not just in theory, the flows operations have to be feasible. Based on which operation is required we get different constraints:

- If the flow needs to be able to do sampling we need:
  - The transformations to be fast in the generative direction i.e. T
  - It to be fast to calculate $|\det \boldsymbol{J}_T(\cdot)|$
- If the flow needs to be able to do density estimation we need:
  - The transformations must be fast in the normalizing direction i.e. T^{-1}
  - It must be fast to calculate $|\det \boldsymbol{J}_{T^{-1}}(\cdot)|$

Meeting these conditions on the singular transformation level, i.e. for each T_i, also means that the flow meets the conditions by (4.1) and (3.6). There are different ways to tackle these constraints which leads to different kinds transformations. In this chapter we will only present two of them. For a more exhaustive summary see [Pap+19] and [KPB19].

Common for all of the different variations of normalizing flows is the focus on making the calculations of the determinant of the Jacobian as fast as possible. For an arbitrary transformation $T : \mathbb{R}^{\mathbb{D}} \to \mathbb{R}^{\mathbb{D}}$ the worst-case scenario is that the Jacobian is a matrix where all fields are non-zero. In that case, the computational cost of calculating the determinant is $\mathcal{O}(D^3)$, which might be too expensive for a machine learning model that has to go through multiple epochs of a big data set.

The presented normalizing flows fall into two different categories. The first category is autoregressive flows which use an autoregressive structure with simple transformations to ensure invertibility and fast Jacobian determinants. The other category is residual

flows that constrain the transformation to theoretically ensure that an inverse exists. As we shall show each approach has its strengths and weaknesses.

In this chapter, we focus on a single transformation, T, inside of an entire flow. We call such a transformation a *layer*. These layers can be implemented in a normalizing flow with the forward transformation as the generative direction or be reversed depending on the required flow properties. We denote the input to this layer $\boldsymbol{x} \in \mathbb{R}^D$ and the output of the layer $\boldsymbol{x'} \in \mathbb{R}^D$.

4.1 Autoregressive Flows

One way to tackle the inversion and log determinant problem is to make the transformation itself really simple and easy to invert. However, simplifying the transformation comes at the cost of reducing the expressiveness of the flow. Hence very simple transformations might lose the ability to capture non-linear dependencies in the data. Flows based on autoregressive layers use a bit of a trick to hide non-linear dependencies in the transformation.

The idea is: for an input $\boldsymbol{z}$, we split the dimensions, generate parameters based on some of the dimensions and use those parameters in the transformation of the other dimensions. The generation of these parameters can be non-linear as the transformation itself does not become more complex, and the inversion of the transformation is invariant to how the parameters were found. The dimensions can be split with varying granularity. In the simplest case the dimensions are split in half:

$$\begin{aligned} T : \mathbb{R}^{\mathbb{D}} \to \mathbb{R}^{\mathbb{D}}, \quad \boldsymbol{x} \mapsto \boldsymbol{x'} \\ \text{s.t.} \\ \boldsymbol{x'}_{<d} = \boldsymbol{x}_{<d} \\ (\boldsymbol{h}_{d+1}, \ldots, \boldsymbol{h}_D) = c(\boldsymbol{x}_{<d}) \\ x'_i = \tau(x_i; \boldsymbol{h}_i) \quad i > d, \end{aligned} \tag{4.2}$$

where $d = \lfloor D/2 \rfloor$. τ is called the *transformer* and c is called the *conditioner*. This layer is called a *Coupling Layer* and was introduced in [DSB16]. Note that any dimension in $\boldsymbol{x'}$ is either just the same as the corresponding dimension in $\boldsymbol{x}$ for $i < d$ or only conditioned on earlier dimensions for $i > d$. If this structure is preserved and the dimensions are split one by one we have the fully autoregressive case:

$$\begin{aligned} T : \mathbb{R}^{\mathbb{D}} \to \mathbb{R}^{\mathbb{D}}, \quad \boldsymbol{x} \mapsto \boldsymbol{x'} \\ \text{s.t.} \\ x'_i = \tau(x_i; \boldsymbol{h}_i) \quad \boldsymbol{h}_i = c_i(\boldsymbol{x}_{<i}). \end{aligned} \tag{4.3}$$

Note the need for multiple conditioners c_i. This is because for each dimension the conditioner needs to take into account a different amount of other dimensions, i.e., c_i needs

to condition on $i-1$ dimensions.

Autoregressive layers can be constructed using different transformers and conditioners as long as the transformer is invertible. No matter the choice of conditioner and transformer the inverse transformation is given by

$$\begin{aligned} \boldsymbol{x}_{<d} &= \boldsymbol{x'}_{<d} \\ (\boldsymbol{h}_{d+1}, \ldots, \boldsymbol{h}_D) &= c(\boldsymbol{x}_{<d}) \\ x_i &= \tau^{-1}(x'_i; \boldsymbol{h}_i) \quad i > d \end{aligned} \tag{4.4}$$

for the coupling layers and

$$x_i = \tau^{-1}(x'_i; \boldsymbol{h}_i) \quad \boldsymbol{h}_i = c_i(\boldsymbol{x}_{<i}) \tag{4.5}$$

in the fully autoregressive case. Note that neither requires the inverse of the conditioner. Hence the conditioner can be an arbitrarily complex function as the invertablitity of the layer is unaffected. However in Equation (4.5) $\boldsymbol{x}_{<i}$ is needed in order to find x_i. This means that $\boldsymbol{x'}$ has to be inverted iteratively one dimension at a time. Hence the inversion in the fully autoregressive case have a cost of $\mathcal{O}(D)$ passes through the conditioner. In contrast the affine coupling layer only needs a single pass through the conditioner in order to do the inverse transformation. As such the affine coupling layers have one-pass forward and inverse transformations.

This solves the first constraint on T, if τ is invertible so is T. Next we need to show that $|\det \boldsymbol{J}_T(\cdot)|$ is fast to calculate. Since x'_i is only dependent on x_k for $k < i$ the Jacobian is a lower triangular matrix:

$$J_T(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial \tau}{\partial x_1}(x_1; \boldsymbol{h}_1) & & 0 \\ & \ddots & \\ L(\boldsymbol{x}) & & \frac{\partial \tau}{\partial x_D}(x_D; \boldsymbol{h}_D) \end{bmatrix}. \tag{4.6}$$

where $L(\boldsymbol{x})$ the lower triangular part of the matrix beneath the diagional. For the coupling layer the Jacobian simplifies even more to

$$J_T(\boldsymbol{x}) = \begin{bmatrix} I & 0 \\ L(\boldsymbol{x}) & D(\boldsymbol{x}) \end{bmatrix}, \tag{4.7}$$

where I is a d by d identity matrix, $L(\boldsymbol{x})$ is some the lower triangular part of the matrix beneath the diagonal and $D(\boldsymbol{x})$ is a diagonal matrix with $\frac{\partial \tau}{\partial x_i}(x_i, \boldsymbol{h}_i)$ along the diagonal. Since both of the Jabobians are lower triangular matrices $L(\boldsymbol{x})$ can be ignored as the determinant of a triangular matrix is just the product of the diagonal. This means that the log determinant of the Jacobian becomes a sum of the diagonal for the autoregressive case

$$\log|\det J_T(\boldsymbol{x})| = \log|\prod_{i=1}^{D} \frac{\partial \tau}{\partial x_i}(x_i; \boldsymbol{h}_i)| = \sum_{i=1}^{D} \log|\frac{\partial \tau}{\partial x_i}(x_i; \boldsymbol{h}_i)| \tag{4.8}$$

or half diagonal in the coupling layer case

$$\log|\det J_T(\boldsymbol{x})| = \log|\prod_{i=d}^{D}\frac{\partial\tau}{\partial x_i}(x_i;\boldsymbol{h}_i)| = \sum_{i=d}^{D}\log|\frac{\partial\tau}{\partial x_i}(x_i;\boldsymbol{h}_i)|. \tag{4.9}$$

Hence for both the cost of calculating the determinant of the Jacobian is $\mathcal{O}(D)$ which is much better than $\mathcal{O}(D^3)$.

Depending on the choice of transformer, τ and conditioner c we get very different flows. The simplest transformer is an affine transformation that only scales and translates the dimensions:

$$\tau(x_i;\boldsymbol{h}_i) = \alpha_i x_i + \beta_i \quad \boldsymbol{h}_i = \{\alpha_i, \beta_i\}. \tag{4.10}$$

Here $\boldsymbol{h}_i$ is the output from a conditioner, so even though the transformation is only affine, $\boldsymbol{h}_i$ can be some output from a non-linear function of the other dimensions, enabling the layer to capture non-linearities. However even though the layer can express more than linear dependencies it is still limited in its expressiveness and hence longer flows might be needed to get flows expressive enough to model the data. The affine transformation is invertible if and only if $\alpha_i \neq 0$. α_i can be restricted to be non-zero by letting $\bar{\alpha}_i$ be the output from the conditioner and transforming it by

$$\alpha_i = \exp(\tilde{\alpha}_i). \tag{4.11}$$

When $\alpha_i \neq 0$ then the inverse is given by

$$\tau^{-1}(x_i;\boldsymbol{h}_i) = \frac{x_i - \beta_i}{\alpha_i}, \quad \boldsymbol{h_i} = \{\alpha_i, \beta_i\} \tag{4.12}$$

and the derivative of the transformer then becomes

$$\left|\frac{\partial\tau}{\partial x_i}(x_i;h_i)\right| = \left|\frac{\partial}{\partial x_i}\alpha_i x_i + \beta_i\right| = |\alpha_i|. \tag{4.13}$$

Then by (4.8) and (4.11) the log determinant of the Jacobian becomes

$$\log\left|\det J_{f_\phi}(z)\right| = \sum_{i=1}^{D}\log|\alpha_i| = \sum_{i=1}^{D}\tilde{\alpha}_i. \tag{4.14}$$

Hence for the affine transformer the log determinant of the Jacobian can be directly calculated based on the output of the conditioner.

As mentioned earlier, there are no restrictions on the conditioner. Therefore we can choose any function that is expressive enough to capture any non-linear dependencies in the data. A natural choice for such a non-linear function is an *Artificial Neural Network* of some kind. However, depending on how finely the dimensions are split, different structures might be needed. In the coupling layer case, where the dimensions are split in two, a simple *Feed Forward Neural Network* FFNN can be used. The FFNN takes half of

the dimensions, $\boldsymbol{x}_{<d}$, and outputs $\boldsymbol{h}_i$ in the shape that is needed for the transformer. In the fully autoregressive case x'_i needs to be calculated based on $\boldsymbol{x}_{<i}$ for different sizes of $\boldsymbol{x}_{<i}$. Using FFNNs for this would require a different FFNN for each possible size of $\boldsymbol{x}_{<i}$. This can quickly become very expensive in memory, and the flow would end up with a huge number of parameters, which can make training of the flow infeasible. Therefore a structure that can use the same net for all sizes of $\boldsymbol{x}_{<i}$ is needed. One possibility is to use *Recurrent Neural Network* structures. Here the dimensions are fed as a sequence of values $(x_1, x_2, \ldots, x_D)$. The hidden state after each step $(\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots, \boldsymbol{h}_D)$ can then be used by the transformer. Another common approach is to use a *Masked Feed Forward Neural Network*. This is a FFNN masked such that the dependencies on the output follow an autoregressive structure on the input. Hence this can be seen as a network that can calculate all of the "hidden states" in one pass of the network. Using the masked feed forward network allows a fully autoregressive layer to do a full forward transformation with a single pass through the conditioner. However, as mentioned earlier, the inverse transformation still needs D passes through the conditioner. In Figure 4.1 we can see the layout of a affine coupling layer with an FFNN conditioner in the case of $D = 4$ and $d = 2$. For more discussion on the choice of conditioner in the fully autoregressive case, see [PPM17] and [Kin+16].

No matter the granularity of the split chosen, there is still a problem with the autoregressive approach: A dimension, x'_i, can only be conditioned upon x_k if $k < i$. Hence any information in dimension x_j where $j > i$, cannot influence the transformation of x_i to x'_i. This can be remedied by interlacing the transformation layers in a normalizing flow with dimension permutations layers. This way, the dimensions x_i can influence changes from layer to layer in the flow. Permutations are always easily invertible and volume-preserving. Hence they do not influence the invertibility of the flow, and the log determinant of the Jacobian is always 1. As such, permutations can be added into a flow without any problems.

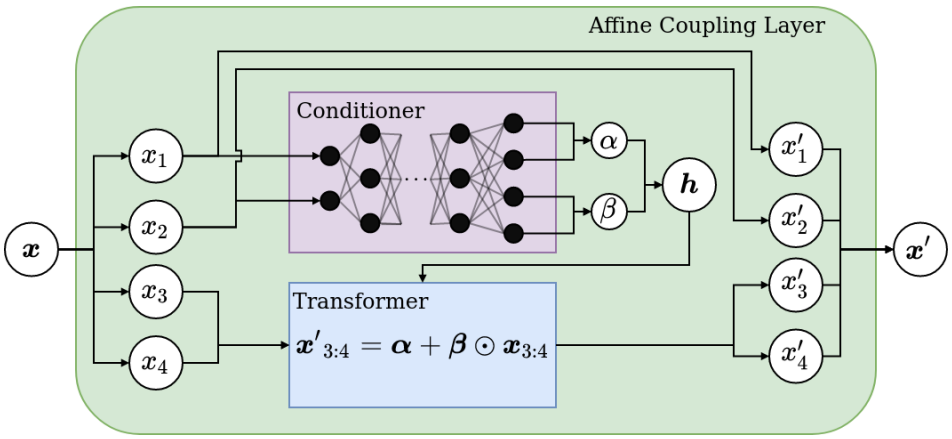


Figure 4.1: Diagram of the layout of an affine coupling transformation for an input dimension of 4.

So the affine coupling layers have one-pass forward and inverse transformations with $\mathcal{O}(D)$ calculation of the log absolute determinant. Fully autoregressive layers also have one-pass forward and $\mathcal{O}(D)$ calculation of the log absolute determinant, but they require D passes through the conditioner to do an inverse transformation. For both types of transformations, we have to stack multiple layers to make sure the normalizing flow is expressive. Each layer only changes the complexity of a the flow with a multiplicative factor i.e. for a flow with K coupling or autoregressive layers the cost of calculating $|\det \boldsymbol{J}_T(\cdot)|$ is $\mathcal{O}(KD)$ by (4.9) and the fact that $|\det \boldsymbol{J}_{T_k}(\cdot)|$ for $0 < k < K$. So depending on the dimensionality of the problem, this scales much better than $\mathcal{O}(D^3)$.

4.2 Residual Flows

Another type of finite flow is residual flows. Residual flows use a different approach to adhering to the constraints than autoregressive flows. Here we will first lay out the theoretical principle behind residual flows before presenting a specific layer, namely the planar layer, which is one of the first layers originally from [RM15].

Residual flows have the general form

$$\boldsymbol{x'} = f_{\boldsymbol{\phi}}(\boldsymbol{x}) = \boldsymbol{x} + g_{\boldsymbol{\phi}}(\boldsymbol{x}), \tag{4.15}$$

where $g_{\boldsymbol{\phi}}$ is some neural network that takes the input of the flow layer and generates a translation of the input variables. This has the clear advantage over the autoregressive flows that all inputs can affect all outputs in each layer of the flow. As before, the transformation in Equation (4.15) has to be a diffeomorphism, i.e., the transformation is invertible, and the determinant of the Jacobian is feasible to compute. We will start by showing that a residual flow is, in fact, reversible, given that $g_{\boldsymbol{\phi}}$ is a *contractive* mapping. A contractive map $F : \mathbb{R}^D \to \mathbb{R}^D$ is a map which is Lipschitz continuous with a constant smaller than one, i.e., there exists a constant $L < 1$ such that for any two inputs $\boldsymbol{z}_a$ and $\boldsymbol{z}_b$

$$\delta(F(\boldsymbol{z}_a), F(\boldsymbol{z}_b)) \leq L\delta(\boldsymbol{z}_a, \boldsymbol{z}_b), \tag{4.16}$$

where δ is a distance function. In more simple terms, this means that any two inputs are brought closer together by F. Since $\mathbb{R}^D$ is a complete metric space, the *Banach fixed-point theorem* ([Kre78, pp. 299]) states that any contractive map has exactly one fixed point $\boldsymbol{z}^*$ s.t. $F(\boldsymbol{z}^*) = \boldsymbol{z}^*$. The theorem also states that this point can be found by taking an arbitrary element in $\mathbb{R}^D$ and defining the sequence $\{\boldsymbol{z}_n\}_{n=0}^{\infty}$ by $\boldsymbol{z}_n = F(\boldsymbol{z}_{n-1})$ for $n \geq 1$. Then $\boldsymbol{z}_n \to \boldsymbol{z}^*$ as $n \to \infty$.
If $g_{\boldsymbol{\phi}}$ is a contractive map then the map $F(\boldsymbol{x}) = \boldsymbol{x'} - g_{\boldsymbol{\phi}}(\boldsymbol{x})$, where $\boldsymbol{x'}$ is given, is also a contractive map with the same Lipschitz constant[1]. Then by the Banach fixed-point

[1]This is easily seen for distances induced by norms, which is what we will use for the planar flow:

$$\begin{aligned}\delta(F(\boldsymbol{z}_a), F(\boldsymbol{z}_b)) &= \delta(\boldsymbol{z'} - g_{\boldsymbol{\phi}}(\boldsymbol{z}_a), \boldsymbol{z'} - g_{\boldsymbol{\phi}}(\boldsymbol{z}_b)) = \|(\boldsymbol{z'} - g_{\boldsymbol{\phi}}(\boldsymbol{z}_a)) - (\boldsymbol{z'} - g_{\boldsymbol{\phi}}(\boldsymbol{z}_b))\| \\ &= \|g_{\boldsymbol{\phi}}(\boldsymbol{z}_a) - g_{\boldsymbol{\phi}}(\boldsymbol{z}_b)\| = \delta(g_{\boldsymbol{\phi}}(\boldsymbol{z}_a), g_{\boldsymbol{\phi}}(\boldsymbol{z}_b)) \leq L\delta(\boldsymbol{z}_a, \boldsymbol{z}_b).\end{aligned} \tag{4.17}$$

theorem, there exists a fixed point $\boldsymbol{x}^*$ for each F with given $\boldsymbol{x'}$ s.t.

$$\begin{aligned} \boldsymbol{x}^* &= \boldsymbol{x'} - g_\phi(\boldsymbol{x}^*) \\ \Leftrightarrow \boldsymbol{x'} &= \boldsymbol{x}^* + g_\phi(\boldsymbol{x}^*) \\ \Leftrightarrow \boldsymbol{x'} &= f_\phi(\boldsymbol{x}^*). \end{aligned} \tag{4.18}$$

where f_ϕ is the transformation in Equation (4.15). Since $\boldsymbol{x}^*$ is unique by the Banach fixed-point theorem, the above shows that given a $\boldsymbol{x'}$, we can find a unique $\boldsymbol{x}^*$ such that f_ϕ maps $\boldsymbol{x}^*$ to $\boldsymbol{x'}$, i.e., the residual transformation f_ϕ is invertible. Note that there is no analytical inverse, but the above gives an iteration procedure that gets closer and closer to to the true inverse for each iteration. Start with $\boldsymbol{x'}$ as $\boldsymbol{z}_0$ and define the sequence $(\boldsymbol{z}_0, \boldsymbol{z_1}, \ldots)$ by iteratively applying F, i.e.

$$\boldsymbol{z}_{k+1} = \boldsymbol{x'} - g_\phi(\boldsymbol{z}_k) \text{ for } k \geq 0. \tag{4.19}$$

Then by the above the sequence will converge to $\boldsymbol{x}^* = f_\phi^{-1}(\boldsymbol{x'})$.[2] The rate of convergence can be bounded as follows. First use the triangle inequality to bound the distance between points $\boldsymbol{z}_m$ and $\boldsymbol{z}_k$ after m and k iterations with $m > k$ using the fact that F is contractive:

$$\begin{aligned} \delta(\boldsymbol{z}_m, \boldsymbol{z}_k) &\leq \delta(\boldsymbol{z}_k, \boldsymbol{z}_{k-1}) + \delta(\boldsymbol{z}_{k-1}, \boldsymbol{z}_{k-2}) + \cdots + \delta(\boldsymbol{z}_1, \boldsymbol{z}_0) \\ &\leq L^{m-1}\delta(\boldsymbol{z}_1, \boldsymbol{z}_0) + L^{m-2}\delta(\boldsymbol{z}_1, \boldsymbol{z}_0) + \cdots + L^k\delta(\boldsymbol{z}_1, \boldsymbol{z}_0) \\ &= L^k\delta(\boldsymbol{z}_1, \boldsymbol{z}_0) \sum_{k=0}^{m-k-1} L^n. \end{aligned} \tag{4.20}$$

Then let m go to ∞ and in the limit

$$\begin{aligned} \delta(\boldsymbol{z}^*, \boldsymbol{z}_k) &\leq L^k\delta(\boldsymbol{z}_1, \boldsymbol{z}_0) \sum_{n=0}^{\infty} L^n \\ &= \frac{L^k}{1-L}\delta(\boldsymbol{z}_1, \boldsymbol{z}_0) \end{aligned} \tag{4.21}$$

since $\lim_{m\to\infty} \boldsymbol{z}_m = \boldsymbol{z}^*$. So for smaller L, the convergence of the inversion procedure gets faster, but the flow will also be less expressive because the transformation has to be more contractive.
It is clear that if the maps $F_1, F_2, \ldots, F_k$ have Lipscitz constants $L_1, L_2, \ldots, L_k$ then the composition of these maps will have Lipschitz constant $\prod_{i=1}^k L_i$, e.g. in the case of $k = 2$ with $F = F_2 \circ F_1$ we have

$$\delta(F(\boldsymbol{z_a}), F(\boldsymbol{z}_b)) = \delta(F_2(F_1(\boldsymbol{z}_a)), F_2(F_1(\boldsymbol{z}_b))) \leq L_2\delta(F_1(\boldsymbol{z}_a), F_1(\boldsymbol{z}_b)) \leq L_2L_1\delta(\boldsymbol{z}_a, \boldsymbol{z}_b). \tag{4.22}$$

[2] In fact the sequence will converge to $\boldsymbol{z}^*$ for any starting point $\boldsymbol{z}_0$.

This is very helpful when g_ϕ is an artificial neural network, as this means that if all the layers in the net have $L \leq 1$ and at least one layer has $L < 1$ then the entire network will be contractive. Many common activation functions like the logistic sigmoid, tanh, and ReLU are, in fact, Lipschitz continuous with $L \leq 1$, but making a full linear layer contractive requires a bit more work. In [Beh+18] they make general linear layers contractive with respect to the Euclidean norm using spectral normalization. Here, we will only do this for the single-layer setup used in the planar layer, which we will return to shortly. With this, we have shown that the transformations in residual flows are contractive if g_ϕ is constrained properly.

Next, we turn to how we can compute the determinant of the Jacobian. No general procedure for computing the determinant of the Jacobian is known, but for flows constructed in the correct way, the *matrix determinant lemma* can be used. The matrix determinant lemma states that

$$\det(\boldsymbol{A} + \boldsymbol{V}\boldsymbol{W}^\top) = \det(\boldsymbol{I} + \boldsymbol{W}^\top \boldsymbol{A}^{-1}\boldsymbol{V}) \det \boldsymbol{A}, \tag{4.23}$$

where $\boldsymbol{A} \in \mathbb{R}^{DxD}$ is an invertible matrix that is tractable to invert and $\boldsymbol{V}, \boldsymbol{W} \in \mathbb{R}^{D\times M}$. When $M < D$ the matrix determinant lemma gives a faster way to compute the determinant of $\boldsymbol{A} + \boldsymbol{V}\boldsymbol{W}^\top$. When $\boldsymbol{A}$ is a diagonal matrix the left hand side of Equation (4.23) costs $\mathcal{O}(D^3+D^2M)$ operations to compute while the right hand side costs $\mathcal{O}(M^3+DM^2)$. So the influence of the dimensions of the matrices is switched, and the right hand side is clearly favourable when $M < D$.

With this we are ready to define the planar layer as presented in [RM15]. The transformation is defined as

$$\boldsymbol{x'} = f_\phi(\boldsymbol{x}) = \boldsymbol{x} + \boldsymbol{v}\sigma(\boldsymbol{w}^\top \boldsymbol{x} + b). \tag{4.24}$$

The parameters of the transformation are $\boldsymbol{\phi} = \{\boldsymbol{w}, \boldsymbol{v}, b\}$ with $\boldsymbol{w}, \boldsymbol{v} \in \mathcal{R}^D$, $b \in \mathcal{R}$ and σ is a differentiable activation function. We will use the hyperbolic tangent function for σ as in [RM15]. The planar flow is a residual flow where $g_\phi = \boldsymbol{v}(\sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$ is a single layer artificial neural network with a single hidden unit that scales a vector v. As discussed above the hyperbolic tangent is contractive, but (4.24) has a full layer, so $\boldsymbol{v}$ and $\boldsymbol{w}$ has to be constrained in order to ensure that g_ϕ is contractive. In Equation (4.24) $\boldsymbol{x}$ is split into $\boldsymbol{x}_\parallel$ parallel to $\boldsymbol{w}$ and $\boldsymbol{x}_\perp$ perpendicular to $\boldsymbol{w}$ s.t. $\boldsymbol{x} = \boldsymbol{x}_\parallel + \boldsymbol{x}_\perp$ to get

$$\boldsymbol{x'} = f_\phi(\boldsymbol{x}) = \boldsymbol{x}_\perp + \boldsymbol{x}_\parallel + \boldsymbol{v}\sigma(\boldsymbol{w}^\top \boldsymbol{x}_\parallel + b). \tag{4.25}$$

We isolate $\boldsymbol{x}_\perp$ to get an equation that can be solved given $\boldsymbol{x}_\parallel$ and $\boldsymbol{x'}$

$$\boldsymbol{x}_\perp = \boldsymbol{x'} - \boldsymbol{x}_\parallel - \boldsymbol{v}\sigma(\boldsymbol{w}^\top \boldsymbol{x}_\parallel + b). \tag{4.26}$$

This equation can be solved uniquely. Hence if we can determine $\boldsymbol{x}_\parallel$ for $\boldsymbol{x'}$, then $\boldsymbol{x}_\perp$ is also determined uniquely. Therefore we can take the dot product of Equation (4.25) with $\boldsymbol{w}$, eliminating $\boldsymbol{x}_\perp$, without loss of generality. At the same time we expand the parallel exponent as $\boldsymbol{x}_\parallel = \alpha\frac{\boldsymbol{w}}{\|\boldsymbol{w}\|^2}$. The resulting equation is

$$\boldsymbol{w}^\top f(\boldsymbol{x}) = \alpha + \boldsymbol{w}^T\boldsymbol{v}\sigma(\alpha + b). \tag{4.27}$$

This equation is invertible with relation to α (which in turn gives $\boldsymbol{x}_{\|}$ and then $\boldsymbol{x}$) if its right hand side is a non decreasing function. This corresponds to

$$1 + \boldsymbol{w}^\top \boldsymbol{v} \sigma'(\alpha + b) \geq 0 \equiv \boldsymbol{w}^\top \boldsymbol{v} \geq -\frac{1}{\sigma'(\alpha + b)}. \tag{4.28}$$

As σ is the hyperbolic tangent the derivative can be bounded by $0 \leq \sigma'(\alpha + b) \leq 1$. So a sufficient condition is $\boldsymbol{w}^\top \boldsymbol{v} \geq -1$. This can be enforced by modifying the component of $\boldsymbol{v}$ parallel to $\boldsymbol{w}$ to create $\hat{\boldsymbol{v}}$ s.t. $\boldsymbol{w}^\top \hat{\boldsymbol{v}} > -1$. This can be done by:

$$\hat{\boldsymbol{v}} = \boldsymbol{v} + \left[m(\boldsymbol{w}^\top \boldsymbol{v}) - \boldsymbol{w}^\top \boldsymbol{v} \right] \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|^2}, \tag{4.29}$$

where $m(x) = -1 + \log(1 + e^x)$. Doing this ensures the planar layers invertibility. Next is the calculation of the Jacobian determinant. The Jacobian of the transformation is given by

$$\boldsymbol{J}_{f_\phi}(\boldsymbol{x}) = \boldsymbol{I} + \sigma'(\boldsymbol{w}^\top \boldsymbol{x} + b) \boldsymbol{v} \boldsymbol{w}^\top. \tag{4.30}$$

The determinant of the Jacobian can be calculated using the matrix determinant lemma with $M = 1$ as

$$\det \boldsymbol{J}_{f_\phi} = 1 + \sigma'(\boldsymbol{w}^\top \boldsymbol{x} + b) \boldsymbol{w}^\top \boldsymbol{v}. \tag{4.31}$$

The computational cost of computing the determinant is $\mathcal{O}(D)$.

As with other flows, we can stack multiple layers on top of each other. The architecture of the planar flow can be seen in Figure 4.2. Unlike the affine coupling layer described earlier, the planar layer allows all input variables to affect all output variables in each transformation. The transformation can be seen as an expansion or contraction perpendicular to the hyperplane $\boldsymbol{w}^\top \boldsymbol{z} + b = 0$. In the affine coupling flow, the transformation of half the dimensions was conditioned upon the other half of the dimensions

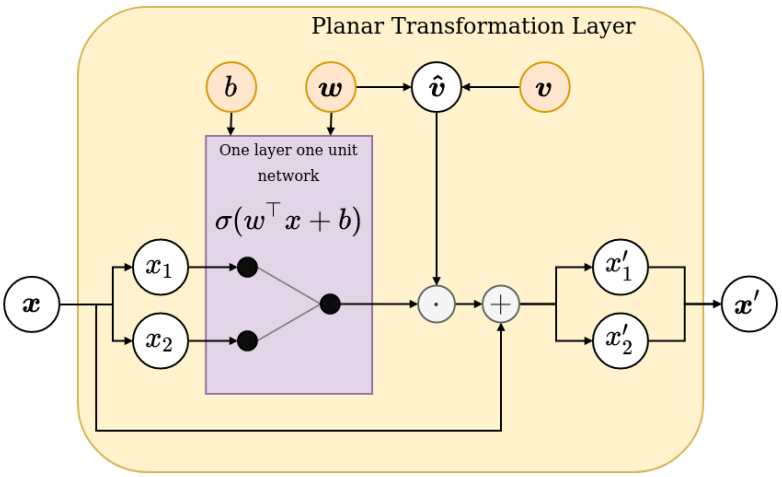


Figure 4.2: Diagram of the Planar transformation in the case of $D = 2$. The variables in yellow circles are the learnable parameters.

by the conditioner. This conditioner was unconstrained and could be a deep artificial neural network. As such, any complex non-linearities between the two splits of $\boldsymbol{x}$ could be modeled. In the planar layer, the non-linearities within $\boldsymbol{x}$ are modeled with what corresponds to a single layer perceptron. Therefore, even though the planar layer affects both dimensions, the complexity of the non-linearities a single layer can capture is lower. Furthermore, the architecture of the planar layer comes with the cost of having no well-defined inverse. Note, however, as discussed in Section 3.2 that we can fit flows as long as one of the directions is well defined, and we can flip the direction of the planar layer to match our use case.

4.3 Batchnormalization

One problem with training models that have neural networks is that there is no closed-form solution for the best network. Hence to train such models, we have to use optimization with some variant of gradient descent. The underlying optimization landscape is very high dimensional. Depending on the data and model at hand, the landscape might also be very irregular and unsmooth with peaks and flat surfaces. This can make training models slow as we have to resort to lower learning rates to correctly descend the irregular landscape. In [IS15] the authors propose *batchnormalization* to speed up training. In the original paper, they propose batchnormalization as a way to combat what they call *internal covariate shift*. The problem is that the input to an internal layer in the neural network is the output of another layer. As such, we can have shifts in the input to internal layers as the model's parameters update during training. This makes it much harder for the network to learn as updates one layer makes might be mitigated by updates subsequent layers in the model take. However, in [San+18], they propose that batchnormalization does not change the internal covariance shift. In fact, they show that internal covariance shift does not impact model training. Instead, they claim that the reason that batchnormalization works is that it smooths out the optimization landscape. This leads to more predictable gradients and thus allows for higher learning rates and more stable training. No matter the reason, experiments show the effectiveness of batchnormalization in practice.
The idea behind batchnormalization is that we normalize the inputs before each layer in the neural network. In order for this to be feasible computationally, the normalization has to be done for each dimension separately and done over the minibatch. Otherwise the model would have to go through all of the data, just for the normalization, each time the parameters are updated. For a minibatch $\mathcal{B} = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_m\}$ of observations with d-dimensions, $\boldsymbol{x} = (x^{(1)}, \dots, x^{(d)})$, the batchnormalization would be

$$\hat{\boldsymbol{x}_i} = \frac{\boldsymbol{x}_i - \boldsymbol{\mu}_{\mathcal{B}}}{\sqrt{\boldsymbol{\sigma}^2_{\mathcal{B}} + \boldsymbol{\epsilon}}}, \tag{4.32}$$

where $\boldsymbol{\mu}_{\mathcal{B}}$ is the minibatch mean and $\boldsymbol{\sigma}^2_{\mathcal{B}}$ is the minibatch variance. However, forcing the normalization could potentially lower the representational power of the layers. Therefore

we add in learnable parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ and add in a linear transformation. The full batchnorm transformation therefore becomes

$$T(\boldsymbol{x}_i) = \boldsymbol{\gamma} \odot \hat{\boldsymbol{x}}_{\boldsymbol{i}} + \boldsymbol{\beta} = \boldsymbol{\gamma} \odot \frac{\boldsymbol{x}_i - \boldsymbol{\mu}_{\mathcal{B}}}{\sqrt{\boldsymbol{\sigma}_{\mathcal{B}}^2 + \boldsymbol{\epsilon}}} + \boldsymbol{\beta} = \boldsymbol{x'}_i, \tag{4.33}$$

where $\boldsymbol{x'}$ is the output of the batchnorm. Then for the k'th dimension it is clear that setting $\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$ and $\beta^{(k)} = \mathrm{E}[x^{(k)}]$ we get

$$x'^{(k)}_i = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)} = \sqrt{\mathrm{Var}[x_i^{(k)}]} \frac{x_i^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}] + \epsilon^{(k)}}} + \mathrm{E}[x^{(k)}] \approx x_i^{(k)}, \tag{4.34}$$

which is (very close to) the identity transformation.

It is common to use batchnormalization or some related method in flows to stabilize training. While the original batchnormalization is a layer between layers of units in an artificial neural network, batchnormalization in normalizing flows works as a layer between transformation layers in a normalizing flow. *RealNVP* from [DSB16], for example, uses a standard batchnormalization between layers in the flow. In *Glow* from [KD18], the authors use what they call *actnorm* between layers in the flow. *Actnorm* is basically a version of batchnorm that only uses a learnable scale and bias term to do the normalization. That way, the normalization works even for very small minibatch sizes but is dependent on proper initialization. In [KD18], the authors initialize the scale and variance parameter such that the post-actnorm activations of a minibatch have zero mean and unit variance per channel.
In order to add batchnormalization as a layer into a normalizing flow, we need to be able to calculate the log determinant of the Jacobian. As the normalization is just a simple linear rescaling of each dimension independently, this becomes a simple product:

$$\log |\det J_{BN}(\boldsymbol{x})| = \log \left| \prod_{i=1}^{D} \frac{\gamma_i}{\sqrt{{\sigma^{(i)}}_{\mathcal{B}}^2 + \epsilon}} \right|. \tag{4.35}$$

Also the batchnorm has a simple inverse - we simply just rescale in the reverse order:

$$T^{-1}(\boldsymbol{x'}_i) = \boldsymbol{\mu}_{\mathcal{B}} + \frac{\boldsymbol{x}_i - \boldsymbol{\beta}}{\boldsymbol{\gamma}} \odot \sqrt{\boldsymbol{\sigma}_{\mathcal{B}}^2 + \boldsymbol{\epsilon}} = \boldsymbol{x}_i. \tag{4.36}$$

So as the batchnormalization has both inverse and well-defined (and fast to compute) log determinant, we can easily incorporate them as layers in flows. By doing this, we can build long flows by collecting transformation layers, batchnorm layers, and, if needed, permutation layers together in overall layers or steps and stacking those as depicted in Figure 4.3. Note that the first step has no batchnormalization, as we expect normalization of the data to have been done in data preparation.

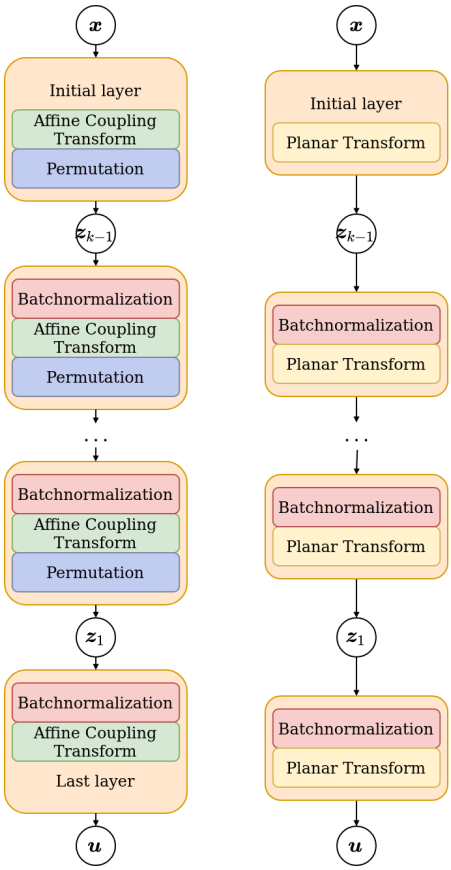


Figure 4.3: (Left) The layouts of the Affine Coupling Flow with permutes and batchnormalization. (Right) The Planar flow with added batchnormalization.

CHAPTER 5
Conditional Normalizing Flows

We now move on to how to do conditional density estimation using normalizing flows. As explained in Section 2.2 the goal of conditional density estimation is to model the conditional distribution of dependent variables $\boldsymbol{y}$ given conditioning variables $\boldsymbol{x}$, i.e., model $p(\boldsymbol{y}|\boldsymbol{x})$. The different normalizing flows we have discussed so far have modeled an unconditional distribution $p(\boldsymbol{y})$. So now, we extend the methods to take in external variables and modify the modeled distribution based on this. As was the case for unconditional normalizing flows, there are different ways of accomplishing this. Here we will present intuitive methods to extend the affine coupling layer and the planar layer. For the affine coupling layer, we extend the conditioner to take in conditioning parameters when generating parameters for the transformation. In the planar layer, we incorporate an FFNN similar to the affine coupling layers conditioner to generate parameters for the transformation. These extensions are simple to implement as they only alter the theory of the flows slightly. Another noteworthy method is to use Bayesian Networks like in [TT18].

The general approach is to extend normalizing flows by conditioning each of the transformations in the flow on $\boldsymbol{x}$. As such, we get the log conditional probability of an observation as:

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}) = \log p_u(\boldsymbol{u}) - \sum_{i=1}^{K} \log \left| \det \boldsymbol{J}_{f^{(k)}_{\boldsymbol{x},\boldsymbol{\theta}}}(\boldsymbol{z}_{k-1}) \right|, \tag{5.1}$$

where $f^{(k)}_{\boldsymbol{x},\boldsymbol{\theta}}$ is the k'th transformation in the flow conditioned on $\boldsymbol{x}$.

5.1 Conditional Affine Coupling Layers

Recall from Section 4.1 that the affine coupling layers use a conditioner, c, to generate parameters that are used by the transformer, τ. In order to do conditional density estimation, we can extend this to take in extra conditioning variables when generating the parameters for the transformation. This way, the transformation itself is not changed and hence the formulas for the transformation, inverse transformation, and log determinant of the Jacobian, i.e., equations (4.10), (4.12) and (4.14) are unchanged. However,

we need to change how the conditioner generates $\boldsymbol{h_i} = \{\boldsymbol{\alpha}_i, \boldsymbol{\beta}_i\}$ such that the context $\boldsymbol{x}$ is taken into account. In the case of the coupling layer, the conditioner is a simple feed forward neural network. Changing the weights of the network is an impossible task, so instead the network is extended to take in the context along with half of the dimensions. Such a coupling layer that takes in the context as well is called a *conditional coupling layer*. This approach has been used in [Ata+19], [Win+19], and [LH19], all of which applied normalizing flows to images. As the context might by high dimensional, e.g., if the context was an image, we follow [Win+19] and [LH19] and add another FFNN, the *conditioning network*[1], to the transformation. The idea is that the conditioning network extracts the features that actually have an impact on the flow. That way, the context added into the conditioner is already information-rich. Since the conditioning network is only a feature extractor, there is no reason to train a conditioning network for each coupling layer, as the features these networks find would probably be the same. Hence, the conditional normalizing flow shares the same conditioning network among all of the coupling layers. See figure 5.1 to see the structure of the conditional coupling layer. A

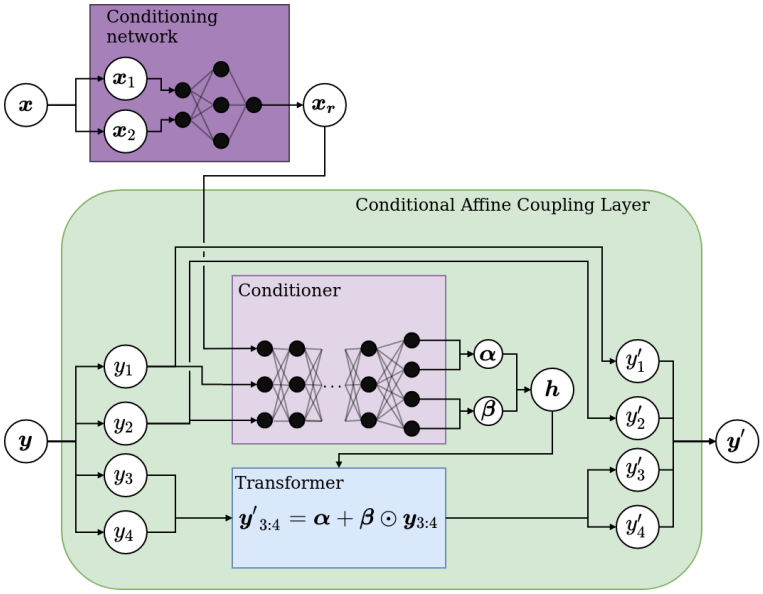


Figure 5.1: Layout of the conditional coupling layer. In this example $D = 4$ and $d = 2$ The conditioning variables $\boldsymbol{x}$ are sent through the *conditioning network* to create the rich conditioning variables $\boldsymbol{x_r}$. These are concatenated to half of the the input dependent variables, $\boldsymbol{y}_{<d}$ and fed into the *conditioner* to generate the variables for transformation, $\boldsymbol{h}$. These are then used to set the *transformer* s.t. it can transform the other half of the input dependent variables $\boldsymbol{y}_{>d}$ to generate the output of the layer $\boldsymbol{y'}$.

full flow consisting of conditional affine coupling transformations with permutation and batchnormalization layers can be seen in Figure 5.3.

[1]Not to be confused with the conditioner in the affine coupling layer.

5.2 Conditional Planar Flow

The planar layer discussed in Section 4.2 can be made conditional in a very similar fashion as the conditional affine coupling flow. In the unconditional flow the transformation has parameters $\boldsymbol{w}, \boldsymbol{v} \in \mathbb{R}^D$ and $b \in \mathbb{R}$. In order to make the flow conditional, we add an FFNN that takes the context as input and outputs $\boldsymbol{w}$, $\boldsymbol{v}$ and b. This network can be seen as a parallel to the conditioner in the affine coupling layer. This way $\boldsymbol{w}$, $\boldsymbol{v}$ and b becomes conditioned upon $\boldsymbol{x}$. The conditioning happens before $\boldsymbol{x}$ is transformed by the flow, hence the transformation, inverse, and absolute determinant of Jacobian are obtained as described in Section 4.2. As in the conditioned affine coupling flow, we add a higher level feed forward net to do feature extraction for the entire flow. The feature-rich context is then inputted to the conditioning network in each individual layer. The structure of the layer can be seen in Figure 5.2. A full flow can be seen in Figure 5.3

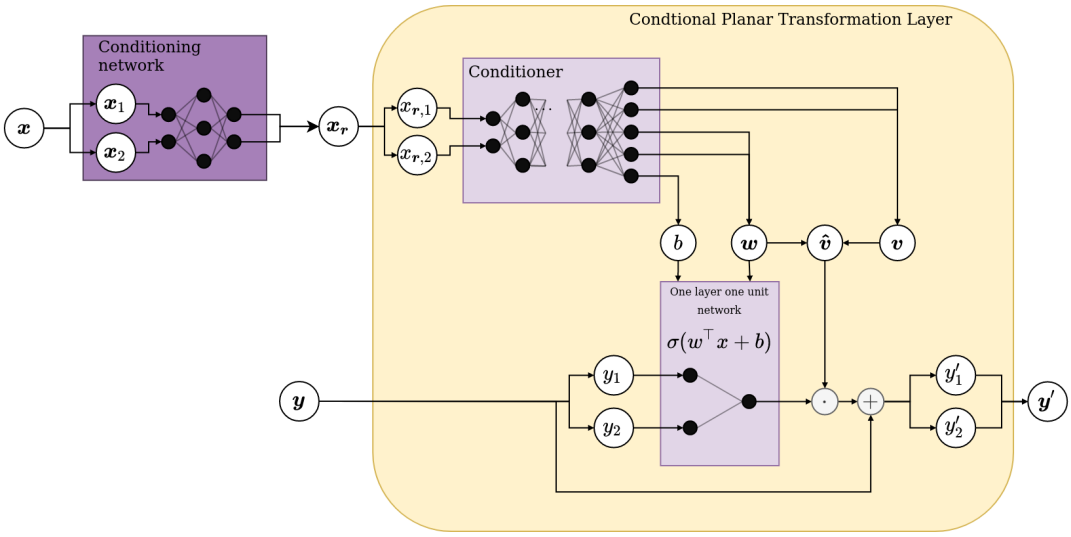


Figure 5.2: Diagram of the conditional planar transformation in the case of $D = 2$. Note that now the conditioner has the learnable parameters instead of b, $\boldsymbol{w}$ and $\boldsymbol{v}$

5.3 Semi-conditional Normalizing Flows for categorical conditioning variables

Sometimes the data sets we use conditional normalizing flows on is incomplete. If some of the conditioning variables are missing for some observations, we can still use these observations by training the conditioned normalizing flow with a semi-conditioned loss function. However, for tractability, this is only possible for categorical variables. The approach is similar to what they do in [Ata+19]. In the case where no data is missing we have that the loss function is simply the negative log likelihood as defined in Equation

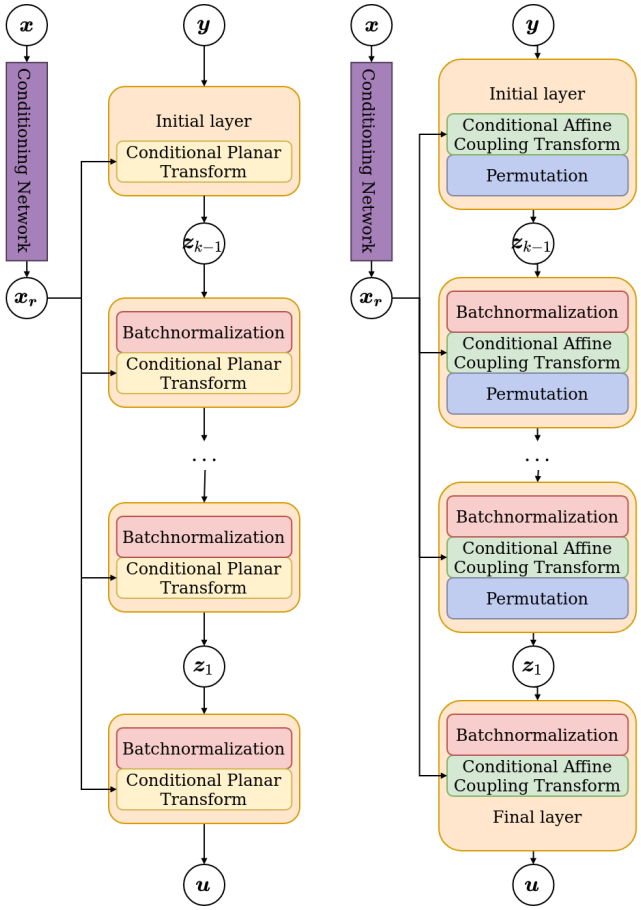


Figure 5.3: (Left) The layouts of the Condtitional Affine Coupling Flow with permutes, batchnormalization and shared conditioning network. (Right) The Conditional Planar flow with added batchnormalization and shared conditioning network.

(5.1), i.e.

$$L(\theta) = -\sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}} \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}), \tag{5.2}$$

where $\mathcal{D} = \{\boldsymbol{y}_i, \boldsymbol{x}_i\}_{i=1}^{N}$ is the data set. Now suppose that we have $\mathcal{D} = (\mathcal{D}_F, \mathcal{D}_S)$ where $\mathcal{D}_F = \{\boldsymbol{y}_i, \boldsymbol{x}_i\}_{i=1}^{N_F}$ is data with no missing data points and $\mathcal{D}_s = \{\boldsymbol{y}_i, \boldsymbol{x}_{\boldsymbol{s}}\}_{i=1}^{N_S}$ is data where $\boldsymbol{x}_{\boldsymbol{s}}$ is conditioning variables with some missing values $\boldsymbol{x}_{\boldsymbol{m}}$. We can then marginalize out these missing variables as

$$L(\boldsymbol{\theta}) = -\sum_{(\boldsymbol{y},\boldsymbol{x})\in\mathcal{D}_F} \log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}) - \sum_{(\boldsymbol{y},\boldsymbol{x_s})\in\mathcal{D}_S} \sum_{\boldsymbol{x}_m\in C} p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x}_{\boldsymbol{s}}, \boldsymbol{x}_{\boldsymbol{m}}) p_{x_m}(\boldsymbol{x}_{\boldsymbol{m}}), \tag{5.3}$$

where $p_{x_m}(\boldsymbol{x}_{\boldsymbol{m}})$ is some prior we set on the missing data and C is all possible combinations of values $\boldsymbol{x}_{\boldsymbol{m}}$ can take. Note that the last term requires $|C|$ passes to calculate the inner sum. Therefore if the missing conditioning variables can take many different values, the training of a conditional normalizing flow with the semi-conditional approach

will be much slower.

CHAPTER 6

Regularization of Conditional Normalizing Flows

Conditional density estimation methods are very prone to overfitting. Therefore it is important to use regularization when training conditional normalizing flows. The conditional normalizing flows proposed in section 5 are both based on artificial neural networks. Therefore it makes sense to try common regularization methods for such networks. We will look at two of the most common methods, namely *L2-regularization* and *dropout*, and compare them to a simple proposed regularization method for conditional density estimation called *noise regularization*. In this, chapter we first motivate $L2$-regularization and dropout, then explain the theory behind noise regularization.

$L2$-regularization is a classic regularization method in statistics. The method works by adding the $L2$-norm of the weights to the loss function. Let $\boldsymbol{W}$ be the weight matrix for a neural network, w_{ij} the individual weights of the matrix, and $\mathcal{L}$ some loss function. Then the $L2$-regularized loss function, $\hat{\mathcal{L}}$, is given by:

$$\hat{\mathcal{L}}(\boldsymbol{W}) = \mathcal{L}(\boldsymbol{W}) + \frac{\alpha}{2}\|\boldsymbol{W}\|_2^2 = \mathcal{L}(\boldsymbol{W}) + \frac{\alpha}{2}\sum_i \sum_j w_{i,j}^2. \tag{6.1}$$

The gradient of the loss function then becomes

$$\nabla_{\boldsymbol{W}} \hat{\mathcal{L}}(\boldsymbol{W}) = \alpha \boldsymbol{W} + \nabla_{\boldsymbol{W}} \mathcal{L}(\boldsymbol{W}). \tag{6.2}$$

Hence for gradient descent methods, the $\alpha \boldsymbol{W}$ term will always decrease the weights a bit each iteration. Therefore $L2$-regularization is also sometimes called *weight decay* in machine learning literature. Intuitively by regularizing the weights in a neural network, the adaptation the network learns from noise in the training data is decayed out while the weights set by actual information in the data survive. The decay rate α is a hyperparameter. For more on $L2$-regularization see [Nie18, Chapter 3].

Dropout regularization [Sri+14] randomly drops units from the neural network during training. Doing this discourages units from co-adapting too much. The intuition is that a unit cannot be certain that the connected units in the previous and next layer

are the same from batch to batch. Therefore it cannot depend on complex adaptations to the data together with these units. Instead, the unit has to create meaningful features on its own. Hence the network overall avoids co-adapting too much and thereby avoids overfitting. The dropping of units happens with a certain probability p that is a hyperparameter for the dropout regularization.

Both $L2$-regularization and dropout regularization work by regularizing the neural network. Either through the weights or the units in the network. How these regularizations end up modifying the target distribution found by the normalizing flow is unclear. Therefore we are interested in a regularization method that works in a more interpretable way. One such interpretable regularization for CDE is proposed by [Rot+19b]. The regularization method is called noise regularization, and the idea is simply to add noise to the data during training. I.e. given an observation from the data $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ we add noise as

$$\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{\xi}_x, \quad \boldsymbol{\xi}_x \sim K_x(\boldsymbol{\xi}_x) \tag{6.3}$$
$$\tilde{\boldsymbol{y}}_i = \boldsymbol{y}_i + \boldsymbol{\xi}_y, \quad \boldsymbol{\xi}_y \sim K_y(\boldsymbol{\xi}_y), \tag{6.4}$$

where $K_x(\xi_x)$ and $K_y(\xi_y)$ are noise distributions. To simplify the notation a bit we stack the dependent and independent variables: $\boldsymbol{z} := (\boldsymbol{x}^\top, \boldsymbol{y}^\top)^\top$. The above then becomes:

$$\tilde{\boldsymbol{z}}_i = \boldsymbol{z}_i + \boldsymbol{\xi}, \quad \boldsymbol{\xi} \sim K(\boldsymbol{\xi}). \tag{6.5}$$

Furthermore we denote $p(\boldsymbol{z}) = p(\boldsymbol{y}|\boldsymbol{x})$ The noise distribution is i.i.d. with expected mean 0 and standard deviation h. I.e.

$$\mathbb{E}_{\boldsymbol{\xi}\sim K(\boldsymbol{\xi})}[\boldsymbol{\xi}] = 0 \text{ and } \mathbb{E}_{\boldsymbol{\xi}\sim K(\boldsymbol{\xi})}[\boldsymbol{\xi}\boldsymbol{\xi}^\top] = h^2\boldsymbol{I}. \tag{6.6}$$

Adding noise to the data during training of a normalizing flow acts as a smoothness regularizer penalizing "spiky" target distributions. To see this we look at the MLE loss function as discussed in Section 3.2:

$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{n=1}^{N} p(z_n). \tag{6.7}$$

In order to show how the noise regularization works as a smoothness regularizer we see how the likelihood of the noise perturbed observations look. We approximate the probability density function of an observation perturbed by a noise vector with a Taylor series expansion:

$$p(\boldsymbol{z}_i + \boldsymbol{\xi}) = p(\boldsymbol{z}_i) + \boldsymbol{\xi}^\top \nabla_z p(\boldsymbol{z}_i)|_{z_i} + \frac{1}{2}\boldsymbol{\xi}^\top \nabla_{z_i}^2 p(\boldsymbol{z}_i)|_{z_i}\boldsymbol{\xi} + \mathcal{O}(\boldsymbol{\xi}^3). \tag{6.8}$$

Assuming that the added noise is small, higher order terms $\mathcal{O}(\boldsymbol{\xi}^3)$ can be ignored. The expected probability is then:

$$\begin{aligned}
\mathbb{E}_{\boldsymbol{\xi}\sim K(\boldsymbol{\xi})}\left[p\left(\boldsymbol{z}_i+\boldsymbol{\xi}\right)\right] &\approx \mathbb{E}_{\boldsymbol{\xi}\sim K(\boldsymbol{\xi})}\left[p(\boldsymbol{z}_i)+\boldsymbol{\xi}^\top\nabla_z p(\boldsymbol{z}_i)|_{\boldsymbol{z}_i}+\frac{1}{2}\boldsymbol{\xi}^\top\nabla_{\boldsymbol{z}_i}^2 p(\boldsymbol{z}_i)|_{\boldsymbol{z}_i}\boldsymbol{\xi}\right] \\
&= p\left(\boldsymbol{z}_i\right)+\mathbb{E}_{\boldsymbol{\xi}\sim K(\boldsymbol{\xi})}[\boldsymbol{\xi}]^\top\nabla_z p(\boldsymbol{z})\Big|_{\boldsymbol{z}_i}+\frac{1}{2}\mathbb{E}_{\boldsymbol{\xi}\sim K(\boldsymbol{\xi})}\left[\boldsymbol{\xi}^\top\nabla_z^2 p(\boldsymbol{z})\Big|_{\boldsymbol{z}_i}\boldsymbol{\xi}\right] \\
&= p\left(\boldsymbol{z}_i\right)+\frac{1}{2}\mathbb{E}_{\boldsymbol{\xi}\sim K(\boldsymbol{\xi})}\left[\boldsymbol{\xi}^\top\mathbf{H}^{(i)}\boldsymbol{\xi}\right] \\
&= p\left(\boldsymbol{z}_i\right)+\frac{1}{2}\mathbb{E}_{\boldsymbol{\xi}\sim K(\boldsymbol{\xi})}\left[\sum_j\sum_k \xi_j\xi_k\frac{\partial^2 p(\boldsymbol{z})}{\partial z^{(j)}\partial z^{(k)}}\Big|_{\boldsymbol{z}_i}\right] \\
&= p\left(\boldsymbol{z}_i\right)+\frac{1}{2}\sum_j\mathbb{E}_{\boldsymbol{\xi}}\left[\xi_j^2\right]\frac{\partial^2 p(\boldsymbol{z})}{\partial z^{(j)}\partial z^{(j)}}\Big|_{\boldsymbol{z}_i}+\frac{1}{2}\sum_j\sum_{k\neq j}\mathbb{E}_{\boldsymbol{\xi}}\left[\xi_j\xi_k\right]\frac{\partial^2 p(\boldsymbol{z})}{\partial z^{(j)}\partial z^{(k)}}\Big|_{\boldsymbol{z}_i} \\
&= p\left(\boldsymbol{z}_i\right)+\frac{h^2}{2}\sum_j\frac{\partial^2 p(\boldsymbol{z})}{\partial z^{(j)}\partial z^{(j)}}\Big|_{\boldsymbol{z}_i} \\
&= p\left(\boldsymbol{z}_i\right)+\frac{h^2}{2}\operatorname{tr}\left(\mathbf{H}^{(i)}\right),
\end{aligned} \tag{6.9}$$

where $\boldsymbol{H}^{(i)}=\nabla_z^2 p(\boldsymbol{z})|_{\boldsymbol{z}_i}$.
As observable, the second derivatives of the probability density function are added to the likelihood. This means that in MLE target distributions with large second derivatives, i.e., "spiky" distributions, are penalized. Thereby the added noise works as a smoothness regularization of the normalizing flow.

The standard deviation of the added noise, h, is a hyperparameter that determines how much the regularization smooths the target distribution. How much regularization is needed is a complex question but [Rot+19b] proposes two heuristics, derived for Kernel Density Methods[1], that depends on the dimensionality of the data and how much training data is available. The first heuristic is known as the *rule of thumb* or *Silverman's rule* ([Sil86]) and is given by

$$h = 1.06\hat{\sigma}n^{-\frac{1}{4+d}}, \tag{6.10}$$

where $\hat{\sigma}$ denotes the estimated standard deviation of the data, n is the number of data points, and d is the dimensionality of the data[2]. This heuristic is optimal assuming that both the added noise $K(\boldsymbol{\xi})$, and target distribution, $p(\boldsymbol{z})$ are Gaussian. However, if $p(\boldsymbol{z})$ is highly non-Gaussian this heuristic might not decay fast enough with the number of data observations hence [Rot+19b] also proposes the *square root decay schedule*:

$$h = n^{-\frac{1}{1+d}}. \tag{6.11}$$

[1] In fact noise regularization and Kernel Density Methods are very similar mathematically, see [Rot+19b] for more.

[2] Note that this is both the dependent and independent variables.

The choice of noise added can be seen as a hyperparameter, and we will test the influence it has on the regularization in the next chapter.

CHAPTER 7

Experiments on synthetic data

In this chapter, we will experiment with the conditional affine coupling flows on a synthetic data set. Since the affine coupling flow was developed for high-dimensional settings, we first wish to evaluate its ability to model distributions in low dimensions. After this, we will experiment with different regularization methods on the conditional affine coupling flow. We will compare the described noise regularization with other common machine learning regularization methods.

In order to evaluate the conditional normalizing flows and regularization methods, we create a synthetic data set. We base the synthetic data set upon the Two Moons data set from the Python package *Scikit-Learn* ([Ped+11]). This data set has also been used in [Ata+19]. Normally the data is generated by random sampling from two moons placed opposite each other; see Figure 7.1. We extend Scikit-Learn's implementation of Two

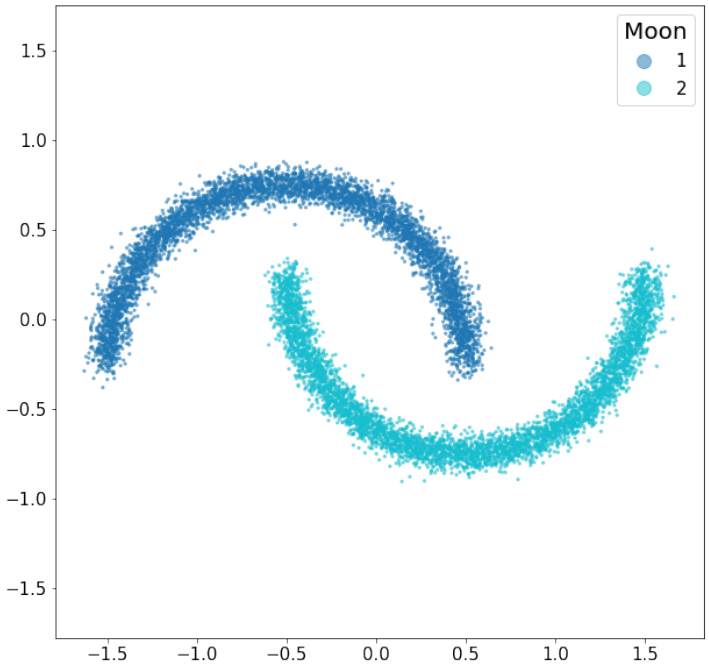


Figure 7.1: 1000 samples from the Two Moons dataset

Moons to also randomly rotate the moons around the center, as depicted Figure 7.2 and with translation of the point in both direction as depicted in Figure 7.3. The dependent variables are then the coordinates of the sampled point with with moon, rotation, and translation as independent variables. We then wish to train a conditional normalizing flow such that it can fit the dependent variables given the independent variables.

7.1 Experiments with Conditional Affine Coupling Flows

First, we fit a conditional normalizing flow based on the conditional affine coupling layer to see if it is able to condition on the rotation and the moon class. We fit the flow to the data using MLE as described in section 3.2, and visualize how the flow transforms the base distribution into the target distribution. The results can be seen in Figure 7.4. As can be seen in the figures the model fits the data nicely. However, with some tail at one end of the moon in Figure 7.4b. We can also visualize how each of the layers in the normalizing flow transforms the distribution. This can be seen in Figure 7.5. It is worth noting that even though the flow seems to be seven transformations long, three of those transformations are permutations. As described in Section 4.1, these layers only permutes the dimensions and cannot be trained in the 2d case. So it is only

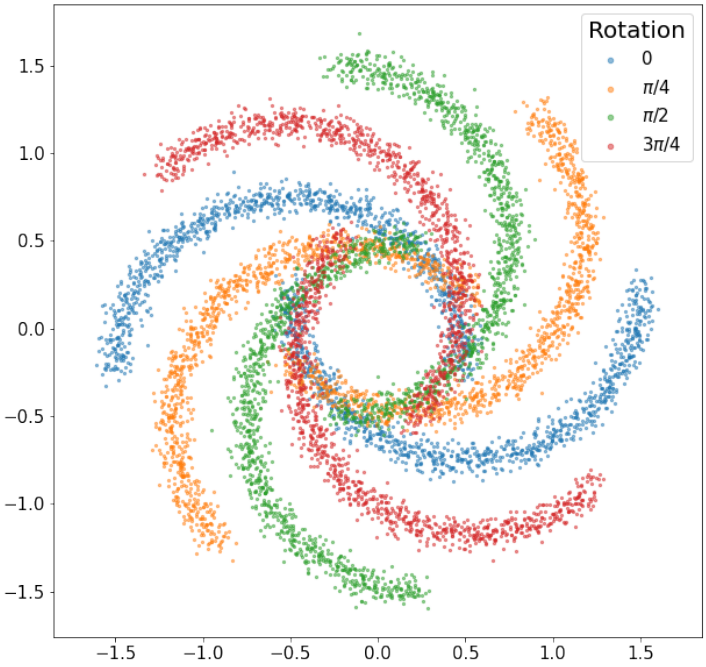


Figure 7.2: 8000 samples from the Rotating Two Moons dataset, with 4 different rotations. Note that the actual data used is randomly rotated between 0 and 2π radians and also has a moon variable as in Figure 7.1

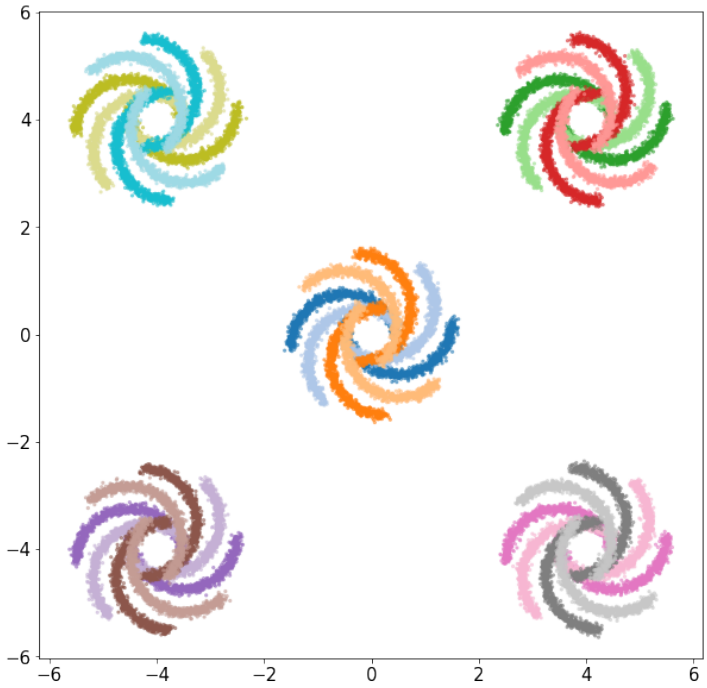
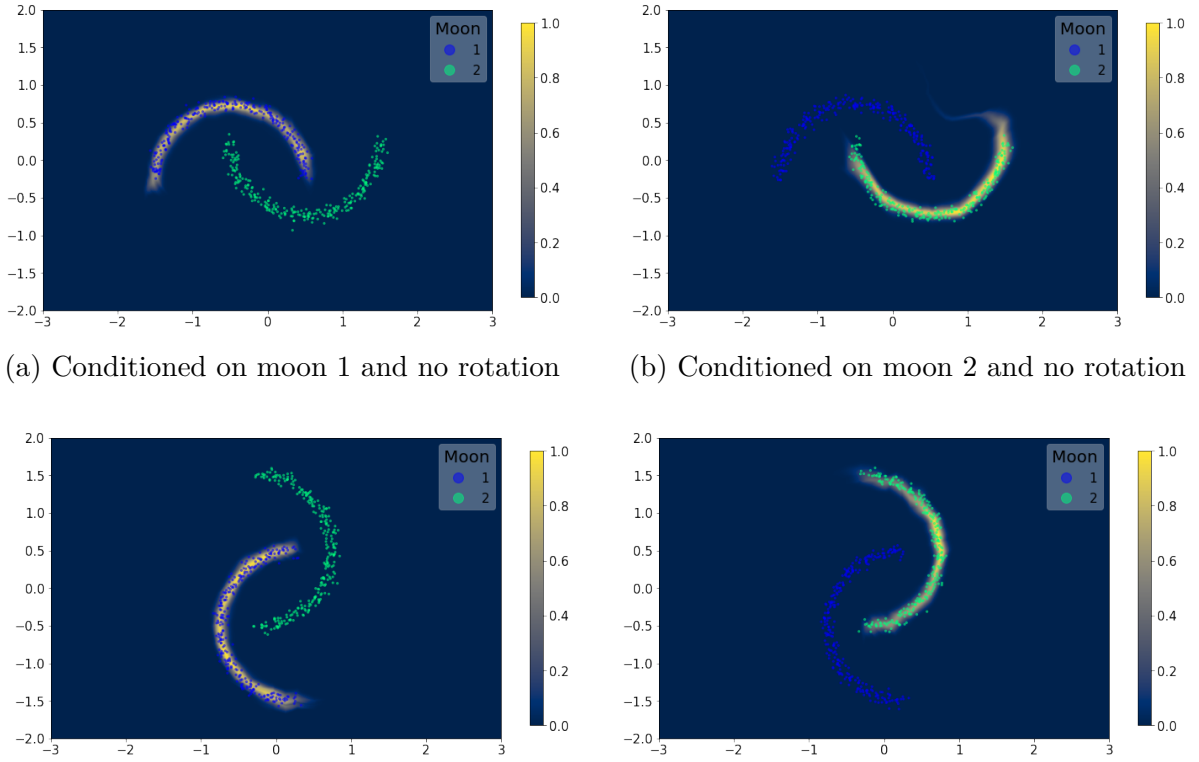


Figure 7.3: 40000 samples from the Translated Rotating Two Moons dataset, with 4 different rotations and 5 different translations. Note that the actual data used is randomly rotated between 0 and 2π radians, randomly translated between -4 and 4 in each direction and also has a moon variable as in Figure 7.1

the four affine coupling layers that learn how to transform the base distribution into a half-moon. Furthermore, the figure nicely visualizes how the affine transformations stretch the distribution along one dimension in each layer. We will return to how this affects the modeled distribution later. From these simple tests on the synthetic data, we conclude that the conditional affine coupling flow is able to condition the modeled target distribution on conditioning variables in this low-dimensional setting.

7.2 Experiments with Noise Regularization

In Section 6 we presented the noise regularization for conditional density estimation. In this section, we compare this regularization to $L2$-regularization and dropout, both of which are very common regularization methods for artificial neural networks. At the same time, we evaluate the impact of using the batchnorm layers presented in Section 4.3. First, we generate a data set of 100.000 samples from the two moons with rotation. Then we fit a conditional affine coupling flow without batchnorm layers, using the conditional affine coupling layer from Section 4.1. We train the flow on the data using no regularization, with weight decay and with dropout as well as with noise regularization with noise standard deviation that follows a constant, square root and Silverman's rule



(a) Conditioned on moon 1 and no rotation

(b) Conditioned on moon 2 and no rotation

(c) Conditioned on moon 1 and $\pi/2$ rotation

(d) Conditioned on moon 2 and $\pi/2$ rotation

Figure 7.4: Here we can see a conditional affine coupling flow trained on the rotating two moons data set with rotation and moon as conditioning variables. The modelled distribution can be seen as the yellow overlay.

heuristic. We do this for data sizes from 100 to 9000. The results can be seen in Figure 7.6. From the figure, we can see that the flow without regularization and the flow with $L2$-regularization are very unstable during training. Therefore for most data sizes, the mean of the three runs is way worse than the regularized models due to one or more of the runs failing during training due to instability. This is especially bad for small data sizes. However, for larger data sizes, we see that both the unregularized and $L2$-regularized flows achieved mean log likelihoods sometimes beat the other regularized models. We also note that these flows achieve the best log likelihood of all regularization methods when they do not fail, as can be seen by the shaded area. For the noise regularized models, we can see that the flows following Silverman's rule and a square root schedule are over regularized. Therefore they achieve a lower log likelihood and show much lower variance between training runs. As described in Chapter 6, Silverman's rule and the square root schedule both assume that the target distribution is a Gaussian. The two moons dataset is highly non-Gaussian, which could explain the schedule's bad performance. The constant noise regularization heuristic performs much better, achieving the best results without much variance, showing that noise regularization works better with a smaller noise standard deviation. Interestingly, dropout performs almost as good, al-

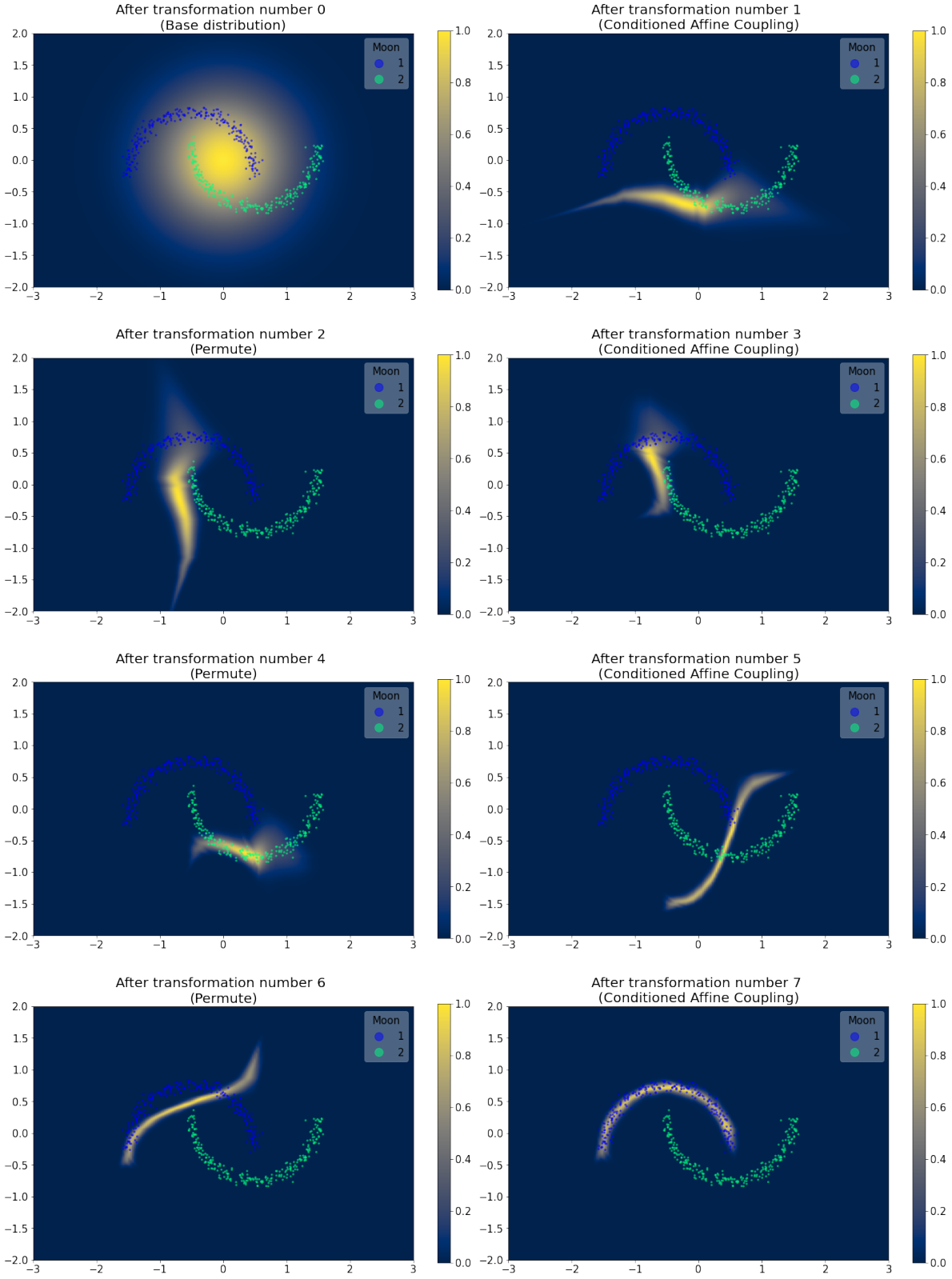


Figure 7.5: Here we can see how the conditional affine coupling flow transforms the base distribution into the target distribution. The flow is trained without batchnorm layers.

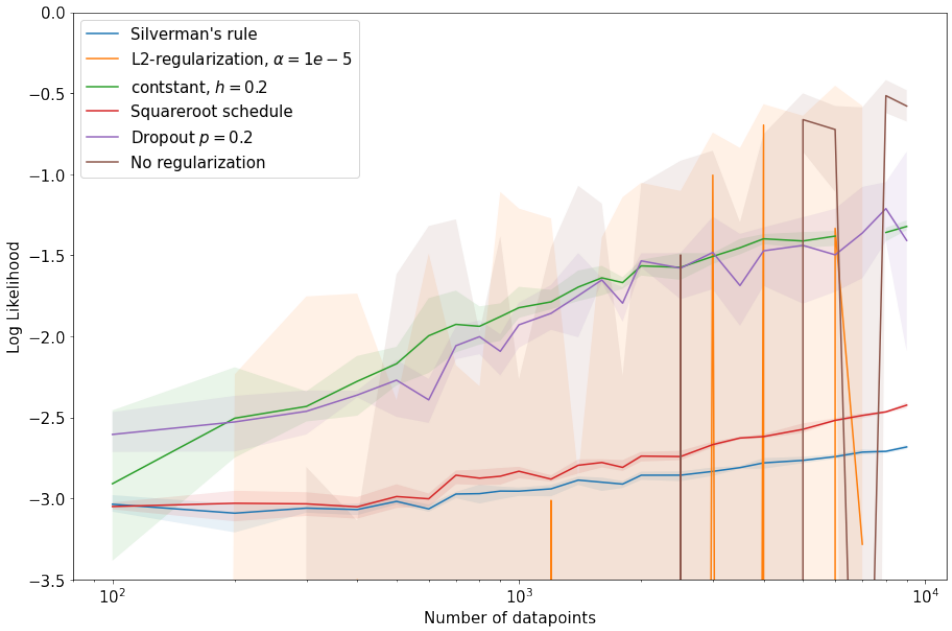


Figure 7.6: The achieved likelihood of the conditional normalizing flow model without batchnorm layers using different regularization shcemes. The lines show the mean of three runs with the area between the best run and worst run shaded. The experiments are done for data sizes between 100 and 9000 data points

beit with a bit more variance.

Next, we repeat the same test but with batchnorm layers added into the flow. We add batchnorm layers into all steps except the first one as depicted in Figure 5.3. The results with batchnorm layers can be seen in Figure 7.7. As can be seen in the figure, the flows without regularization and with $L2$-regularization benefits greatly from the added batchnorm layers. For larger data sizes, these flows achieve the best log likelihoods. Especially the $L2$-regularized flow performs well on data sizes over 1000 observations. However, for smaller data sizes, both of these flows still have stability problems. The performance of Silverman's rule and the square root schedule is unchanged by the addition of the batchnorm layers, and they still over-regularize. The constant noise schedule and the dropout regularization performs slightly better for smaller data sizes with the addition of batchnorm layers. For larger data sizes the performance seems unchanged.

We then add translation to the two moons data set to see how the regularization handles more complex data sets. The setup is the same, but now we have four conditioning variables instead of two. The results can be seen in Figure 7.8. As can be seen in the figure, the more complex data results in that none of the data sizes are large enough for the flow without regularization and the flow with $L2$-regularization to become stable. Silverman's rule and the square root schedule are still over regularizing, limiting the achieved log likelihood of the flows. Again the noise regularized flow with constant noise

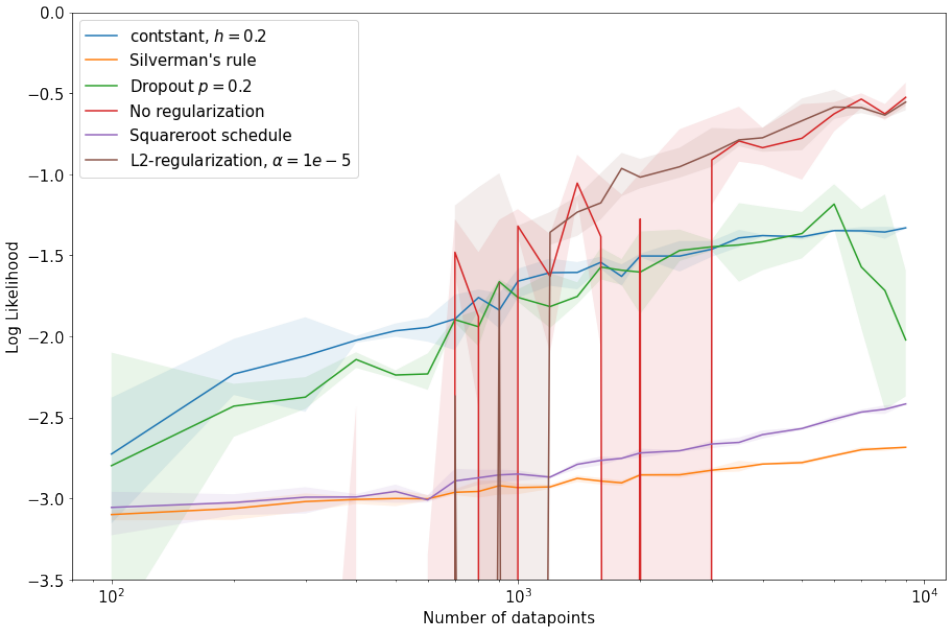


Figure 7.7: The achieved likelihood of the conditional normalizing flow model with batchnorm layers using different regularization shcemes. The lines show the mean of three runs with the area between the best run and worst run shaded. The experiments are done for data sizes between 100 and 9000 data points

schedule performs well. Interestingly dropout regularization performs worse compared to the constant noise schedule for the data with four conditioning variables while it was comparable for the data with two conditioning variables.

From these results, we conclude that noise regularization is, in fact, a good way to regularize normalizing flows. Unfortunately, it seems that Silverman's rule and the square root schedule are over regularizing our test data sets. Therefore we compare the constant noise regularization schedule for different amounts of noise to see if any noise standard deviation performs better than the others. We return to the data with two conditioning variables. We try to fit flows with four different constant noise schedules and compare them to an unregularized and $L2$-regularized flow. The results can be seen in Figure 7.9. As can be seen in the figure, the lower the noise regularization's standard deviation, the lower the overfitting the flow experiences for larger data sizes. For the flows with a standard deviation of 0.005 and 0.0005, the achieved log likelihood for data sizes over 1000 observations is on par with the unregularized and $L2$-regularized flow. However, for smaller data sizes, we can see that the best noise standard deviation changes. For data sizes under 300 observations, it is the largest standard deviation 0.02 that performs best while between 300 observations and 800 observations, it is a standard deviation of 0.01 that performs the best. As such, it would make sense to be able to create some schedule like Silverman's rule or the square root schedule that finds the best standard deviation depending on the data size. Unfortunately, as discussed earlier,

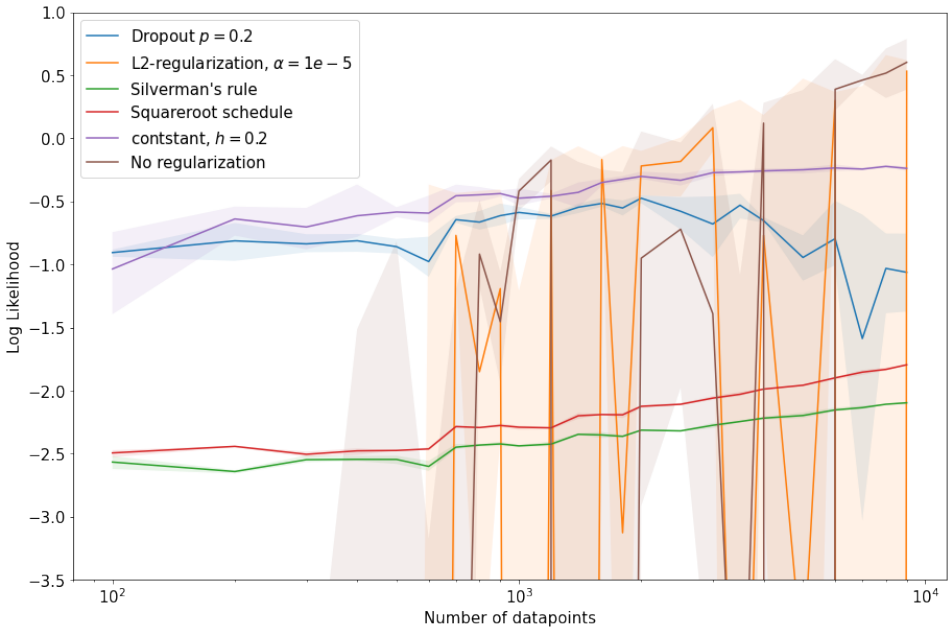


Figure 7.8: The achieved likelihood of the conditional normalizing flow model with batchnorm layers using different regularization schemes. The lines show the mean of three runs with the area between the best run and worst run shaded. The experiments are done for data sizes between 100 and 9000 data points with rotation, translation and moon as conditioning variables.

these schedules assume a Gaussian target distribution, and in our tests, it was found that they overregularize. Hence such a schedule would need to be tailored to the data at hand to perform optimally. So if one has a data set with fixed data size, our results show that one is better off just finding the best noise standard deviation without considering schedules. However, if one has a problem where there comes more and more data from the same distribution, perhaps, it is worth trying to create a schedule that fits the data distribution.

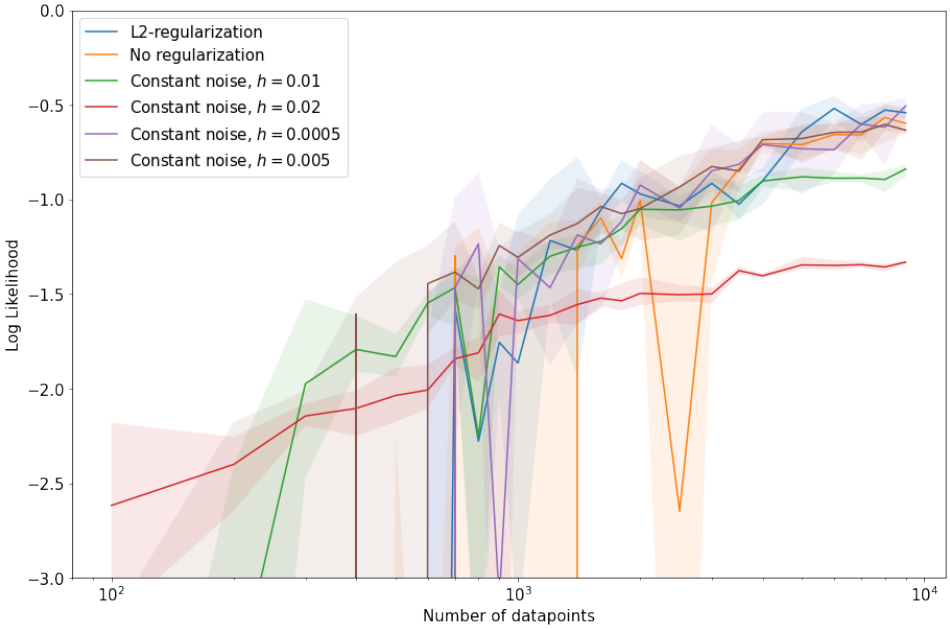


Figure 7.9: The achieved likelihood of the conditional normalizing flow model with batchnorm layers using different regularization schemes. The lines show the mean of three runs with the area between the best run and worst run shaded. The experiments are done for data sizes between 100 and 9000 data points with rotation and moon as conditioning variables.

CHAPTER 8

Modelling taxis in New York

In the last chapter, we found that the conditional affine coupling flow was able to model conditional distributions in a low dimensional setting. In this chapter, we extend the experiments to the real-world problem of geospatial demand estimation. We start by experimenting with the conditional affine coupling flow, showing its strengths and limitations. We then compare it to the conditional planar flow and a mixed flow with both kinds of layers.

8.1 NYC Yellow Taxi Data

A popular data set for geospatial demand estimation is the NYC Yellow Taxi dataset from the New York City Taxi & Limousine Commission[1]. The Taxi & Limousine Commission store information of all yellow taxi trips in New York City. One such data set was used by Kaggle for a competition[2]. This data set contains information of over 1.4 million taxi trips in the central New York area from January 2016. This data set has been used with normalizing flows before in [TT18] and [Rot+19b]. As in [Rot+19b] we will follow the data preprocessing of [Dut+18][3] to create a data set for conditional density estimation.
In the cleaning of the data, we remove all observations where:

- The dropoff or pickup location is outside of the central New York area.
- Any observations where the Euclidean norm of the change in coordinates between pickup and dropoff is less than 0.00001. This corresponds to the trip being less than 1.5 meters long.
- All trips shorter than 30 seconds or longer than 3 hours.

The generated data set has the trip dropoff location in coordinates as dependent variables and the pickup location, hour of the day, and day of the week as conditioning variables. The hour of day and day of the week has been transformed using a sine-cosine transformation. So we have 2-D target values the latitude and longitude of the dropoffs and six conditional variables: the latitude and longitude of the pickup locations and the

[1] `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`
[2] `https://www.kaggle.com/c/nyc-taxi-trip-duration`
[3] The preprocessing is made available at `https://github.com/hughsalimbeni/bayesian_benchmarks`

cosine and sine transformation of the time of day and day of the week.

8.2 Conditional Normalizing Flows on NYC Yellow Taxi Data

We experiment with using conditional normalizing flows on the NYC Yellow Taxi data. Our experiments will start with experimenting with conditional affine coupling flows, examining how the conditioning of the flow impacts the modeled distribution. We train three different kinds of flows on the data. First, we train an unconditional affine coupling flow only based on the dropoff locations. Next, we train a conditional normalizing flow that takes in both the pickup location and the time variables. Lastly, we train a conditional affine coupling flow that only takes in the time variables. To speed up the training of the flows, we take out a subset of 400000 taxi trips from the full data set. We split the subset 80/20 into training and test data, scale the data, and train three of each flows to the data using MLE. The flows all have ten layers. The specifics of the flows can be seen in Table 8.1. All of the flows were trained with noise regularization with a standard deviation of 0.08 and batchnormalization until convergence. In order to compare the models we calculate the log likelihoods on the test data. The results can be seen in Figure 8.1. As can be seen in the figure, the conditional model with both pickup location and time is able to achieve a higher log likelihood than the other models. However, the conditional model with only time dependency is still able to beat out the unconditional model. Therefore we can conclude that the conditional affine coupling flow is able to condition the distribution dropoff locations on the pickup locations and time. It is worth noting that the results also clearly show that the dropoff location is much more dependent on the pickup location than the time.
We then visualize the modeled distributions of the dropoff locations from the different models. In Figure 8.2 an unconditional model is visualized. As can be seen in the figure, the model places most of the distribution in Manhattan. This makes sense as by far the most taxi trips are completed in Manhattan. Note that the flow is complex enough to model central park as an unlikely dropoff location. It can also be seen that the model tries to place some distribution on the two airports, which are also popular taxi trip

Flow model	Number of layers	Conditioner depth	Conditioner hidden dims	Context network depth	Context network hidden dims
Unconditional	10	4	18		
Conditional only on time	10	4	18	5	16
Conditional on time and pickup	10	4	18	5	16

Table 8.1: The number of layers and the dimensions of the neural nets in the flows fitted to the NYC Yellow Taxi data. Note that the conditioner and context neural nets sizes have note been optimized.

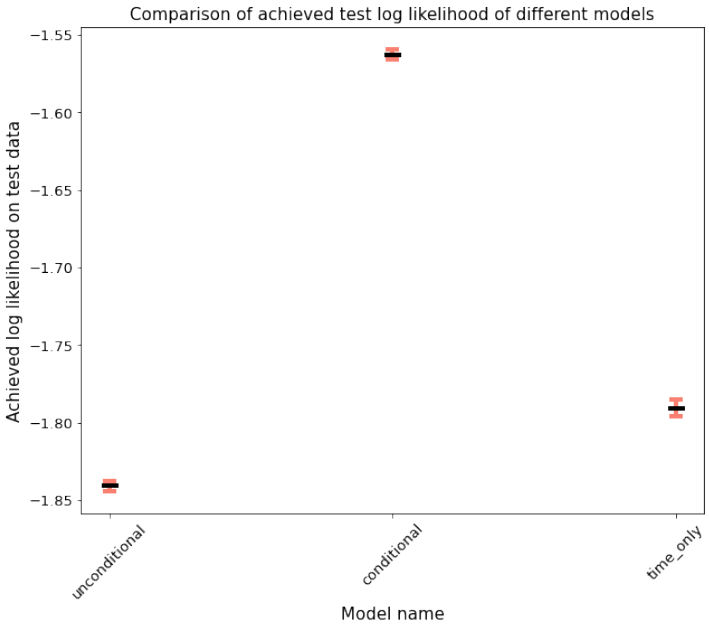


Figure 8.1: The achieved log likelihood on the test data of the 10 layer flows on the NYC Yellow Taxi data set of 400000 taxi trips. The black line shows the mean achieved log likelihood and the whiskers show the worst and best log likelihood over 3 runs

destinations. However, in order to do so, the model has to create paths of non-zero probability density to these areas, which could be a limitation of the model. We call these paths "probability paths" and we will look more into these paths shortly.

In Figure 8.3 a conditional model with both pickup location and time is visualized conditioned on four different contexts. As can be seen in the figure, the modeled distribution varies greatly when conditioned on different pickups locations. For pickups in Manhattan, the model predicts that a dropoff in Manhattan is by far the most likely as well. For pickups in Brooklyn, Manhattan is still modeled to be the most likely, but now the distribution is much more spread out. For pickups at JFK airport, we see that Manhattan is again the most likely but denser just south of Central Park. Furthermore, we can see that the distribution also changes significantly when the pickup time changes. For pickups on Manhattan at 18:00 the distribution is almost solely in downtown Manhattan, but at 2:00 the distribution is much more spread out with more probability in the northern part of Manhattan and Brooklyn.

In Figure 8.4 a conditional affine coupling flow only conditioned on the time variables is visualized. As can be seen in the figure, the modeled distribution varies quite a bit when conditioned on different hours of the day. For example, both of the airports are much more likely during the morning than they are during the night. This would make sense as there are probably more commuters at 7 in the morning than at midnight. The change in modeled distribution, when conditioned at the same time on different days, seems much smaller even when compared between weekdays and weekends. We would have expected a change in commuters to and from jobs. However, perhaps the lack of

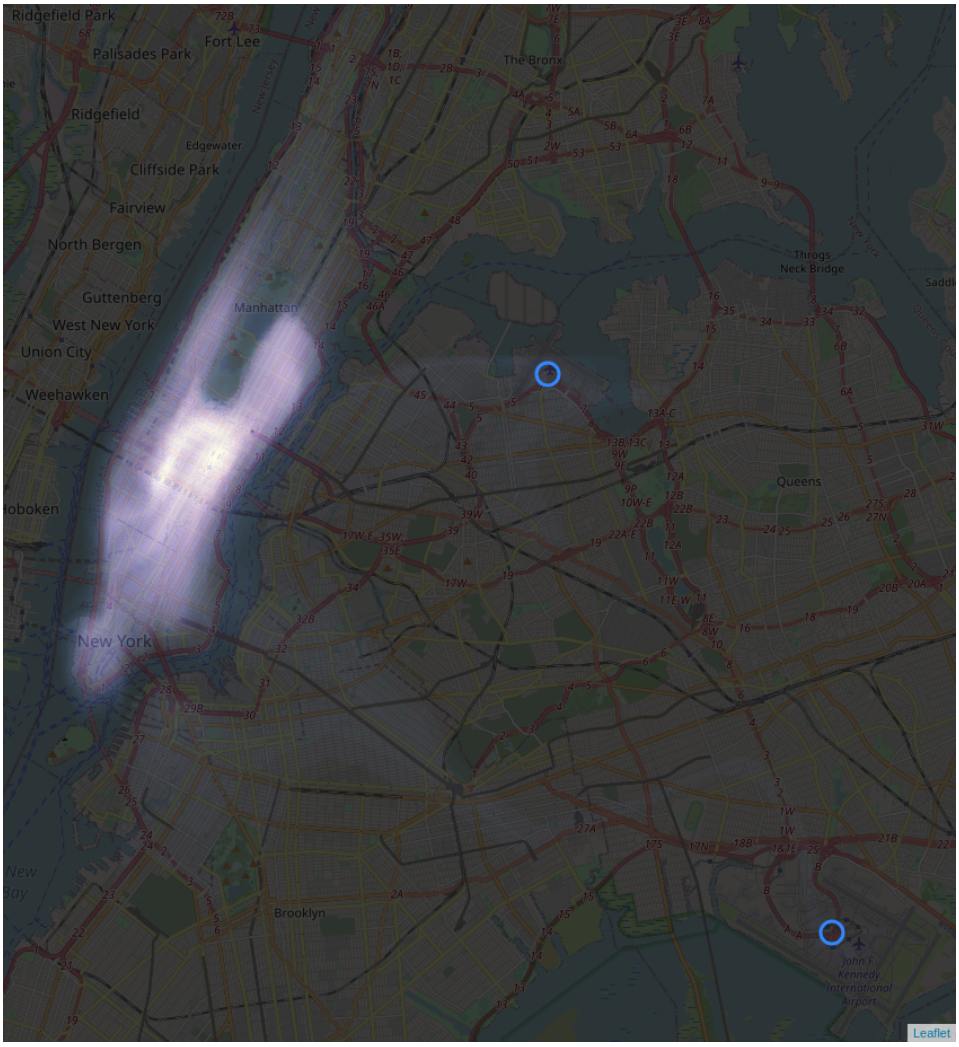
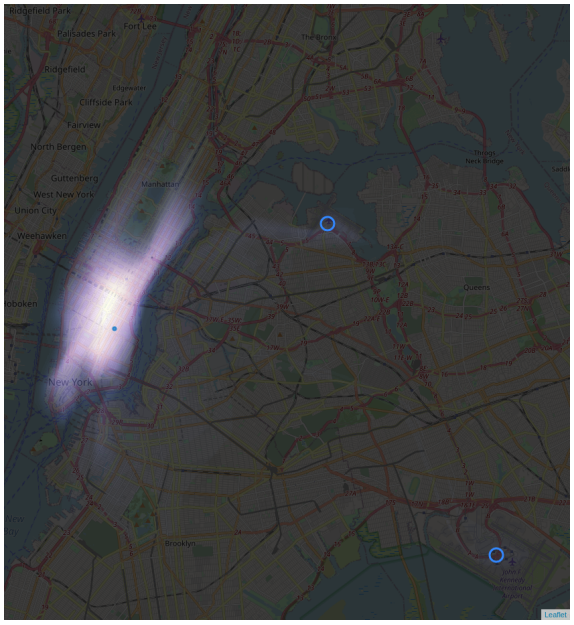


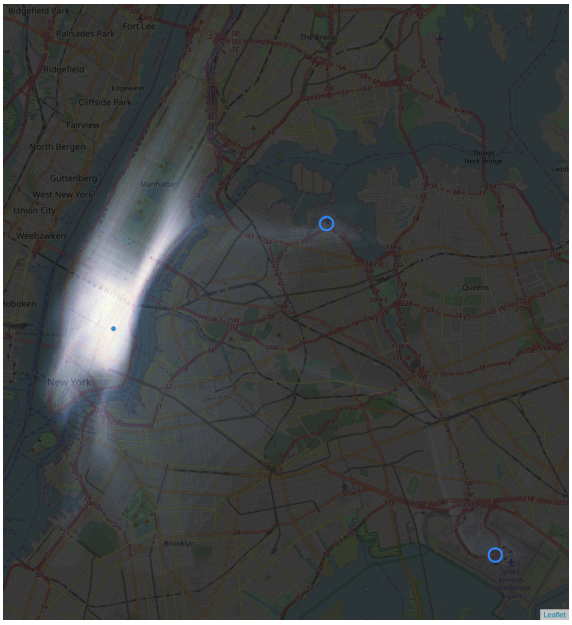
Figure 8.2: Here is a 10 layer unconditional affine coupling flow trained on the NYC yellow taxi data set. The log likelihood of the airports depicted at the terminal coordinates.

this could be explained by tourists being the main users of taxis or that people in New York either also work on Saturdays or use taxis to commute to other activities.

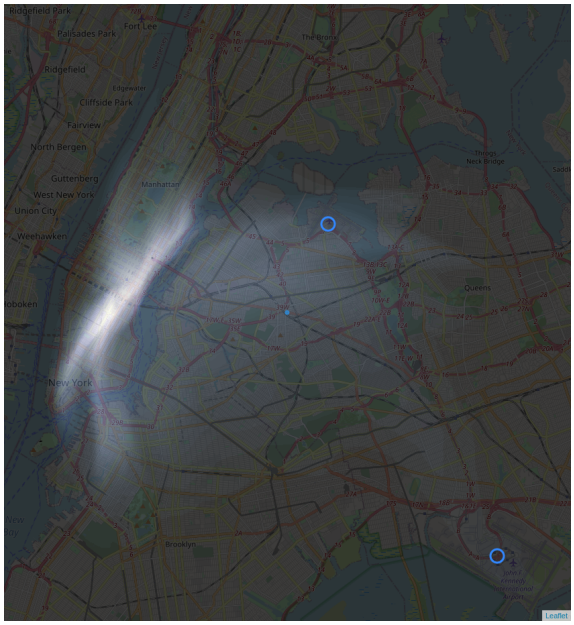
In all of the different affine coupling models on the NYC Taxi, we see that the models have to make stretches of probability in order to cover the airports. In Figure 8.4a and 8.3b this is especially visible with a path stretching from Manhattan to LaGuardia and JFK. We conjecture that this is an artifact from limitations in the model. This claim is backed by the fact that the shape of these paths varies from trained models with the same architecture, training scheme, and comparable log likelihoods on the test data. In Figure 8.5 the three different time only conditional affine coupling flows trained to make Figure 8.1 are visualized in the area between the airports. All of the models have the same hyperparameters, are all conditioned on 7:00 Saturday, and achieve between



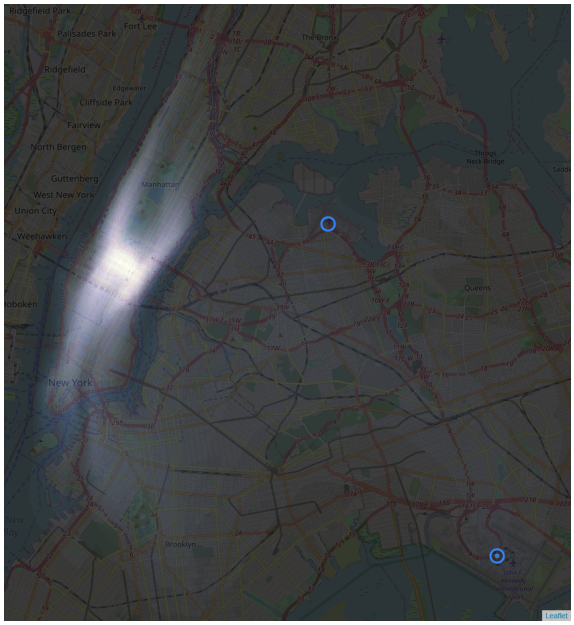
(a) Conditioned on 18:00 Tuesday with pickup on Manhattan



(b) Conditioned on 02:00 Tuesday with pickup on Manhattan

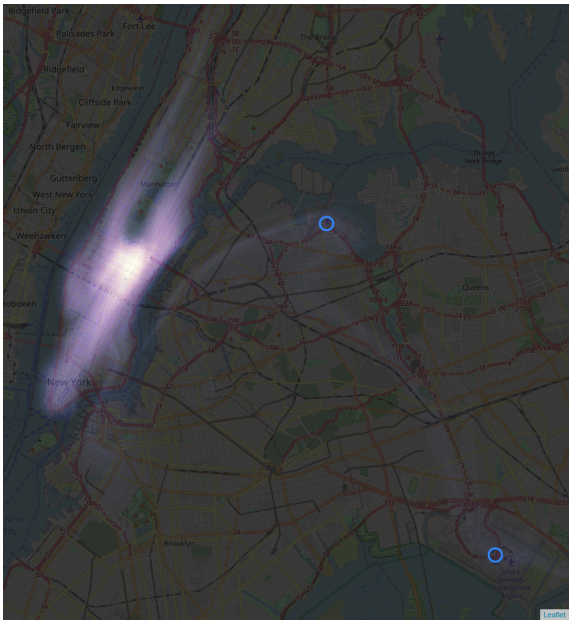


(c) Conditioned on 18:00 Tuesday with pickup in Brooklyn

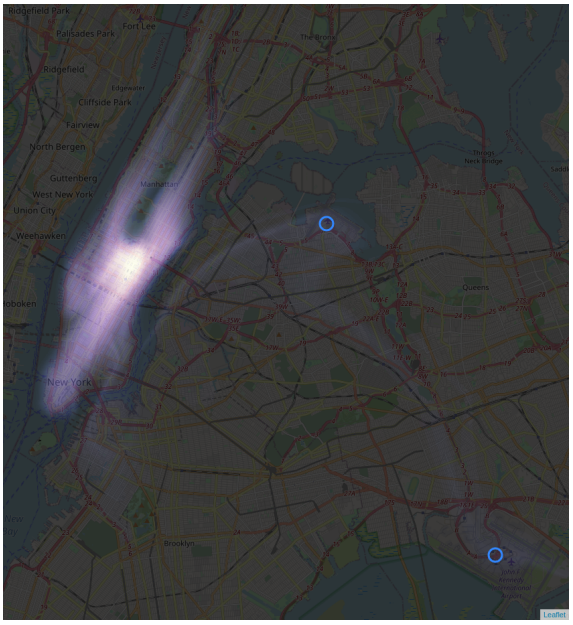


(d) Conditioned on 18:00 Tuesday with pickup at JFK airport

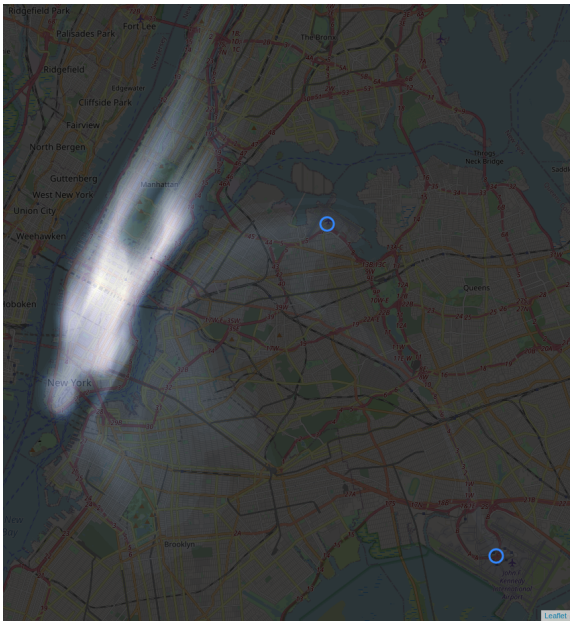
Figure 8.3: Here is a 10 layer conditional affine coupling conditioned on pickup location and time variables. The model is trained on the NYC Yellow Taxi data set and conditioned on four different contexts. The log likelihood of the airports is depicted at terminal coordinates.



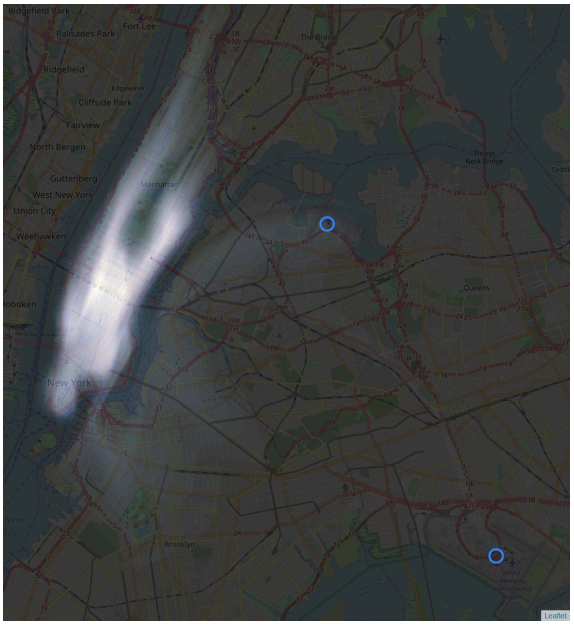
(a) Conditioned on 07:00 Monday



(b) Conditioned on 07:00 Saturday



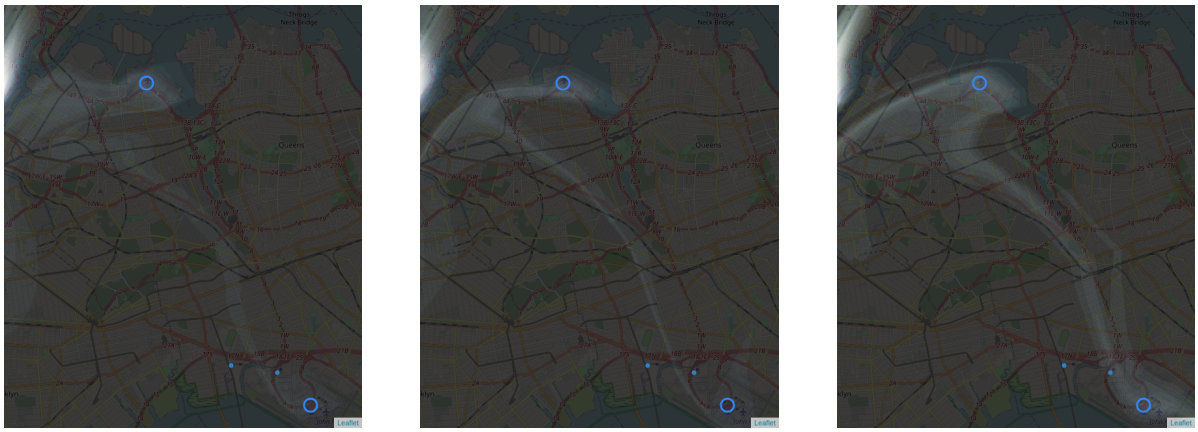
(c) Conditioned on 00:00 Monday



(d) Conditioned on 00:00 Saturday

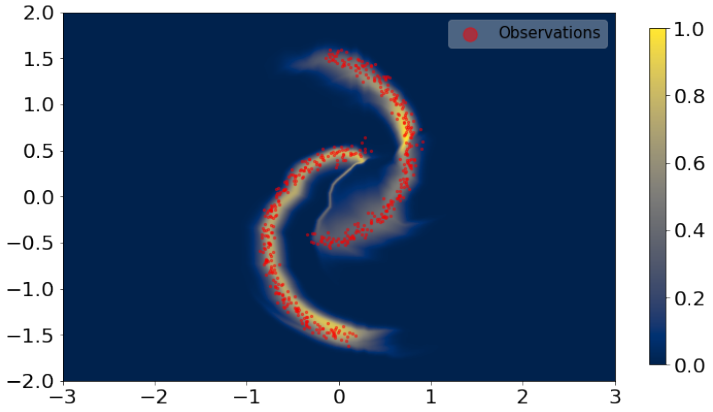
Figure 8.4: Here is a 10 layer conditional affine coupling conditioned on time variables. The model trained on the NYC Yellow Taxi data set and conditioned on four different contexts. The log likelihood of the airports is depicted at terminal coordinates.

−1.785 and −1.796 log likelihood on the test data. However, even though the models have the same hyperparameters and achieve comparable log likelihoods on the test data, we can see that the shape of the probability path between the airports varies quite a bit. In order to visualize the limitation in the affine coupling flow that causes this, we return to the Two Moon synthetic data. If we do not condition the flow on which moon to fit, the flow has to fit both of the disjoint moons at the same time. In Figure 8.6, three conditional affine coupling flows with varying depth are visualized. All of the flows are conditioned on a rotation of 1.5 radians. In Figure 8.6a, we can see that the flow with only four layers is having trouble with fitting the edges of the two moons and has a clear probability path between the moons. In Figure 8.6b, the flow with 24 layers is much better at clearly defining the edges of the two moons, but the flow still has a significant probability path between the moons. In Figure 8.6c the flow with 48 layers has comparable edges to the 24 layer flow, but now the probability path between the moons is much thinner. Looking at the definition of the Affine Coupling Layer, we can see why we end up with these paths. The affine transformer in the coupling layer described in Equation (4.10) is only a linear transformation on half of the dimensions based upon the other half, i.e., for the 2d case the affine coupling layer can only stretch and translate the distribution in one direction in each coupling layer. Using the affine transformation, the affine coupling layer would have to be able to condition the parameters of the transformation on the dimension being transformed. As the conditioner takes in the other dimension, this is not possible. As such, the Affine Coupling Layer can only increase or decrease the variance in a dimension and translate it, based on the other dimension. This is clearly visible in Figure 7.5, where we can see that even though the transformation is conditioned on half of the dimensions and potentially some conditioning variables, it is still only a stretching and translation in one dimension of the distribution. This means

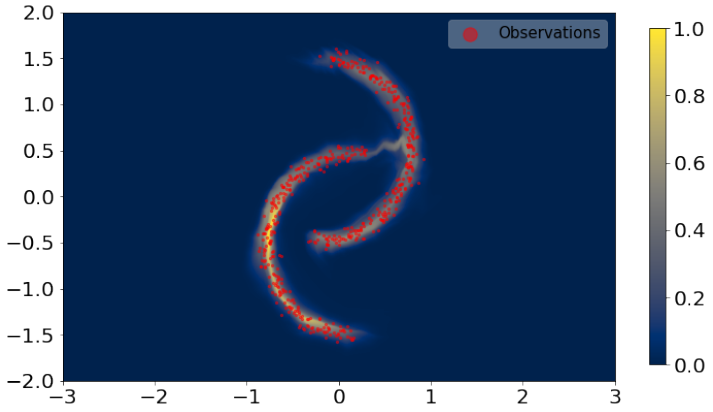


(a) (b) (c)

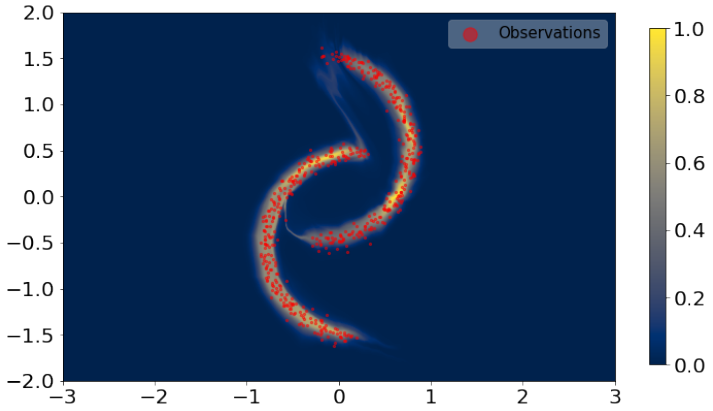
Figure 8.5: The probability paths between the airports from three different conditional affine coupling flows conditioned on time variables. The only difference between the models are the training, all hyperparameters are the same. All models are conditioned on 07:00 Saturday.



(a) 4 layers



(b) 24 layers



(c) 48 layers

Figure 8.6: Three conditional affine coupling flows fitted to the Two Moons synthetic data. The flows are only conditioned on rotation and all flows are conditioned on a rotation of 1.5 radians.

that an Affine Coupling Layer has no way of creating two disjoint probabilities, since it has no way of cleanly separating the distribution into disjoint pieces, hence why we see the probability paths. However, for deeper flows, this probability path can be stretched more thinly, as can be seen in 8.6.

In order to test how this applies to flows trained on geospatial data, we try to fit some deeper flows to the NYC Yellow Taxi data. We fit the same three types of flows as in Figure 8.1 but for depths of 24 layers and 48 layers as well. The results can be seen in Figure 8.7. From the figure, it is clear that deeper flows, in general, achieve a better log likelihood on the test data. This could be because the deeper flow has the ability to make thinner probability paths, or it could be because the deeper flows have more complex edges. In order to test this, we visualize the conditional models conditioned on both pickup location and time for the conditional affine coupling flows of depth 10, 24, and 48. We chose the flow conditioned on both pickup location and time as the modeled distributions have very clear paths when conditioned on a Manhattan pickup. The visualizations can be seen in Figure 8.8. As can be seen in the figure, the overall distributions of the three models are quite similar. However the paths going from Manhattan to the airports vary in shape. This is more visible in Figure 8.9, where the same models are visualized on a smaller area around the airports. As can be seen in the figure, the paths become thinner and thinner, the more layers the flow has. This follows the results from synthetic data we found in Figure 8.6. Based on this, it is clear that in order for an affine coupling flow to be able to properly model a two-dimensional distribution, the flow has to be of adequate depth. Otherwise, the modeled distribution will have clear probability paths between areas of distribution that are disjoint in the true distribution. Even for flows of 48 layers, we still have probability paths, as can be seen in Figure 8.9c. Due to time constraints, we will not try to fit deeper affine conditional flows to the NYC Yellow Taxi data set but try to use the conditional planar flow instead.

Another finding from the above experiments worth mentioning is the variance in the modeled distribution between flows with identical hyperparameters. It is a known problem that models based on artificial neural networks have problems with multiple local optima. As such normalizing flows can get stuck in different local optima based on initialization, resulting in different modeled distributions. This can be seen in Figure 8.5, where flows with identical hyperparameters and comparable log likelihood on test data have quite different modelled distributions. This is especially problematic for the flows with very detailed conditioning variables, as can be seen in Figure 8.10. As can be seen in Figure 8.10a and 8.10b, where the models are conditioned on 06:00 Tuesday with pickup in Brooklyn, the modelled distributions are quite different between the two flows even though they have the same hyperparameters. The log likelihoods of the test data under the two models are -1.495 and -1.514 for flow 1 and 2, respectively, so they are comparable. If we look at the same models conditioned on 18:00 Tuesday with pickup in Manhattan the distributions are much more similar, see Figure 8.10c and 8.10d. This is probably because there is much more data with contexts similar to this. Therefore the model has more data to train on for this context,, and the modeled distribution for

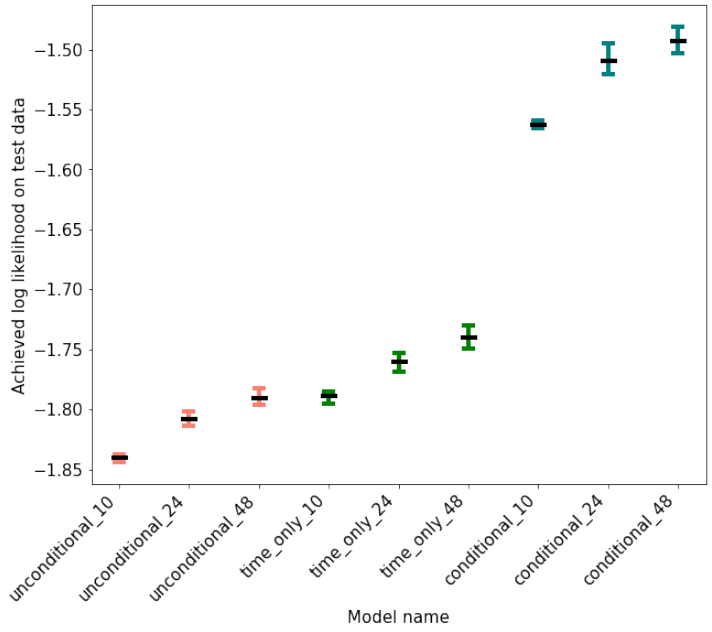
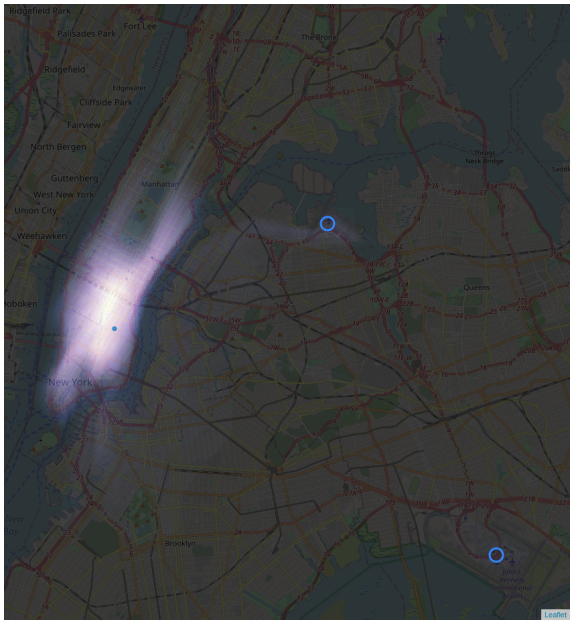


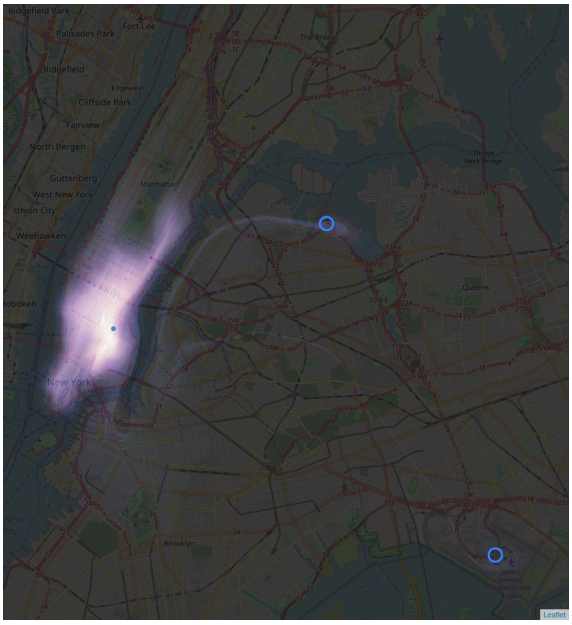
Figure 8.7: The achieved log likelihood on the test data of flows if different depths on the NYC Yellow Taxi data set of 400000 taxi trips. Apart from the depths, written in the model name, the flows have the same hyperparameters as in Table 8.1. The black line shows the mean achieved log likelihood and the whiskers show the worst and best log likelihood over 3 runs

this context has more say in the achieved log likelihood on the test data. Therefore in order to select a model that can be used in decision making, solely comparing the models based on the achieved log likelihood on a test set might not show which model performs best in all contexts. We will return to other ways to compare models when creating models for the Donkey Republic data in the next chapter.

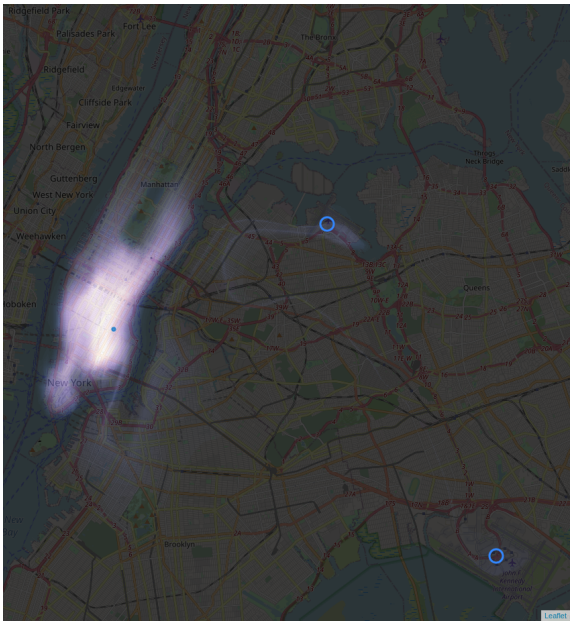
One potential way to deal with the problem with the probability paths could be to use a planar flow instead. Where the affine coupling flow stretches the dimensions iteratively, the planar flow can stretch all dimensions at once, allowing all dimensions to affect each other. The transformation can be seen as expanding or contracting along a hyperplane. As such, the planar flow can "cut" the distribution by placing the hyperplane in the middle of the distribution and expanding. This is still only a stretch and hence not a clean cut, but as seen in Figure 8.11 the transformation is still able to separate the distributions cleanly. In Figure 8.12, we can see that the planar flow is able to fit the two moons data better with more layers, as was the case with the affine coupling flow. However, our experience with fitting the planar flows to the two moons showed that they were more unstable during training and more vulnerable to bad initializations than the affine coupling flow, especially for the shallow flows. However, the conditional planar flow has no problem with probability paths and is able to model two disjoint half-moons.



(a) 10 flow conditioned on 18:00 Tuesday with pickup on Manhattan.

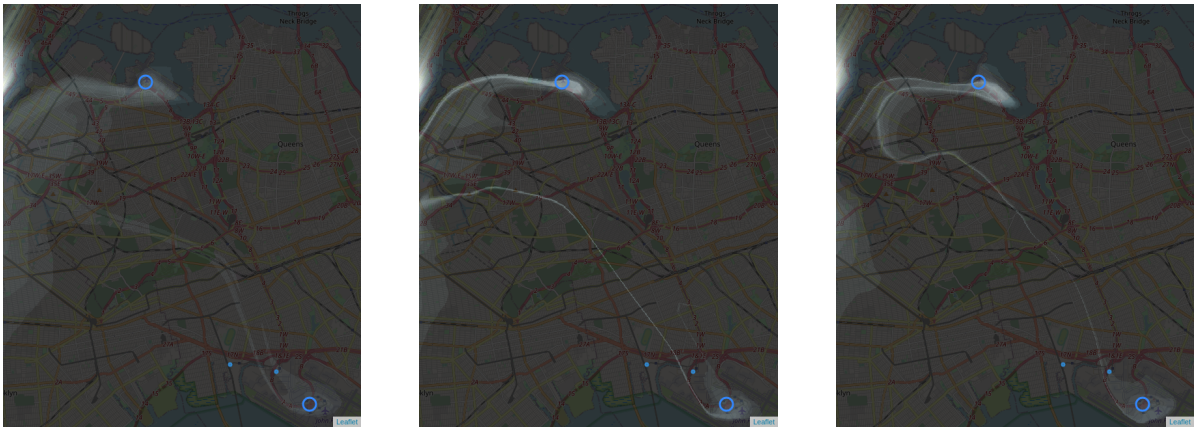


(b) 24 layer flow conditioned on 18:00 Tuesday with pickup on Manhattan.



(c) 48 layer flow conditioned on 18:00 Tuesday with pickup on Manhattan.

Figure 8.8: Three conditional affine coupling flows with varying depth. All flows are conditioned on the same time and have the same hyperparameters except for the depth of the flow.

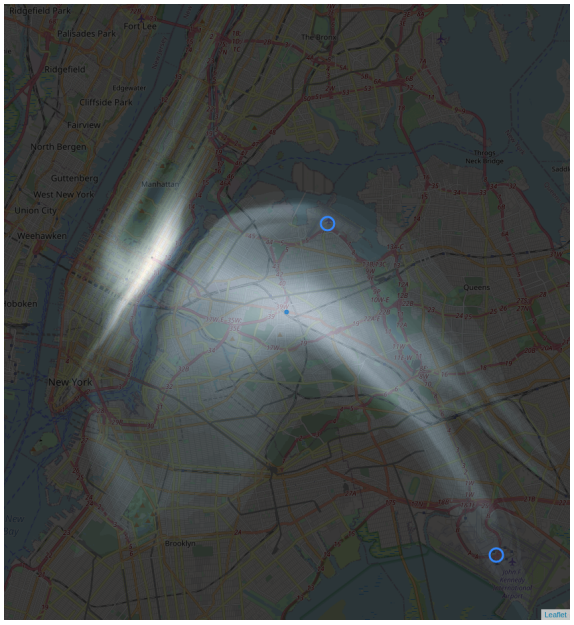


(a) 10 layers. (b) 24 layers. (c) 48 layers.

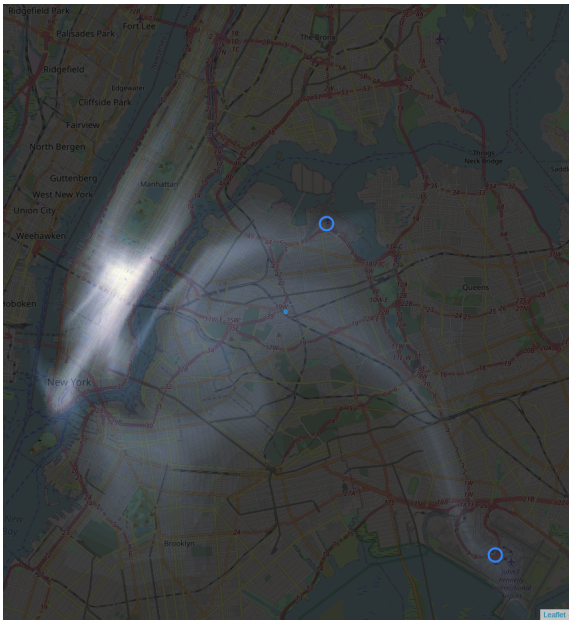
Figure 8.9: The probability paths between the airports from three different conditional affine coupling flows of varying depths. All of the flows are conditioned on 18:00 Tuesday with a pickup on Manhattan.

We tried to fit the conditional planar flow to the NYC dataset. We condition it with both pickup location and time variables. We fit a flow with 48 layers in order to try and match the best conditional affine coupling flow. We trained three flows, and the achieved log likelihoods on the test data for the planar flow models were between -1.77 and -1.81, much worse than the corresponding conditional affine coupling flow. The best model is visualized in Figure 8.13. As can be seen in the figure, the planar flow model is still able to condition the target distribution on the pickup location and time. However, the modeled distribution has a much lower complexity compared to the conditional affine coupling flow of the same size. However, the conditional planar flow does not have the problem with probability paths. This can be seen in Figure 8.13b where there is clear disjoint probability areas around JFK airport and Queens. It is also the case when conditioning the flow on pickup on Manhattan but it is not very visible in Figure 8.13a. In Figure 8.14, we have zoomed in on the area that showed clear paths for the conditional affine coupling flow. For the planar flow, the problem is gone, and we just have clear disjoint probabilities around Manhattan and JFK.

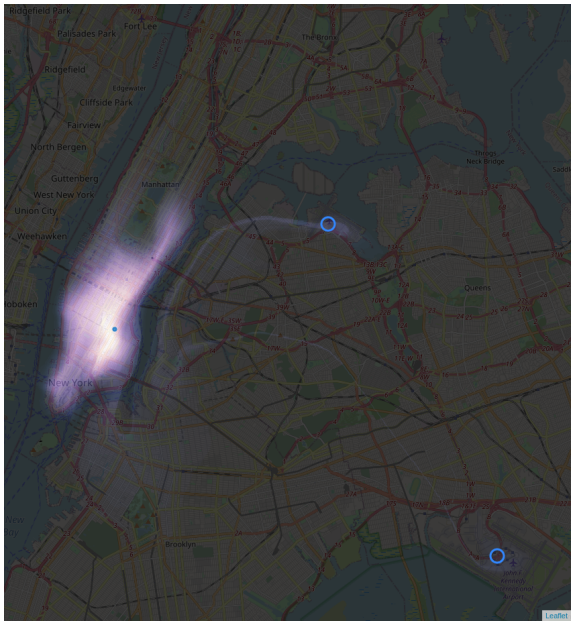
So the planar flow clearly has the ability to model disjoint probability densities, but it has a problem of less complex probabilities than conditional affine coupling flows of the same size. The complexity of the modeled distribution could, of course, be improved with a deeper flow. However, the layer architecture of discrete normalizing flows allows us to try and mix the different kinds of layers into a single flow. The idea is that this mixed flow has the disjoint probabilities of the conditional planar flow and the complexity of the affine coupling flow. The proposed architecture of the flow can be seen in Figure 8.15. In the proposed flow, we do a conditional affine coupling transformation in each layer before we do a conditional planar transformation. This order of layers was chosen as it showed the best performance on shallow networks. However, optimizing the



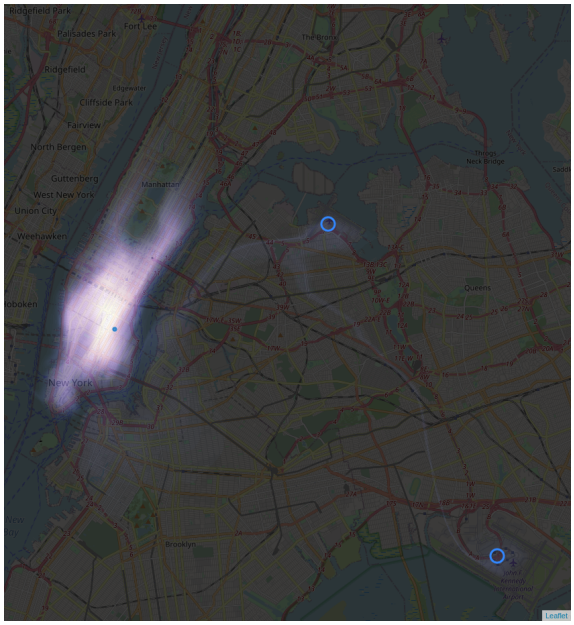
(a) Flow 1 conditioned on 07:00 Tuesday with pickup in Brooklyn



(b) Flow 2 conditioned on 07:00 Tuesday with pickup in Brooklyn



(c) Flow 1 conditioned on 18:00 Tuesday with pickup in Manhattan



(d) Flow 2 conditioned on 18:00 Tuesday with pickup in Manhattan

Figure 8.10: Comparison of the modelled distributions for two conditional affine coupling flows conditioned on two different contexts.

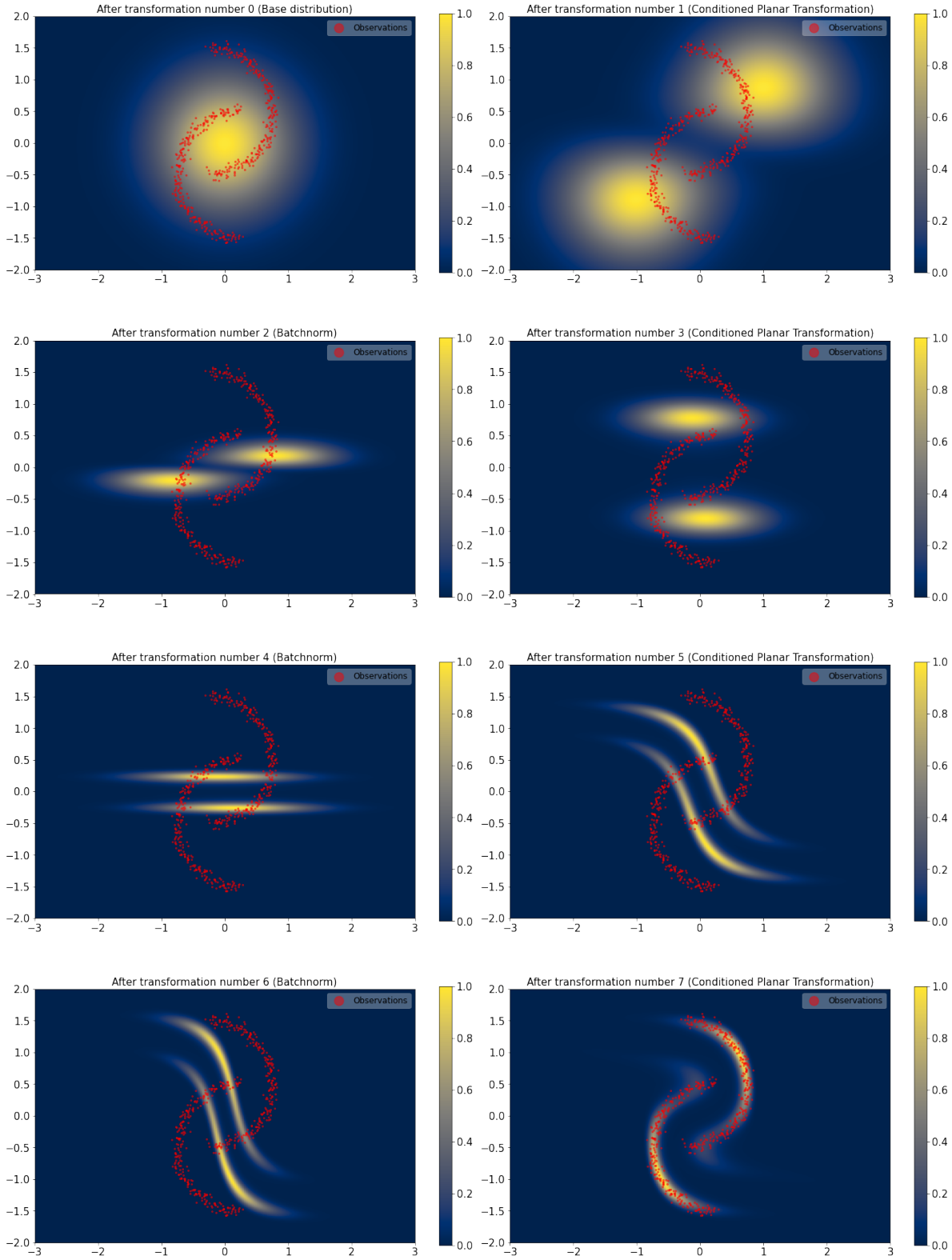
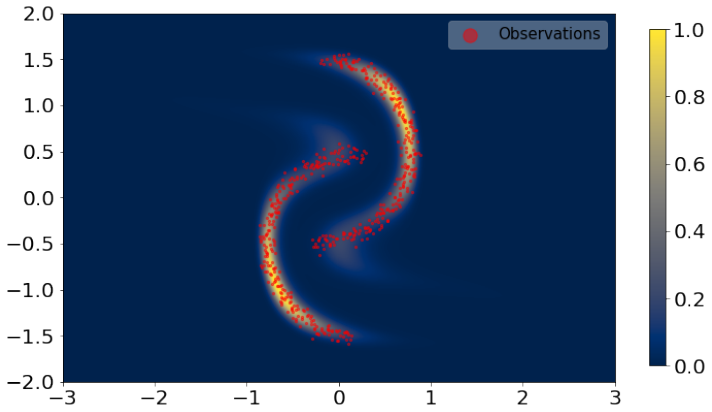
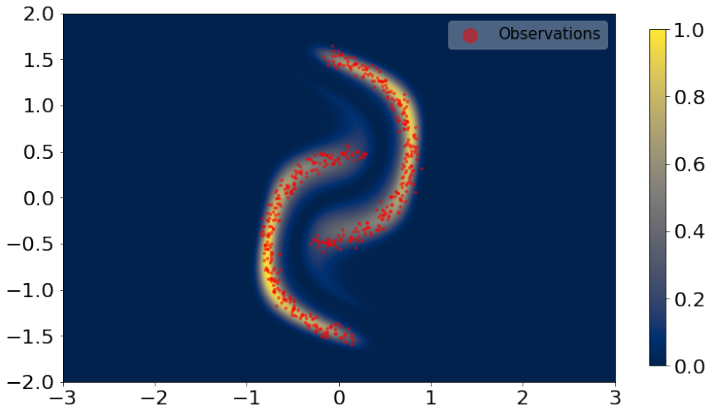


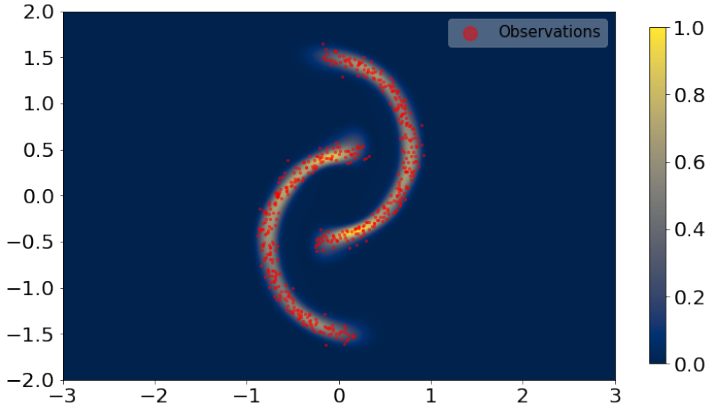
Figure 8.11: Here we can see how the conditional planar flow transforms the base distribution into the target distribution.



(a) 4 layers

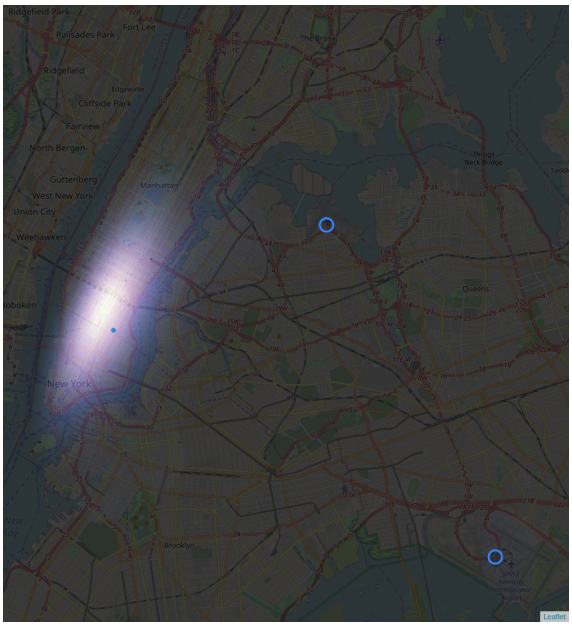


(b) 24 layers

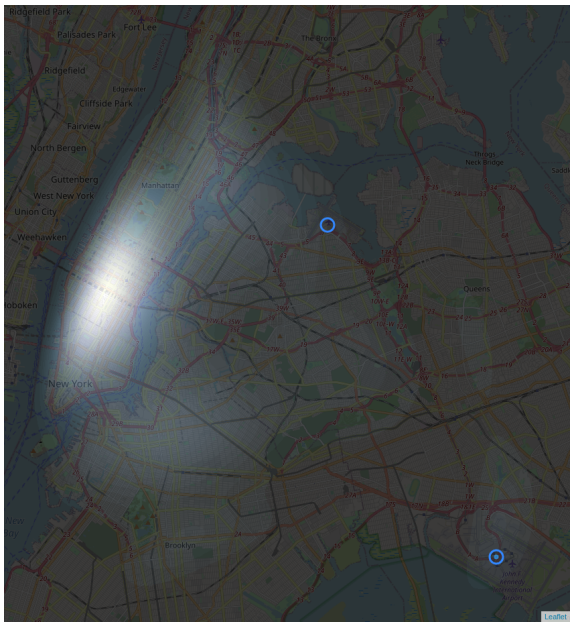


(c) 48 layers

Figure 8.12: Three planar flows fitted to the Two Moons synthetic data. The flows are only conditioned on rotation and all flows are conditioned on a rotation of 1.5 radians.



(a) Conditional Planar flow conditioned on 18:00 Tuesday with pickup in Manhattan



(b) Conditional Planar flow conditioned on 18:00 Tuesday with pickup at JFK

Figure 8.13: Conditional planar flow conditioned on two different pickup locations. Note the disjoint probabilities in b

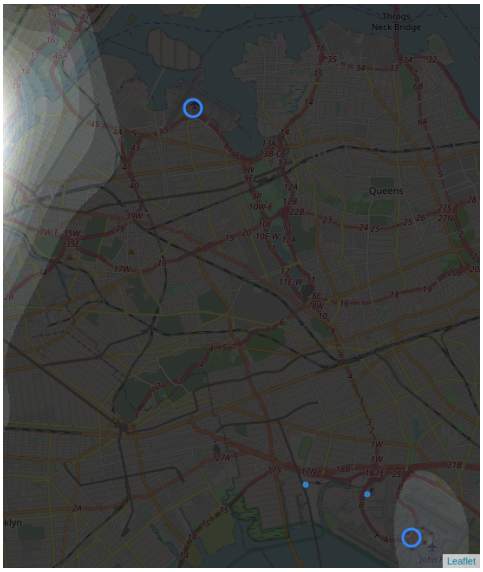


Figure 8.14: The area between Manhattan and JFK airport for a conditional planar flow conditioned on 18:00 Tuesday with a pickup on Manhattan.

structure of the flow is left as future work[4]. First, in order to test if the flow with both planar and affine coupling layers are able to create a disjoint distribution, we test it out on the two moons data set. As before, we try flows of different depths[5]. The result can be seen in Figure 8.16. The shallow flow of 4 layers still shows a probability path, but it is now much less dense, showing that the planar flow indeed did help with lessening the impact of the probability paths. It also has more defined edges than the pure planar flow in Figure 8.12a showing that the affine coupling layers still help with more defined edges. If we look at the deeper flows of 24, we can see that the two distributions are, in fact, disjoint, but we still have some problems with probability paths. These paths are, however, very thin and do not represent much probability mass. The same can be seen in the deep flow of 48 layers. Hence it seems like the combination of both planar layers and affine coupling layers are better at creating disjoint probabilities with defined edges. During training of the mixed normalizing flows, we noticed that these flows are very dependent on initialization, having completely different achieved log likelihoods for identical hyperparameters and training setups. We believe this is because the flow can get caught in local minima where the affine coupling layer tried doing transformations better suited for the planar layer and vice versa. This is not a problem in the flows based on a single type of layer, as all transformation can do the same. Due to this, the mixed normalizing flows were almost impossible to train without batchnormalization layers, even for very shallow flows.

Next, we try to fit the mixed flow to the NYC Yellow taxi data. As with the planar flow, we condition the flow with both pickup location and time variables and fit a 48 layer flow in order to compare with the best conditional affine coupling flow. We trained three flows, and the achieved log likelihoods on the test data for the mixed flows were between -1.62 and -1.60. The achieved log likelihood is better than the conditional planar flow but still not as good as the conditional affine coupling flow. The best model is visualized in Figure 8.17. As can be seen in Figure 8.17a, the mixed flow is able to create a disjoint probability around the JFK airport, just as the planar flow. However, for the mixed flow, the distribution is now complex enough to leave Central Park as a low probability zone just as the Conditional Affine Coupling flow, which the Conditional Planar flow could not do. In Figure 8.17b we can see that the conditional mixed flow still has probability paths for some contexts but as seen in Figure 8.18 the flow shows promise in having disjoint complex densities.

In this chapter, we have shown that conditional normalizing flows are able to successfully model geospatial data with conditional variables. The experiments with the conditional affine coupling flow showed that deeper flows achieve a better log likelihood, but even for the deepest flow of 48 layers, the model still showed limitations in that it is unable to generate disjoint probabilities due to the limitations of the affine coupling layer. The con-

[4]Perhaps doing multiple planar flows first could split the distribution into disjoint areas before the affine couplings massage in the finer details of the distribution.

[5]The depth of the mixed flow is measured in total transformations. As such, a mixed flow of depth four will have two planar layers and two affine coupling layers

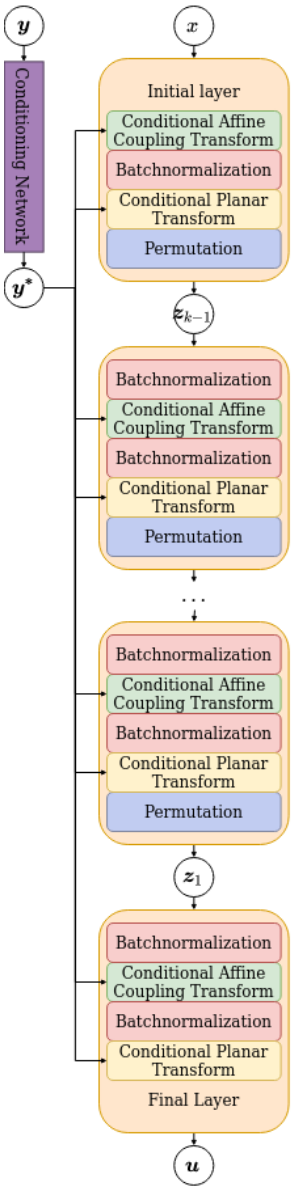
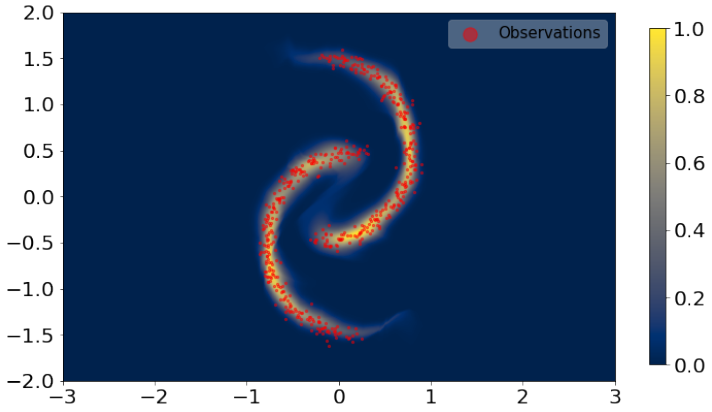
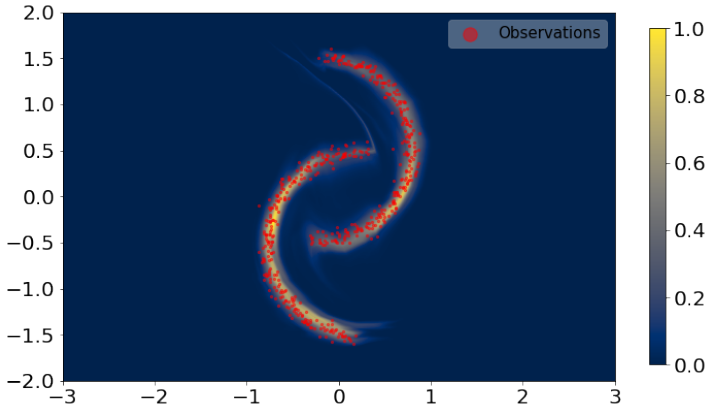


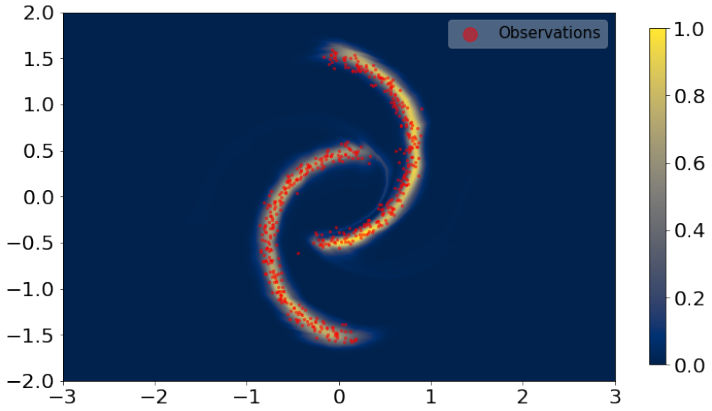
Figure 8.15: The architecture of a flow with both conditional affine coupling transformation layers and conditional planar transformation layers. The flow includes batchnorms to stabilize training and permutes to allow the conditional affine coupling transformation layers to affect all dimensions.



(a) 4 layers

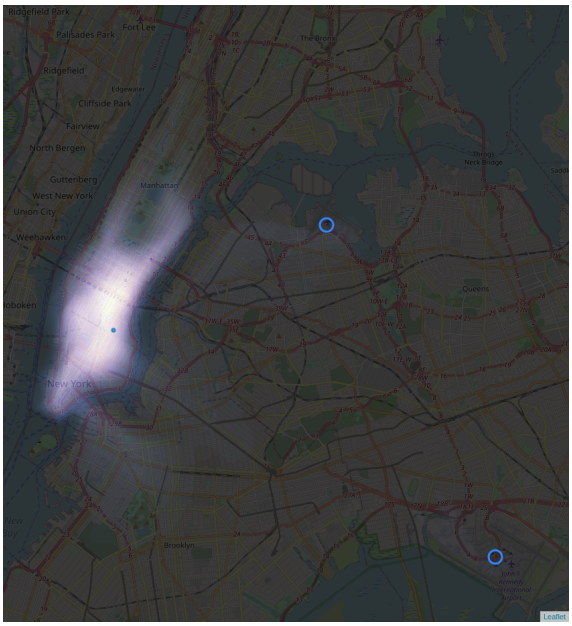


(b) 24 layers

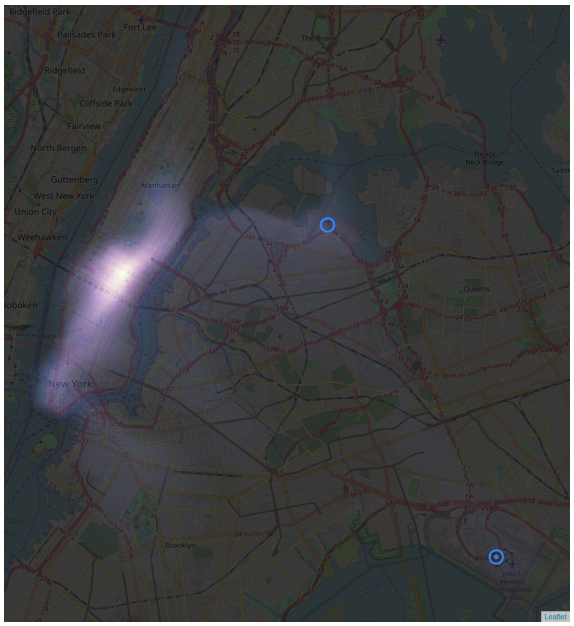


(c) 48 layers

Figure 8.16: Three flows with conditional affine coupling and planar transformations fitted to the Two Moons synthetic data. The flows are only conditioned on rotation and all flows are conditioned on a rotation of 1.5 radians.



(a) Conditional mixed flow conditioned on 18:00 Tuesday with pickup in Manhattan



(b) Conditional mixed flow conditioned on 18:00 Tuesday with pickup at JFK

Figure 8.17: Conditional mixed flow conditioned on two different pickup locations.

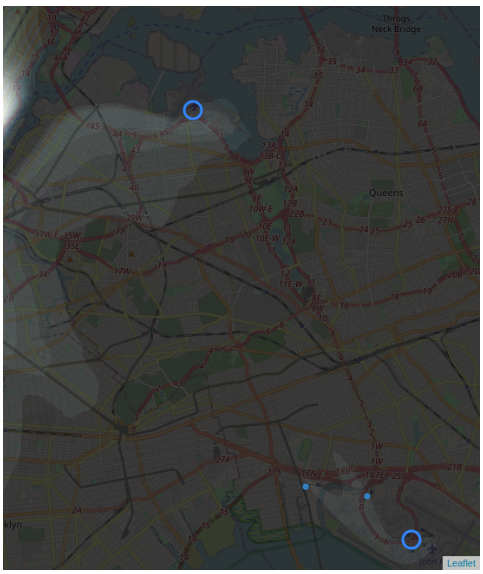


Figure 8.18: The area between Manhattan and JFK airport for a conditional mixed flow conditioned on 18:00 Tuesday with a pickup on Manhattan.

ditional planar flow was able to successfully model disjoint probabilities, but the limited expressiveness of the planar transformation led to a less complex distribution, unable to wrap the distribution around Central Park. By combining the two forms of layers into a mixed flow, we were able to model disjoint probabilities while still having a distribution complex enough to wrap around Central Park. However, for some context, the mixed flow still showed probability paths, and for flows of the same size, the conditional affine coupling flow still achieved the best log likelihood on the test data. This is probably due to the probability paths not having much of an impact on the log likelihood of test data, as the probability mass in these are small compared to the whole distribution and only affect the log likelihood if an observation in the test set is within the path. As such, for achieving a better log likelihood, a more complex distribution from more affine coupling layers seems more important than having disjoint probabilities from planar transformations. We do note that the results in this chapter have been focused on the qualitative performance of the flows. As such, we have not optimized the depth and number of parameters in the flows. Perhaps the ability of complex and disjoint densities of a mixed flow might lead to better performance for deeper, more optimized flows.

CHAPTER 9

Modelling Bike Rentals in Copenhagen

In this chapter, we will train the different normalizing flows on a novel dataset in order to use the models for a downstream task. The data is logs from a bike rental app in Copenhagen. The downstream task we wish to solve is placing bike pickup and dropoff locations in the most optimal locations based on what time it is and whether or not it is raining. In order to solve this task, we first train conditional normalizing flows to model the distribution of the app logs. Then, based on the models, we place hubs at the locations with the most probability. Based on the positions of the hubs, we will present another way of comparing the normalizing flows other than log likelihood. However, since it is a novel data set, we start with presenting the data. The data set is generated by combining two different data sets. The first is the bike rental logs from a company called Donkey Republic, and the second is data from DTU's weather station.

9.1 Bike rental data

Donkey Republic

Donkey Republic is a Copenhagen based company that via electronic locks and an app allows users to rent bikes situated around the city. As of June 2020, Donkey Republic rents out 16000 bikes in 71 cities all over Europe[1]. In Copenhagen, Donkey Republic is the main bike rental provider with 2075 out of 3200 bike rental permits[2]. Donkey Republic rents out these 2075 bikes via what they call the *hub-centric model*. This means that the start and end of a bike rental must happen within a *hub*, which is a circle with a radius of 10 meters around a predetermined GPS location[3]. In Copenhagen, there are 1147 different hubs spread around the city. In Figure 9.1, the active hubs are shown along with some hubs that have been removed since data collection began. The different hubs are placed around the city in locations based on different parameters:

[1] `https://www.donkey.bike/about/`

[2] `https://www.donkey.bike/donkey-republic-to-become-the-main-bike-sharing-provider-of-copenhagen/`

[3] `https://cities.donkey.bike/bike_share_hub-centric-model/`

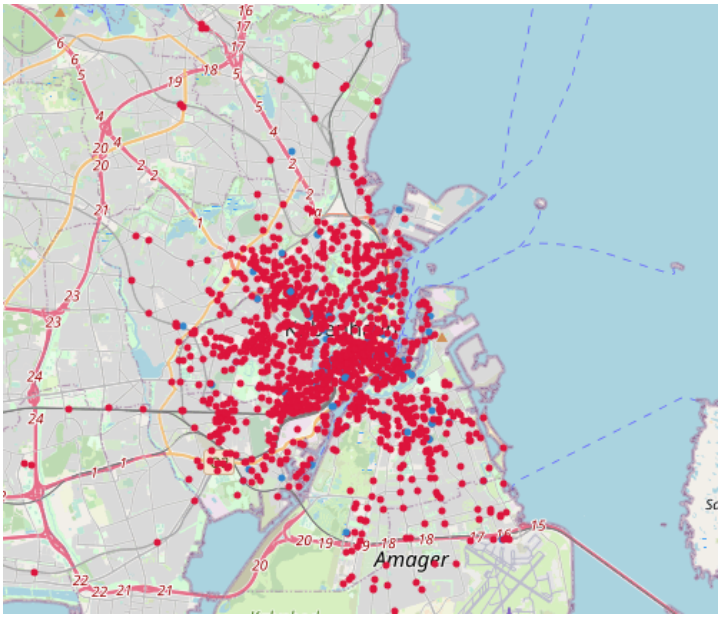


Figure 9.1: Overview of the different hubs in Copenhagen. The red dots are active hubs and the blue dots are removed hubs.

- impact on the accessibility of public space
- pressure on parking capacity
- connectivity to other transport services
- convenient spaces to access the bikes
- demand from users or the municipality

While most of these parameters are enforced by the municipality and practicalities of the available areas, the last is different and much more interesting to us. In order for Donkey Republic to be able to place hubs based on demand from users, they need a model for that demand. We will try to make this model using historical data and conditional normalizing flows. The data set given by Donkey Republic allows us to approximate the demand from users in different ways.

The data we will use comes in 3 different data sets. *Rentals* contains data about bike rentals in Copenhagen, *Hubs* contains data about hubs in Copenhagen and *Searchlogs* contains data about app usage in Copenhagen. See Table 9.1, 9.2 and 9.3 for more detailed description. In order to place the hubs, we wish to model the demand of the Donkey Republic's users. The *Rentals* data corresponds to the direct usage of the users, but we will not use this data for two reasons. The first reason is that the measured rentals might not correspond to the demand of the users. Since users can only rent bikes that are available at the time, the data might be subject to censoring. If ten users wish to rent bikes from a hub that only has nine bikes available, there will be a censoring problem. The demand is clearly ten bikes, but the rental data will only show a usage

Rentals

Data Column	Description
created_at	Timestamp of the start of the rental
finished_at	Timestamp of the end of the rental
pickup_hup_id	Id of the hub where the rental started (See Hubs)
dropoff_hub_id	Id of the hub where the rental ended (See Hubs)
user_id	Anonymous Id of the user

Table 9.1: Data description of the Rentals part of the data

Hubs

Data Column	Description
created_at	Timestamp of when the hub was created
latitude	Latitude of the hub
longitude	Longitude of the hub
id	Id of the hub
name	Name of the hub
deleted_at	Timestamp of when the hub was deleted. NaN if the hub has not been deleted.

Table 9.2: Data description of the Hubs part of the data

Searchlogs

Data Column	Description
user_location_latitude	Latitude of the user
user_location_longitude	Longitude of the user
anonymous_id	Anonymous Id string of the user
user_id	Anonymous Id integer of the user
timestamp	Timestamp of the searchlog

Table 9.3: Data description of the Searhlogs part of the data

of nine bikes. There are modeling approaches that can be taken to mitigate the effect of censoring such that the demand can be approximated from the usage. However, we will not go into that here. The second reason is that we wish to optimize the locations of the hubs, so basing a model on data from the current location of the hubs does not give us the information we need. Instead, we use the Searchlog data, which solves both of the problems. Assuming that users only open the app when they wish to rent a bike, this data corresponds to the actual demand. This assumption might, of course, not be true as users could open the app for other reasons like changing subscription, modifying profile, etc. Furthermore, the searchlogs are situated where the users are, not where the hubs are. Therefore the searchlogs are distributed around Copenhagen where the demand is, not where the rentals are. However, since users need the app to unlock the bike at the hub, the searchlogs are skewed towards the hubs. Still, the searchlogs are distributed in a complex manner throughout the entirety of Copenhagen, where the rental data's distribution is very "spiky" in the hub locations. This behaviour can be seen in Figure 9.2 and 9.3 where the Rental and Searchlog data is showed as a heat map around Nørreport station. Looking at Figure 9.2 and 9.3, we can clearly see that the Rental data is lumped together at the hubs while the Searchlog data is much more spread out. The Searchlog data does have hot spots around hubs, like at the junction of Gothergade and Linnedsgade. However there are also areas with searchlogs without hubs, like along Nørre Farimagsgade. This raises the logical question: If there is demand at Nørre Farimagsgade, why not place a hub there?
The variables in the searchlog data are mostly ready to use. However, in order to use

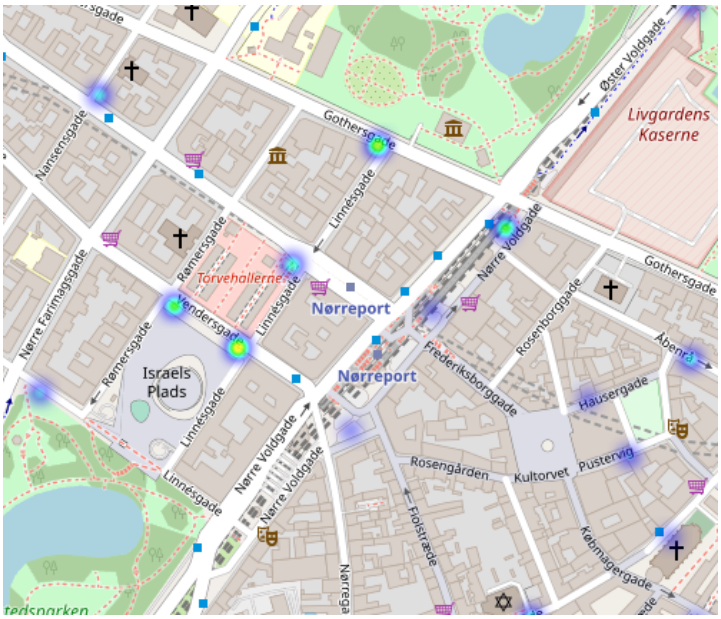


Figure 9.2: Heat map of bike rentals around Nørreport station in Copenhagen. (Small amount of noise was added to coordinates in order to allow heat map implementation to color hubs corresponding to amount of rentals)

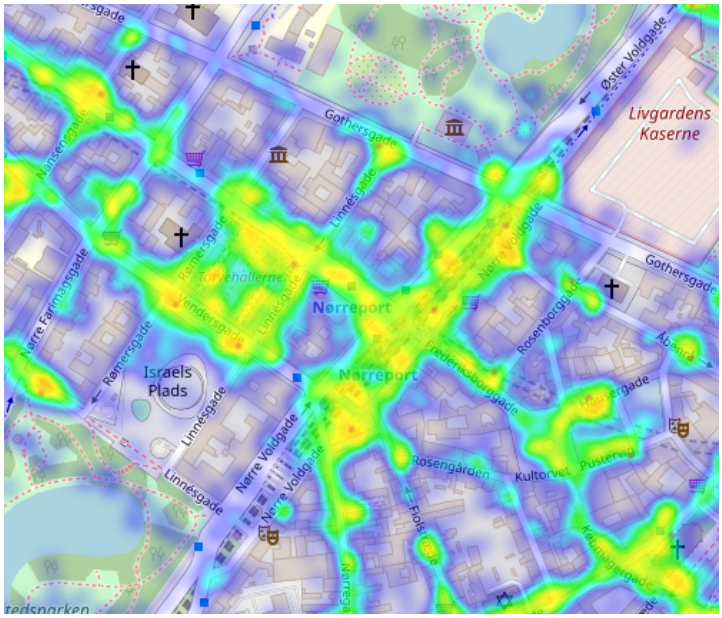


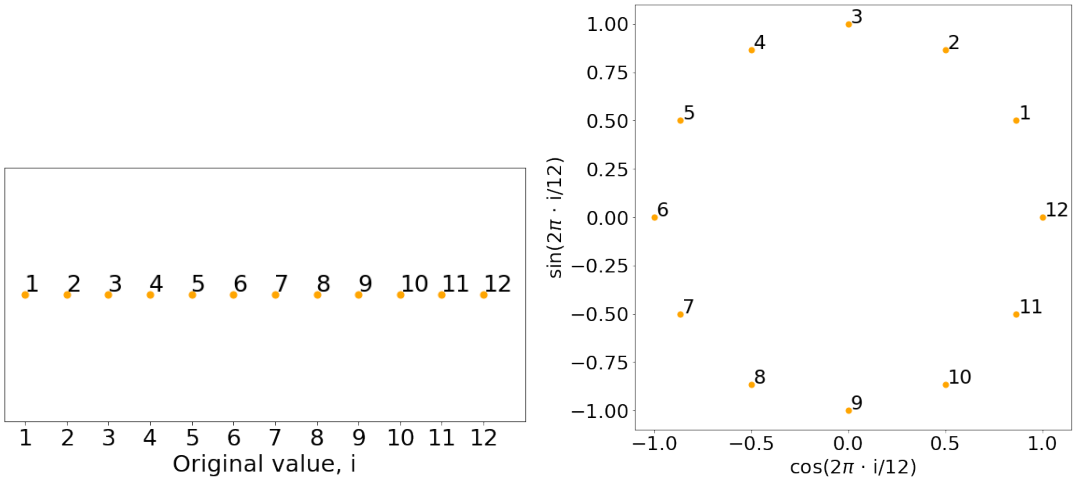
Figure 9.3: Heat map of the searchlogs around Nørreport station in Copenhagen

the timestamp as an input to the model, we split it up into month, weekday, and hour. The simplest way would be to denote the months, days, and hours by integers, i.e., month would be an integer between 1 and 12. However, this would mean that the integer values for January and December, 1 and 12, would be as far apart as possible even though January is right after December. In order to fix this, we use the same preprocessing as presented in [Dut+18]. The fix is to represent the integer value, i, by the point $\left(\cos(\frac{2\pi i}{m}), \sin(\frac{2\pi i}{m})\right)$ where m is the max value i can have. We can think of this as a circle transformation as integers from the number line is transformed to points in a circle. This can be seen in Figure 9.4.

DTU Weather Station

As mentioned earlier, we wish to model the bike rentals using conditional normalizing flows, so we need some covariate data we can condition the model on. This data has to be something that has an influence on bike rentals. From personal experience, one of the biggest influences on the attractiveness of a bike ride is the weather. Therefore we use weather data acquired from DTU Byg's weather station[4]. The station is located on Building 119 at the DTU Lyngby Campus. It is worth noting that this is around 11 kilometers north of Copenhagen city center. Hence the weather measured here does most likely not match the weather in the entirety of Copenhagen.
The data is acquired using a modern weather station that can measure multiple variables of the weather. See [And+14] and [And+17] for details. The different weather variables measured can be seen in Table A.1 in Appendix A.2. Looking at the data, we can

[4] `http://climatestationdata.byg.dtu.dk/`



(a) (b)

Figure 9.4: A plot of how the circle transformation works. In (a) we see original values which is just integer values from 1 to 12. In (b) we can see how the transformed values $\left(\cos(\frac{2\pi i}{12}), \sin(\frac{2\pi i}{12})\right)$ correspond to points going counter clockwise around a circle.

see that the data from the weather station is erroneous for most of April 2018 (02-05-2018 to 23-05-2018) and for most of August, September, and October 2018 (2018-08-12 to 2018-11-06). For a specific analysis of how the data is erroneous, see the GitHub implementation[5].
Since our goal is to create a model of the Donkey Republic data conditioned upon the weather data, we categorize the weather data such that we have more observations with each value of the weather data. If we did not do this we could run into a problem of not having enough data to train and validate our model, i.e., we might only have a 100 observations of bike rentals at 20:00 on a $20°C$ Saturday with $10mm$ of rain accumulation with a wind of $10m/s$ however we might have 4000 observations of bike rentals at 20:00 on a warm Saturday with rain and high winds. However, we assume that the fact that it rains, is warm or is windy has more influence on bike rentals than the exact meteorological measurements. Therefore the weather station data is aggregated into a simplified version. The new data variables and values can be seen in table 9.4.

Created dataset

We merge together the data from Donkey republic together with the data from the DTU weather station in order to get a data set with both dependent and independent variables. In the created data set, we have the searchlog coordinates as dependent variables while the cosine-sine transformation of the hour and whether or not it is raining is the independent variables. As mentioned above, the data sets contain more variables we could use as independent variables such as wind and the day of the week. However,

[5] `https://github.com/MathiasNT/Thesis_Density_Estimation_w_Normalizing_Flows`

Name	Values	Description
Time(utc)		Time given in utc. The data is given for each hour within the good areas of the data.
wind_dir_sin wind_dir_cos	Real number between -1 and 1 Real number between -1 and 1	The wind direction is first simplified to north, east, south and west and afterwards the value is circle transformed.
windy	True, False	True if it is blowing more than 3.4 m/s. False otherwise. 3.4 m/s corresponds to the cutoff between 2 and 3 on Beaufort's Scale.
air_temp	1, 2, 3	Integer value that corresponds to it being: 1: below 10° degrees 2: between 10 and 20 degrees 3: over 20 degrees
rain	True, False	Whether or not it raining, characterised by any measured accumulation at all.

Table 9.4: The different variables in the simplified weather data

experiments showed that hour and rain are the variables with the largest influence on the bike rentals. So due to time constraints, we will focus on these.
The created data set is split in two: a part with all variables and a part where there is no rain data due to the erroneous data from the weather station. However, as explained in Section 5.3, we can fit a conditional normalizing flow to the data with a semi-conditional setting.

Before we fit conditional normalizing flows to the data, we first wish to check if the independent variables can be used to condition the dependent variables. To do this, we fit KCDEs and a KDE to a subset of the data. We extract a week from the data set, split it in a test and training set and use *statsmodels* ([SP10]) implementation of KCDE and KDE[6] to fit an unconditional KDE, a KCDE conditioned on hour and a KCDE conditioned on hour and rain. All of the models were fitted using cross-validation on the bandwidth. The resulting models have the following log likelihoods on the test data:

- Unconditional KDE: -1.138
- KCDE conditioned on hour: -0.996
- KCDE conditioned on hour and rain: -0.992

As can be seen from the log likelihoods, conditioning on the hour and rain leads to a higher log likelihood. Hence the hour and rain have an influence on bike rentals, and it is possible to use the hour and rain to condition the searchlog coordinates. The different kernel methods were only fitted to a subset of the data as the implementation of kernel methods in *statsmodels* scales poorly with data size, making fitting a model on all of the data unfeasible. A discussion of this can be seen in Appendix A.3. As discussed

[6]These implementations follows the theory presented in Section 2

in the appendix *sklearn* ([Ped+11]) has an implementation of KDE that scales better with data size. In order to get a base model to compare the different normalizing flows against, we fit a KDE to the full data. The model achieves a log likelihood of -1.671.

9.2 Conditional Normalizing Flows on Donkey Republic data

Now we move on to fitting conditional normalizing flows to the Donkey Republic data. We will fit three different flows to the data: one based on conditional affine couplings, one based on planar transformations, and one flow with a mix. All of the flows will have the same depth and a similar number of parameters in order to be able to compare the different architectures. The hyperparameters of the flows can be seen in Table 9.5. All flows have been trained with noise regularization with a standard deviation of 0.02, and the data have been split 80/20 into train and test data.
The depth of the flows has been set as high as possible while still allowing a large enough batch size for the flows to trained in around 4.5 hours in order to allow iterations of the hyperparameters. Note that the use of the semi-conditioned loss function slows down the training considerably as mentioned in Section 5.3.
The achieved log likelihoods of the models can be seen in Table 9.6. As can be seen in the table, the conditional planar flow achieves a log likelihood much worse than the other flows. The conditional affine coupling flow and the conditional mixed flow perform better with the conditional affine coupling flow performing slightly better than the conditional mixed flow. It is also worth noting that all of the conditional normalizing flow models all achieve log likelihoods much worse than the unconditional KDE model.

Flow model	Number of layers	Conditioner depth	Conditioner hidden dims	Context network depth	Context network hidden dims
Conditional affine coupling	24	5	24	5	24
Conditional planar	24	5	24	5	24
Conditional mixed	24	5	24	5	24

Table 9.5: The hyperparameters for the architecture of the different conditional normalizing flow models. They are all identical to allow for comparison between models.

Model	Log likelihood
Affine Coupling	-2.17
Planar	-2.63
Mixed	-2.25

Table 9.6: The achieved log likelihood on the test data for the different conditional normalizing flows.

As discussed in the last chapter, the performance of the conditional normalizing flow models could be improved by training deeper flows, but this still shows how strong the performance of non-parametric approaches can be. However, the normalizing flows have the clear advantage of much faster density estimation once trained. Next, we wish to solve the downstream task of placing hubs around Copenhagen based on the different models and use the placed hubs to compare the different models. As this task is based on density estimation, the KDE model is terribly slow at this. Therefore we will focus on the conditional normalizing flow models.

We place the hubs around Copenhagen based on the distributions conditioned on some context, e.g., at 17:00 on a rainy day. In order to do this, we are going to make an approximation. The exact way to do it would be to find the locations with the most probability given the model and placing a hub there. However, this is an expensive optimization task and also raises the question of how spread out the hubs should be. Instead, we are covering Copenhagen in a 200 by 200 grid, creating 40000 different locations in which to place a hub. This grid corresponds roughly to having a grid point at each street. Then in order to place the different hubs, we rank the probability densities at the different grid points given a model and place hubs based on this ranking.

Doing this also allows us to create a metric for how well the model places the hubs. The goal of the model is to optimize the placement of the Donkey Republic hubs such that the demand of the users is met as best possible. So we want to score our models against how well they achieve this goal. One way to do this is to look at the *Cumulative Accuracy Profile* (CAP) of the model. The CAP looks at the cumulative score of some accuracy metric of the model against some classifying parameter. This method is often used in economics to compare rating models, see e.g., [EHT03]. In our case, we use the models to cover Copenhagen in hubs, placing the hubs at the spots the models deems most likely for searchlogs to appear. Then after that, we will score how many searchlogs from the test set the placed hubs cumulatively can cover against how much of Copenhagen is covered by hubs. We consider a searchlog covered by the hub if it is within the grid cell in which the hub was placed. Doing this, we can generate a CAP curve by plotting how big a percentage of Copenhagen is covered versus how big a percentage of the searchlogs in the test set are covered. Since the trained models are conditional models, we are going to filter the test data such that we only consider searchlogs in the test data that fit the context when calculating the CAP curve. In Figure 9.5, the CAP is depicted for the three different flows for a context of 08:00 on a day without rain. Along with the models, we have also computed the CAP of three different baselines. The first baseline is the CAP of the hubs Donkey Republic already has in place. The way we computed this CAP is by first ranking the hubs based on the number of rentals the hubs have in the rentals data set, then assigning each hub to the nearest of 40000 grid points and computing the CAP as for the flow models. If any of the hubs are assigned to the same grid point, only the first hub is used. The second baseline CAP is from a random model that ranks the 40000 grid points randomly and computes the CAP based on this ranking. The last baseline CAP is from a perfect model that ranks the

40000 grid points based on the covered search logs and calculates the CAP from this ranking. As can be seen, all of the flow models are much better than the random model, so at least placing the models based on any of the flow models is better than placing them randomly. Furthermore, we can see that both the conditional flow based on conditional affine couplings and the flow based on a mix of transformations are better than the hubs Donkey Republic has while the CAP of the conditional planar flow is worse. However, the conditional affine coupling flow and the mixed flow have very similar CAP curves. In order to better compare them, we make a Riemann sum approximation of the area under the curve of the different CAP curves. The results can be seen in Table 9.7. Based on the AUC of the different flows, we can see that based on the full CAP curve, the difference between the conditional affine coupling flow and the conditional mixed flow is negligible. However, if we look at the AUC of only the first 1147 grid points[7] we can see that, in fact, the conditional mixed model is slightly better. This is contrary to what we got when comparing the log likelihoods of the two flows. Hence even though the conditional affine coupling flow is slightly better at approximating the distribution, the conditional mixed flow is better at the downstream task of placing hubs.

Next, we condition the models on 08:00 with rain and compare the models. The CAP curves can be seen in Figure 9.6 and the AUC's can be seen in Table 9.8. As can be seen from the figure and the table with this context, we get some quite different results.

[7]Donkey Republic have 1147 hubs in Copenhagen

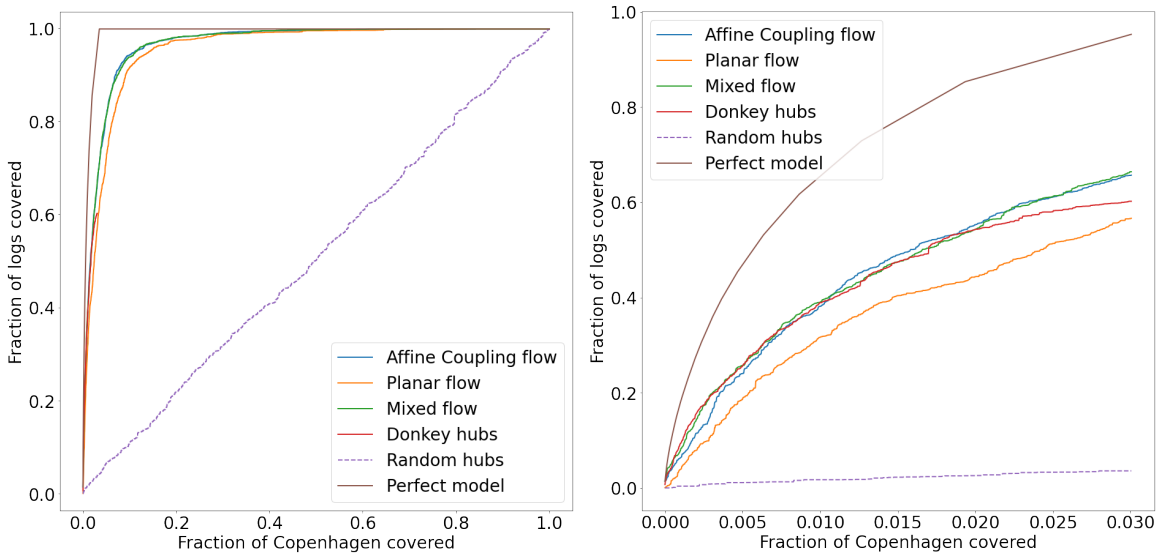
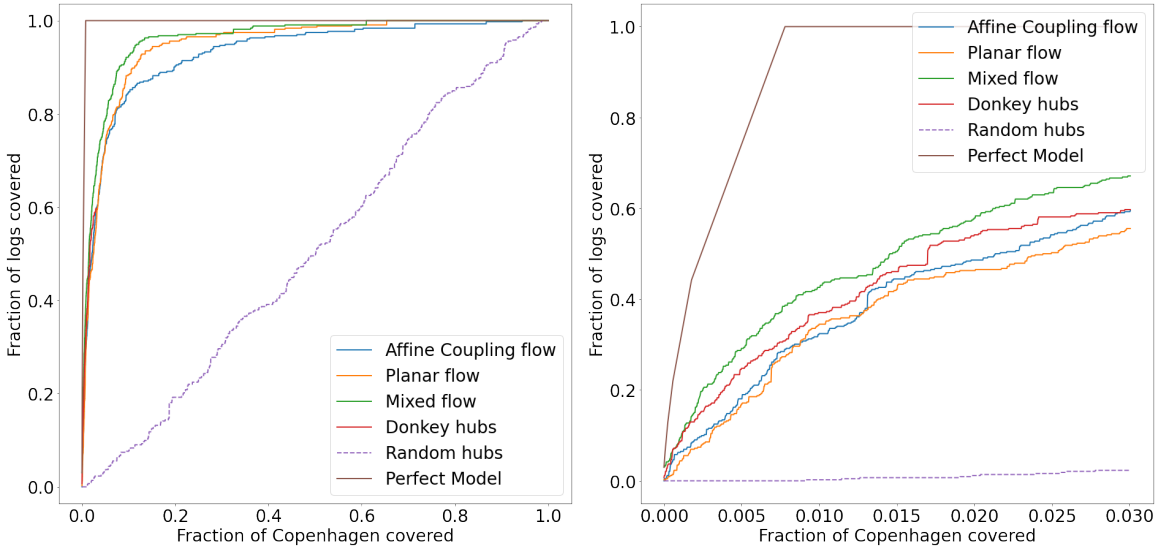


Figure 9.5: The computed CAP of the different three different kinds of conditional flows along with the CAP of the hubs, a perfect model and a random model. The test data has been filtered for 08:00 without rain and the flows have been conditioned upon this context.

Model	AUC for full CAP	AUC for first 1147 hubs
Random	0.493	0.0003834
Perfect	0.989	0.02115
Donkey Republic	-	0.01297
Conditional Affine Coupling Flow	0.9657	0.01325
Conditional Planar Flow	0.9556	0.01080
Conditional Mixed Flow	0.9657	0.01332

Table 9.7: The area under the CAP curves for the different models on a context of 08:00 and no rain.



(a) (b)

Figure 9.6: The computed CAP of the different three different kinds of conditional flows along with the CAP of the hubs, a perfect model and a random model. The test data has been filtered for 08:00 with rain and the flows have been conditioned upon this context.

Model	AUC for full CAP	AUC for first 1147 hubs
Random	0.5130	0.0006137
Perfect	0.9970	0.0270815
Donkey Republic	-	0.01263
Conditional Affine Coupling Flow	0.9318	0.01146
Conditional Planar Flow	0.9486	0.01090
Conditional Mixed Flow	0.9617	0.01413

Table 9.8: The area under the CAP curves for the different models on a context of 08:00 and rain.

Now both the conditional planar flow and the conditional affine coupling flow are worse than the Donkey Republic hubs. In fact, when looking at the complete CAP curve, the planar flow performs better than the affine coupling flow. The conditional mixed flow is still the best flow, and it still outperforms the Donkey Republic hubs.

As such, we can conclude that it is, in fact, possible to use normalizing flows for the downstream task of optimizing hub locations. Furthermore, we conclude that even though the conditional affine coupling flow achieves a better log likelihood on test data, the conditional mixed flow achieves a better AUC on the CAP curves and, as such, is the better choice for placing the hubs.

CHAPTER 10

Conclusion

In this thesis, we have presented the theory behind normalizing flows. We have shown how the intuitive idea of transforming a simple distribution through a of series simple, invertible transformations allows us to approximate complex, highly non-gaussian distributions. We have described how the different operations of the flow can be used to train the flow in different settings, allowing normalizing flows to be used for a variety of different tasks. We presented the theory behind finite normalizing flows focusing on affine coupling flows and planar flows and how these flows can be extended to conditional versions through simple extensions on each layer of the flow. We also presented how the simple idea of adding noise to observations during training could be used for noise regularization of flows with nice theoretical interpretation. With experiments on synthetic data, we showed how conditional normalizing flows are able to successfully rotate a modeled distribution based on a conditioning variable. Furthermore, we experimented with different regularization methods showing that with the correct hyperparameters, noise regularization was able to successfully regularize normalizing flows. Other methods like dropout and $L2$-regularization also showed promise, but the interpretability and performance of the noise regularization showed it as a clear favorite.

The experiments on the NYC Yellow Taxi data showed that conditional affine coupling flows are able to model the low-dimensional problem of geospatial demand prediction, even though they were developed for the high-dimensional problem of image data. However, the conditional affine coupling flows were shown to have a clear problem with creating disjoint probabilities in the modeled distribution. The autoregressive structure of the couplings led to clear paths of non-zero probability density that deeper flows were not able to remove even though they were stretched thinner. Conditional planar flows were shown not to have this problem but instead had trouble creating complex edges in the modeled distribution. By mixing both kinds of transformations together in a single flow, we were successfully able to model complex edges and disjoint areas in the modeled distribution with a single flow of the same depth. However, the mixed flows showed vulnerability to initialization and local minima only partly removed with the use of batchnormalization layers. We note that the structure of the mixed flow was not optimized and see it as an interesting possibility for future work to look into how different orderings of transformations impact the performance of the flow.

From the experiments on the NYC Yellow Taxi data, we also saw how flows that achieved similar log likelihoods on held-out data, modeled very different distributions. This was

especially prevalent in contexts with low amounts of data and for the probability paths in conditional affine coupling flows. As such we identified that selecting models based on achieved log likelihood on a test set does not take all aspects of the model into account. We also showed how conditional normalizing flows could be used for the downstream task of optimizing the placement of bike rental hubs in Copenhagen. We trained conditional normalizing flows on searchlog data conditioned on time of day and whether or not it was raining. By placing hubs based on fitted mixed flows, we were able to outperform Donkey Republics' hub placement. The conditional planar flows were unable to outperform Donkey Republics' placement of hubs. We conjecture this is because the conditional planar flow has a hard time modeling distributions with complex edges, as was also evident on the NYC Yellow Taxi data. For the conditional affine coupling flow, we saw that while the flow was able to outperform the Donkey Republic hubs when conditioned on no rain, the flow was not able to do so when conditioned on rain. This is probably due to the much lower amount of data with rain. As we saw on the NYC Yellow taxi data set, the shape of the modeled distribution, when conditioned upon uncommon contexts, is very irregular even between identical models. However, for the mixed conditional flow, we saw that the flow was able to beat the Donkey Republic hubs consistently across different contexts.

When comparing the three different proposed flows, it is clear that flows based on affine couplings and planar transformations have different trade-offs. While the affine couplings layers allowed for very complex shapes to be modeled, the autoregressive structure of the coupling layer means that the flow is unable to create disjoint distributions. The planar transformations were able to create disjoint distributions but had a much harder time modeling complex edges of distributions. By mixing the two kinds of layers together to a single flow, we could, at least qualitatively, show that it was possible to get the best of both worlds - complex edges and disjoint distributions. However, the conditional mixed flow was unable to beat the conditional affine coupling flow in log likelihood on test data. This could be because well-defined edges of the distribution have more influence on the achieved log likelihoods than probability paths have. By switching some affine coupling layers for planar transformation layers, the mixed flow does have a harder time fitting edges than the affine coupling flow. It could be interesting future work to see if this is also true for even deeper flows. After a certain depth, it is possible that the complexity of the affine coupling flow and the mixed flow are similar. At this point, we would expect the mixed flow's ability to remove probability paths to mean a better achieved log likelihood.

Another point worth mentioning when comparing the different flows is the fact that the affine coupling flow is fast both in the generative and normalizing direction. As such, a trained affine coupling flow can be used both for density estimation and sampling. The planar transformation has no fast inverse, and therefore, both the planar flow and mixed flow can only be used for density estimation when trained with MLE. As the performance of the affine coupling flow is comparable to the mixed flow, even in low dimensions as here, we conclude that if both density estimation and sampling is required by a trained flow, affine coupling flows are the obvious choice.

Through our work, we also got multiple ideas that could lead to interesting future work. One of the big drawbacks of the affine coupling layers was the limited expressiveness of the affine transformer. As such, it would be interesting to see how the affine coupling flow would perform if different transformers were used instead. It has recently been proposed using transformers based on splines instead of affine transformations, which has been shown to increase the expressiveness of affine coupling layers while retaining fast inverses. See [Dur+19a] and [Dur+19b]. Another possibility could be to extend the affine coupling layer to also have a transformation of the first half of the dimensions. This transformation is based on learnable parameters that are the same for all inputs to the layer. This transformation will be much less expressive than the transformation on the other half but might help with creating disjoint densities. It could also be interesting to see if the affine coupling flows inability to create disjoint distributions could be remedied by changing the flow's base distribution. As long as the distribution is a well-defined distribution, there are no restrictions on it. As such, creating a flow with a disjoint base distribution could potentially mean that the flow itself does not need to split the distribution. However, as density evaluation of the base distribution is used during training the chosen base distribution would need a fast way to do this. This thesis has only considered planar and affine coupling layers. It could be interesting to extend it to both radial layers [RM15] and Sylvester layers [Ber+18], both of which are residual layers.

The findings of Chapter 8 have been submitted as a paper to the 2020 ICML Workshop on Invertible Neural Networks, Normalizing Flows, and Explicit Likelihood Models[1]. The paper can be seen in Appendix B

[1] https://invertibleworkshop.github.io/

APPENDIX A

Appendices

A.1 Implementation details

Our implementation of the conditional normalizing flows are based upon Pyro's[1] implementations. Pyro is a probabilistic programming languate (PPL) build on top of Python and PyTorch. Pyro's main focus is to create Bayesian models and doing Bayesian inference. In this thesis we will train normalizing flows via stochastic gradient descent on the exact likelihood, hence we will not need most of the tools in Pyro. However Pyro has an implementation of Affine Coupling flows we can use as a base for our implementation of the Conditional Affine Coupling Flow. Our implementation can be seen at `https://github.com/MathiasNT/Thesis_Density_Estimation_w_Normalizing_Flows`. As of April 7 2020 Pyro 1.3.1 has it's own implementation of the Conditional Affine Coupling. As we implemented our version before and there's a negligible difference this thesis is done using our own implementation. Pyro also has an implementation of a Conditional Planar Flow. However the Conditional Planar Flow is only implemented in the generating direction. As we will train the flow with MLE we have implemented an inverted version of the flow such that the normalizing direction is well implemented. We note that this implementation does not allow for sampling from the trained model, however a sampling method can be implemented using the methods described in Section 4.15.

Training of most flows was done on DTU cluster[2] on Nvidia Tesla V100 32GB GPU's. All training on DTU's cluster must be done via bash scripts. As such all training is set up such that all setups with bash and a GPU can just run the bash scripts to replicate results. However do note that the scripts are setup to utilize the V100's 32GB of memory.

All visualization is done by loading the trained model into GPU enabled Google Collab notebooks.

Any problems or questions with the GitHub can be directed to `s153583@student.dtu.dk`.

[1] https://pyro.ai/

[2] `https://www.hpc.dtu.dk/?page_id=2520` and `http://www.cc.dtu.dk/?page_id=2129`

A.2 Full weather data description

	Full name	Range	Resolution	Accuracy
GHI	Global shortwave solar irradiance		0.1 W/m^2	1.4 %
DHI	Short wave horizontal solar irradiance		0.1 W/m^2	1.4 %
DNI	Short wave normal solar irradiance		0.1 W/m^2	1 %
LWD	Downwelling horizontal longwave irradiance		0.1 W/m^2	<1%
wind_dir_min	Wind direction minimum [°]	0 - 360°(The direction from where the wind blows. 0°=North, 90°=East, 180°=South, 270°=Vest)	1°	3°
wind_dir_avg	Wind direction average [°]			
wind_dir_max	Wind direction maximum [°]			
wind_speed_min	Wind speed minimum [m/s]	0 – 60 m/s	0.1 m/s	The greater of 0.3 m/s or 3% of the measurement range 0…35 m/s 5% for the measurement range of 36…60 m/s
wind_speed_avg	Wind speed average [m/s]			
wind_speed_max	Wind speed maximum [m/s]			
air_temperature	Air temperature [°C]	-52 – 60 °C	0.1 °C	0.2 K at ambient temperature of -50 °C – 0 °C 0.3 K at ambient temperature of 20 °C. 0.4 K at ambient temperature of 40 °C 0.7 K at ambient temperature of 60 °C
relatvie_humidity	Relative humidity [%]	0 – 100 %RH	0.1 %RH	3 %RH at 0…90 %RH 5 %RH at 90…100 %RH
air_pressure	Air pressure [hPa]	600 – 1100 hPa	0.1 hPa	1 hPa at ambient temperature range -52…+60 °C
rain_accumulation	Rain fall [mm]	Collecting area: 60 cm2	0.01 mm	Cumulative accumulation after latest auto reset. <5 %, weather dependent. Due to the nature of the phenomenon, deviations caused by spatial variations may exist in precipitation readings, especially in short time scale. The accuracy specification does not include possible wind induced error.
rain_duration	Rain duration [s]		10 s	Counting each 10-second increment whenever droplets detected.
rain_intensity	Rain intensity [mm/h]	0 – 200 mm/h		Running one minute average in 10-second steps.½
hail_accumulation	Hail [hits/cm2]		0.1 hits/cm2	Cumulative amount of hits against collecting surface.
hail_duration	Hail duration [s]		10 s	Counting each 10-second increment whenever hailstone detected.
hail_intensity	Hail intensity [hits/m2/h]		10 hits/cm2/h	One-minute running average in 10-second steps.

Table A.1: All of the different meteorological measurements that the DTU Byg weather station can measure. Source http://climatestationdata.byg.dtu.dk/ClimateStation

A.3 Limitation of KCDE

As mentioned in Section 2 a KDE model is determined by the data set, kernel and bandwidth. Therefore there is no cost to fitting the model. However a naïve implementation of KDE has to go through points in the data set in order to compute the probability at a query point. This gives a time complexity of scoring against a test set of $\mathcal{O}(N_Q N_T)$ where N_Q is the number of query points in the test set and N_T is the number of observations in the training set. This can become a problem when determining the bandwidth for large data sets. Even though there exists multiple statistical rules for determining the optimal bandwidth like *Silverman's rule* [Sil86, pp. 45] they come with assumptions on the data that might not hold. Therefore the best approach to select the bandwidth is grid search with cross validation[3]. For cross validation the computational cost of naïve KDE can really be felt making fitting a KDE to a large data set intractable. We tried using *statsmodels* implementation on a dataset with 400000 observations but the cross validation was unable to finish within the 48 hours we had available of compute time.
In order to make cross validation tractable we have tried an approximating KDE method from [GM03] that uses kd-trees to speed up the computation of probabilities of query points. We will not go into detail here but the main idea is to build a tree structure of the data that can determine the distance between the query point and points in the data set. The tree is then used to determine if the contributions of data observations in a sub tree can be exchanged with a scaled mean contribution without introducing more than a predetermined relative error. That way the amount of kernels that have to be computed can be pruned considerably speeding up computation. In [GM03] they show how complexity goes from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$ for the case where $N_Q = N_T$. In fact they conjecture that it can go as low as $\mathcal{O}(N)$ for a dual tree approach where a second kd-tree is constructed with the query points. However the implementation we uses is *sklearn*'s *KernelDensity* which only seems to have implemented a single tree approach. We note however this is undocumented and is based on our look through the code. Using this approach we was able to train the KDE model presented in Chapter 9. However no implementaion of KCDE using KD-trees are readily available. It is possible to modify *sklearn*'s implementation to use the Nadarya-Watson conditional density estimator as presented in Section 2.2 however due to time constraints this is left as future work. Another interesting avenue could be implementing a proper dual tree approach to KCDE as described in [HGI07].

[3]We also tried out Bayesian optimization but grid search performed better.

APPENDIX B

ICML INNF+ 2020 submission

In this appendix is the paper submitted to the ICML Workshop on Invertible Neural Networks, Normalizing Flows, and Explicit Likelihood Models. The paper summarizes the theory of Chapters 2-6 and the results from Chapter 8. The full paper in PDF format can be found in the GitHub.

Strengths and Limitations of Normalizing Flows in Geospatial Data Modeling

Mathias Niemann Tygesen [1] **Daniele Gammelli** [1] **Sergio Garrido** [1] **Filipe Rodrigues** [1]

Abstract

Geospatial density estimation problems often exhibit complex and highly multi-modal structures. In this work, we explore the use of Normalizing Flows (NFs) in such geospatial settings. NFs offer a principled approach to represent rich and flexible distributions and in this work we experiment with conditional affine coupling layers and planar transformations layers, showing their strengths and highlighting their limitations. We also experiment with a proposed mixed flow showing how using both kind of layers can increase the flexibility of the flow.

1. Introduction

Much of the research in Normalizing Flows (NF) focuses on creating new and expressive flows for a high-dimensional setting. However many real-world problems such as geospatial density estimation are not high-dimensional. As such we are interested in analyzing how the NFs developed for high-dimensional problems work in lower dimensional settings.

We experiment with NFs based on conditional affine couplings and conditional planar transformations, highlight some of their limitations in the low dimensional setting and show how combining affine coupling layers and planar transformations can remedy this.

2. Background

2.1. Normalizing Flows

Normalizing flows model a distribution by transforming a simple, known distribution through a series of transformations. As long as the transformations are differentiable and have differentiable inverses the change in probability can be calculated via a change of variable. Let T be a transformation with inverse T^{-1} such that T and T^{-1} are both differentiable and that $\boldsymbol{u} = T^{-1}(\boldsymbol{x})$, where $\boldsymbol{u}$ comes from some known distribution $p_u(\boldsymbol{u})$. Then the distribution of $\boldsymbol{x}$ can be found by $p_x(\boldsymbol{x}) = p_u(\boldsymbol{u})|\det \boldsymbol{J}_T(\boldsymbol{u})|^{-1}$ If the flow consists of multiple transformations this can be extended using the property that the transformations are composable. For the case of K transformations the modelled distribution becomes

$$p_x(\boldsymbol{x}) = p_u(\boldsymbol{u}) \prod_{k=1}^{K} \left|\det \boldsymbol{J}_{T_k}(\boldsymbol{z}_{k-1})\right|^{-1}, \tag{1}$$

where $\boldsymbol{z}_k$ is the value after the k'th transformation with $\boldsymbol{z}_0 = \boldsymbol{u}$ and $\boldsymbol{z}_K = \boldsymbol{x}$. Often these transformations are defined as a function $f_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$.

Two common transformations are the affine coupling layers (Dinh et al., 2016) and the planar transformations (Rezende & Mohamed, 2015). The affine coupling layer uses an autoregressive structure where half of the dimensions in the input are used to condition the other half. Let $\boldsymbol{x}$ denote the input to the transformation and $\boldsymbol{x'}$ denote the output of the transformation. The transformation is then

$$\begin{aligned} \boldsymbol{x'}_{<d} &= \boldsymbol{x}_{<d} \\ (\boldsymbol{h}_{d+1}, \dots, \boldsymbol{h}_D) &= c(\boldsymbol{x}_{<d}) \\ x'_i &= \tau(x_i; \boldsymbol{h}_i) \;\; i > d, \end{aligned} \tag{2}$$

where τ is an affine transformation

$$\tau(x_i; \boldsymbol{h}_i) = \alpha_i x_i + \beta_i \;\; \boldsymbol{h}_i = \{\alpha_i, \beta_i\} \tag{3}$$

and c is a feed forward neural network with parameters $\boldsymbol{\theta}$. The planar transformation uses a transformation of the form

$$\boldsymbol{x'} = \boldsymbol{x} + \boldsymbol{v}\sigma(\boldsymbol{w}^\top \boldsymbol{x} + b), \tag{4}$$

where the parameters are $\boldsymbol{\theta} = \{\boldsymbol{w}, \boldsymbol{v}, b\}$ and σ is the hyperbolic tangent function. A diagram of the affine coupling layer and the planar transformation can be seen in Appendix A. For an in-depth presentation of NFs see (Papamakarios et al., 2019) and (Kobyzev et al., 2019).

2.2. Conditional Normalizing Flows

In order to extend normalizing flows to conditional density estimation that condition the dependent variables $\boldsymbol{y}$ on the

[1]Technical University of Denmark, Kgs. Lyngby, Denmark, 2800. Correspondence to: Mathias Niemann Tygesen <s153583@student.dtu.dk>.

Proceedings of the 37^{th} International Conference on Machine Learning, Vienna, Austria, PMLR 108, 2020.

independent variables $\boldsymbol{x}$, each of the transformations in the flow is conditioined on $\boldsymbol{x}$. As such the log conditional probability of an observation is:

$$\log p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta}) = \log p_u(\boldsymbol{u}) - \sum_{i=1}^{K} \log \left| \det \boldsymbol{J}_{f_{\boldsymbol{x},\boldsymbol{\theta}}}(\boldsymbol{z}_{k-1}) \right|.$$

The affine coupling layer can be extended to a Conditional Affine Coupling Layer as in (Lu & Huang, 2019). The independent variables $\boldsymbol{y}$ are first sent through a neural network cn for feature extraction. The extracted features are then sent through c along with half of the dimensions

$$\begin{aligned} \boldsymbol{x}_r &= cn(\boldsymbol{x}) \\ \boldsymbol{y'}_{<d} &= \boldsymbol{y}_{<d} \\ (\boldsymbol{h}_{d+1}, \ldots, \boldsymbol{h}_D) &= c(\boldsymbol{y}_{<d}, \boldsymbol{x}_r) \\ y'_i &= \tau(y_i; \boldsymbol{h}_i), \quad i > d, \end{aligned} \tag{5}$$

where $\boldsymbol{y}$ are the dependent variables before the transformation and $\boldsymbol{y'}$ are the dependent variables after the transformation.
The planar transformation can be extended to a conditional planar transformation as in the implementation in the probabilistic programming language *Pyro* (Bingham et al., 2018). The independent variables $\boldsymbol{x}$ are sent through a feature extraction network cn and then the features are sent through a neural network c in order to generate the parameters for the transformation $\boldsymbol{\theta} = \{\boldsymbol{w}, \boldsymbol{v}, b\}$. The full transformation is then

$$\begin{aligned} \boldsymbol{x}_r &= cn(\boldsymbol{x}) \\ (\boldsymbol{w}, \boldsymbol{v}, b) &= c(\boldsymbol{x}_r) \\ \boldsymbol{y'} &= \boldsymbol{y} + \boldsymbol{v}\sigma(\boldsymbol{w}^\top \boldsymbol{y} + b). \end{aligned} \tag{6}$$

A diagram of the conditional affine coupling layer and the conditional planar transformation can be seen in Appendix A

2.3. Noise Regularization

As conditional generative models are prone to overfitting, we regularize the normalizing flows using *noise regularization* as presented in (Rothfuss et al., 2019). The regularization works by pertubing the training data with a small bit of noise, often Gaussian, before each epoch of training. It can be shown that this is equivalent to adding a cost for the second derivative of the modelled distribution in the loss function. Let l be a loss function and $\boldsymbol{z}$ some observation. The expected loss with noise regularization is then

$$\mathbb{E}_{\boldsymbol{\xi} \sim K(\boldsymbol{\xi})} \left[l\left(\boldsymbol{z} + \boldsymbol{\xi}\right) \right] \approx l\left(\boldsymbol{z}\right) + \frac{h^2}{2} \operatorname{tr}\left(\mathbf{H}\right) \tag{7}$$

where $K(\boldsymbol{\xi})$ is a noise distribution and $\boldsymbol{H} = \nabla^2_{\boldsymbol{z}} l(\boldsymbol{z})$.
In maximum likelihood estimation of conditional density estimation $l(\boldsymbol{x}, \boldsymbol{y}) = -\log p(\boldsymbol{y}|\boldsymbol{x})$ and the noise regularized loss function becomes

$$\begin{aligned} l(\boldsymbol{x}, \boldsymbol{y}) \approx &- \log p\left(\boldsymbol{y}|\boldsymbol{x}\right) - \frac{h^2}{2} \sum_{j=1}^{d_y} \frac{\partial^2 \log p(y|x)}{\partial y^{(j)} \partial y^{(j)}} \\ &- \frac{h^2}{2} \sum_{j=1}^{d_x} \frac{\partial^2 \log p(y|x)}{\partial x^{(j)} \partial x^{(j)}} \end{aligned} \tag{8}$$

where d_x and d_y are the dimension of $\boldsymbol{x}$ and $\boldsymbol{y}$ respectively. The second term is a regularization of the modelled distribution while the third term is a regularization of the conditioning of the density estimation.

3. Experiments

3.1. Three kinds of normalizing flows

In order to see how conditional affine coupling flows perform in a low dimensional settings, we construct a full flow along with two comparison flows. The conditional affine coupling flow consists of conditional affine coupling layers along with batchnorm layers and permutation layers. The batchnorm layers are used to better propagate the training signal as presented in (Dinh et al., 2016) and the permutation is just a simple switch of the dimensions. The first comparison flow is a conditional planar flow with batchnorms and the second comparison flow is a mixed flow with alternating conditional affine coupling layers and planar transformation layers starting with a conditional affine coupling layer. Before each layer we have a batchnorm layer and after each conditional planar transformation layer we have a permutation layer[1]. The layout of all of the flows can be seen in Appendix A.

3.2. Synthetic data

In order to visually compare the expressiveness of the different flows we first train them on a synthetic data set. The data set is the two moons data set as used in (Atanov et al., 2019) but it has been extended to rotate around the center with uniform probability. We train three flows of depth 4, 24 and 48. Each flow is conditioned on a rotation of 1.5 radians. In the first row of Figure 1 we can see that the conditional affine coupling flow becomes better at defining the edges of the distribution the deeper the flow is. However, for all of the depths the conditional affine coupling flow has clearly visible paths of non-zero probability density between the two half moons. We will call such paths "probability paths". In the second row we can see that the conditional planar flow has problems with defining the edges of the distribution for shallower flows but get better with depth. For all of the conditional planar flows it is clearly visible that the

[1]Implementation can be found at https://github.com/MathiasNT/Normalizing_Flows_on_Geospatial_Data.

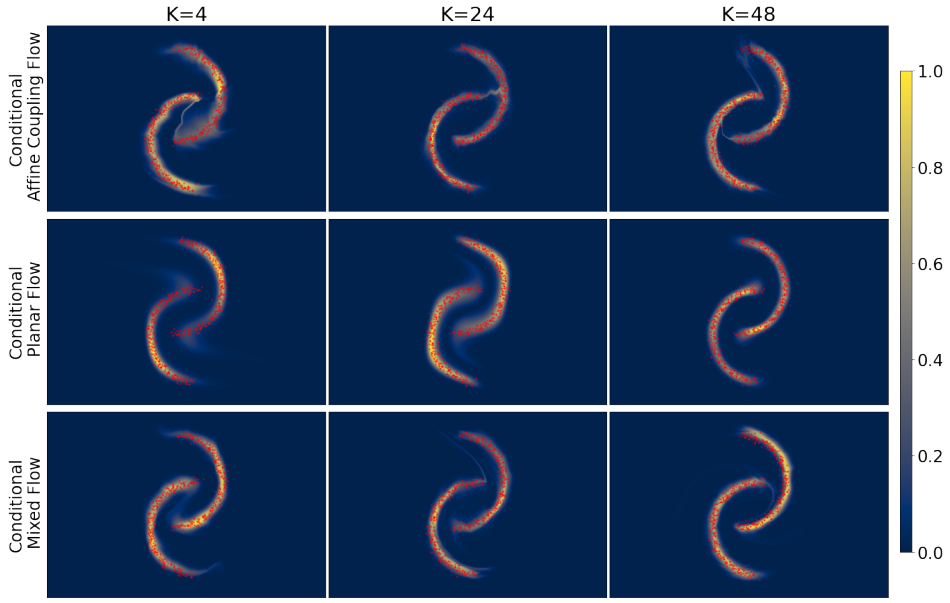


Figure 1. Here we see three different kinds of conditional normalizing flows fitted to the rotating two moons data with the a depth K of 4, 24 and 48. The first row are conditional affine coupling flows, the second row is conditional planar flows and the third row are mixed flows with alternating affine coupling layers and planar transformation layers. All of the flows are conditoned on a rotation of 1.5 radians. The red dots are samples from the corresponding two moons.

planar transformation has no problem with modelling the disjoint moons. In the third row, we can see how a mixed flow with both conditional affine coupling layers and planar transformations have the ability to define clear edges due to the conditional affine coupling layers and is better than the conditional affine coupling flow at creating disjoint distributions due to the planar transformations.

3.3. Geospatial transport data

One real world geospatial setting where conditional normalizing flows could be used is modelling the distribution of the demand for transport. One example of this, prevalent in literature, is the problem of modelling taxi dropoff locations on Manhattan. The data set is a processed NYC Yellow Taxi data set originally presented in (Dutordoir et al., 2018) and also used in (Rothfuss et al., 2019). The data set has the longitude and latitude of the taxi drop off location as dependent variables while the coordinates of the taxi pickup, hour of the day and day of the week are independent variables. We trained flows of each kind with a depth of 48 to the data. All of the flows were conditioned on a pickup at 18:00 on a Tuesday on Manhattan. In Figure 2 we have visualized the conditional affine coupling flows modelled distribution in the area between Manhattan and JFK airport. In the figure there are clear probability paths between Manhattan and JFK airport. In Figure 3 we have visualized the same area for the conditional planar flow. For the conditional planar flow there is no visible probability path and the area around JFK is modelled as its own disjoint probability. However the distribution on Manhattan extends out over the East River. In Figure 4 the same area is visualized for the conditional mixed flow. Again the area around JFK is modelled as a disjoint probability, this time extending out to cover the airport train stations. Furthermore the distribution around the Brooklyn area has complex enough shape to allow it to avoid the East River. As such we can see that mixing together both affine coupling flows and planar flow in the same flow gives both the ability to model disjoint probabilities and complex distributions at lower flow depth than a single type of layer can do on its own.

4. Discussion

As shown by the experiments, the conditional affine coupling flow is capable of modelling complex distributions but have problems with creating disjoint areas of distribution. The affine transformation of a dimension is done given the same dimension, albeit with parameters conditioned upon

Figure 2. A conditional affine coupling flow conditioned on 18:00 Tuesday with pickup on Manhattan. The blue circles show the JFK and LaGuardia Airport and the blue dots show the JFK airport train stations. Upper right corner shows overview of NYC.

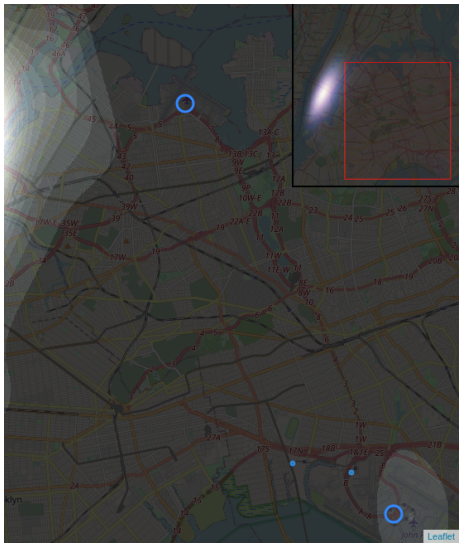


Figure 3. A conditional planar flow conditioned on 18:00 Tuesday with pickup on Manhattan. The blue circles show the JFK and LaGuardia Airport and the blue dots show the JFK airport train stations. Upper right corner shows overview of NYC

the other dimension. As such the transformation of a dimension can only be a scale and translation, corresponding to spreading and pushing the distribution around. Therefore the affine coupling layer is unable to create disjoint probabilities, however for deeper flows these probability paths can be stretched thin. As deeper NFs need more regularization to avoid overfitting as they become more expressive, perhaps shallower mixed NFs with the right kind of expressiveness are easier to regularize. While this problem of probability paths is clearly visible in low dimensions it could theoretically persist in higher dimensions.

The planar transformation can be seen as expanding or contracting the distribution perpendicular to a hyperplane. As such it has no problem creating disjoint areas of distribution but as the conditioning between the dimensions is much simpler this leads to distributions that have a much simpler shape. This can be remedied by resorting to deeper flows.

However as we have demonstrated by combining the two types of layers we can have both. The mixed flow showed the ability to create disjoint areas of probability while maintaining flexible enough shape to model complex distributions.

This raises multiple questions: Are the probability paths a problem in higher dimensions in practice? If they are, how do they impact the modelling of e.g. images? Could more complex transformers in coupling layers solve this problem? What is the best way to combine different kinds of layers together in a single flow model?

Figure 4. A conditional mixed flow conditioned on 18:00 Tuesday with pickup on Manhattan. The blue circles show the JFK and LaGuardia Airport and the blue dots show the JFK airport train stations. Upper right corner shows overview of NYC

References

Atanov, A., Volokhova, A., Ashukha, A., Sosnovik, I., and Vetrov, D. Semi-Conditional Normalizing Flows for Semi-Supervised Learning. may 2019. URL http://arxiv.org/abs/1905.00505.

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., and Goodman, N. D. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 2018.

Dinh, L., Sohl-Dickstein, J., and Bengio, S. Density estimation using Real NVP. may 2016. URL http://arxiv.org/abs/1605.08803.

Dutordoir, V., Salimbeni, H., Deisenroth, M., and Hensman, J. Gaussian Process Conditional Density Estimation. oct 2018. URL http://arxiv.org/abs/1810.12750.

Kobyzev, I., Prince, S., and Brubaker, M. A. Normalizing Flows: An Introduction and Review of Current Methods. aug 2019. URL http://arxiv.org/abs/1908.09257.

Lu, Y. and Huang, B. Structured Output Learning with Conditional Generative Flows. may 2019. URL http://arxiv.org/abs/1905.13288.

Papamakarios, G., Nalisnick, E., Rezende, D. J., Mohamed, S., and Lakshminarayanan, B. Normalizing Flows for Probabilistic Modeling and Inference. dec 2019. URL https://arxiv.org/abs/1912.02762.

Rezende, D. J. and Mohamed, S. Variational Inference with Normalizing Flows. may 2015. URL http://arxiv.org/abs/1505.05770.

Rothfuss, J., Ferreira, F., Boehm, S., Walther, S., Ulrich, M., Asfour, T., and Krause, A. Noise Regularization for Conditional Density Estimation. jul 2019. URL http://arxiv.org/abs/1907.08982.

A. Diagrams of different transforms

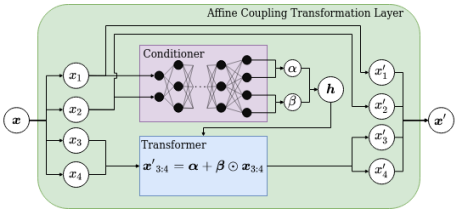


Figure 5. The Affine Coupling Layer visualized for four dimensions.

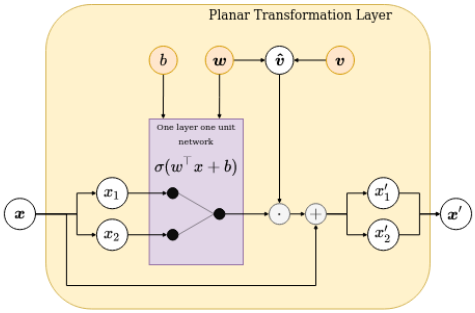


Figure 6. The Planar Transformation visualized for two dimensions.

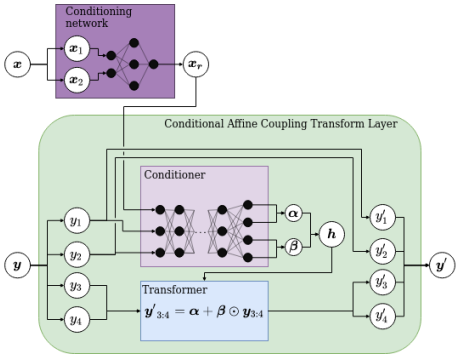


Figure 7. The Conditional Affine Coupling Layer visualized for a flow with four dimensional dependent variables and two dimensional independent variables.

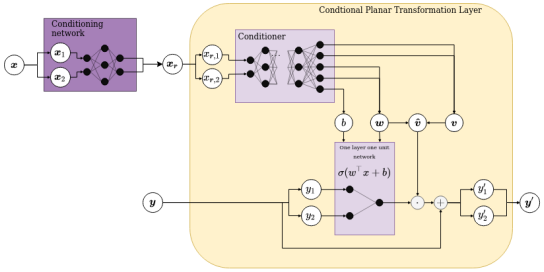


Figure 8. The Conditional Planar Transformation visualized for a flow with two dimensional dependent variables and two dimensional independent variables.

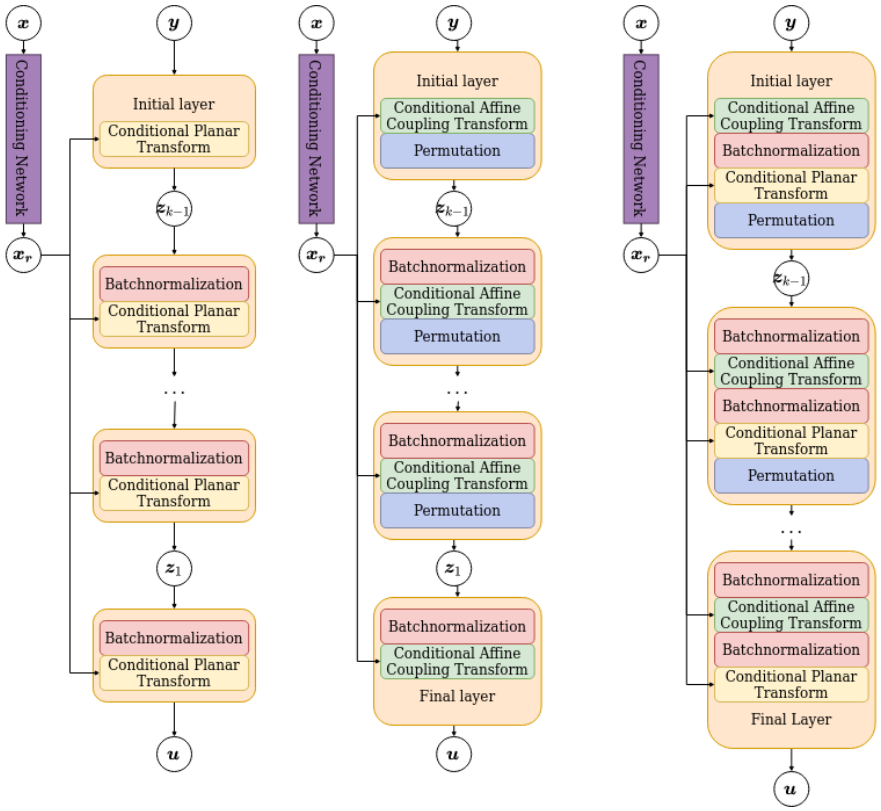


Figure 9. To the left we see a conditional planar flow with batchnorm layers. In the middle we see a conditional affine coupling flow with batchnorm layers and permutation layers. To the right we a mixed flow with both conditional affine coupling layers and planar transformation layers along with batchnorm and permuation layers. Note that all of the layers have no initial batchnorm and the layers with permutation layers have the last permutation removed.

Bibliography

[And+14] Elsa Andersen et al. *Upgrade and extension of the climate station at DTU Byg*. English. Report no.SR 14-01 (UK). Technical University of Denmark, Department of Civil Engineering, 2014. ISBN: 1601-8605.

[And+17] Elsa Andersen et al. *Availability of high quality weather data measurements*. English. DTU Civil Engineering Reports R-379. Technical University of Denmark, Department of Civil Engineering, 2017.

[Ata+19] Andrei Atanov et al. "Semi-Conditional Normalizing Flows for Semi-Supervised Learning". In: (May 2019). arXiv: 1905.00505. URL: http://arxiv.org/abs/1905.00505.

[Beh+18] Jens Behrmann et al. "Invertible Residual Networks". In: (November 2018). arXiv: 1811.00995. URL: http://arxiv.org/abs/1811.00995.

[Ber+18] Rianne van den Berg et al. "Sylvester Normalizing Flows for Variational Inference". In: (March 2018). arXiv: 1803.05649. URL: http://arxiv.org/abs/1803.05649.

[Bis06] Christopher M. Bishop. *Pattern recognition and machine learning*. Springer, 2006, page 738. ISBN: 9781493938438. URL: https://findit.dtu.dk/en/catalog/2304902913.

[DSB16] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. "Density estimation using Real NVP". In: (May 2016). arXiv: 1605.08803. URL: http://arxiv.org/abs/1605.08803.

[Dur+19a] Conor Durkan et al. "Cubic-Spline Flows". In: (June 2019). arXiv: 1906.02145. URL: http://arxiv.org/abs/1906.02145.

[Dur+19b] Conor Durkan et al. "Neural Spline Flows". In: (June 2019). arXiv: 1906.04032. URL: http://arxiv.org/abs/1906.04032.

[Dut+18] Vincent Dutordoir et al. "Gaussian Process Conditional Density Estimation". In: (October 2018). arXiv: 1810.12750. URL: http://arxiv.org/abs/1810.12750.

[DZ03] Jan G De Gooijer and Dawit Zerom. "On conditional density estimation". In: *Statistica Neerlandica* 57.2 (2003), pages 159–176. ISSN: 14679574, 00390402. DOI: 10.1111/1467-9574.00226.

[EHT03] Bernd Engelmann, Evelyn Hayden, and Dirk Tasche. "Measuring the Discriminative Power of Rating Systems". In: 2003,01 (2003). URL: https://EconPapers.repec.org/RePEc:zbw:bubdp2:2225.

[GM03] A G Gray and A W Moore. "Nonparametric density estimation: Toward computational tractability". In: *Siam Proc S* (2003), pages 203–211.

[HGI07] Michael P Holmes, Alexander G Gray, and Charles Lee Isbell. "Fast nonparametric conditional density estimation". In: *Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence, Uai 2007* (2007), pages 175–182.

[Ho+19] Jonathan Ho et al. "Flow++: Improving Flow-Based Generative Models with Variational Dequantization and Architecture Design". In: (February 2019). arXiv: 1902.00275. URL: https://arxiv.org/abs/1902.00275.

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: (February 2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[KD18] Diederik P. Kingma and Prafulla Dhariwal. "Glow: Generative Flow with Invertible 1x1 Convolutions". In: (July 2018). arXiv: 1807.03039. URL: https://arxiv.org/abs/1807.03039.

[Kin+16] Diederik P. Kingma et al. "Improving Variational Inference with Inverse Autoregressive Flow". In: (June 2016). arXiv: 1606.04934. URL: http://arxiv.org/abs/1606.04934.

[KPB19] Ivan Kobyzev, Simon Prince, and Marcus A. Brubaker. "Normalizing Flows: An Introduction and Review of Current Methods". In: (August 2019). arXiv: 1908.09257. URL: http://arxiv.org/abs/1908.09257.

[Kre78] Erwin Kreyszig. *Introductory functional analysis with applications*. Wiley, 1978, XIV, 688 S (unknown). ISBN: 0471507318, 0471504599, 9780471504597.

[LH19] You Lu and Bert Huang. "Structured Output Learning with Conditional Generative Flows". In: (May 2019). arXiv: 1905.13288. URL: http://arxiv.org/abs/1905.13288.

[Nie18] Michael A. Nielsen. *Neural Networks and Deep Learning*. misc. 2018. URL: http://neuralnetworksanddeeplearning.com/.

[Oor+17] Aaron van den Oord et al. "Parallel WaveNet: Fast High-Fidelity Speech Synthesis". In: (November 2017). arXiv: 1711.10433. URL: http://arxiv.org/abs/1711.10433.

[Pap+19] George Papamakarios et al. "Normalizing Flows for Probabilistic Modeling and Inference". In: (December 2019). arXiv: 1912.02762. URL: https://arxiv.org/abs/1912.02762.

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pages 2825–2830.

[PPM17] George Papamakarios, Theo Pavlakou, and Iain Murray. "Masked Autoregressive Flow for Density Estimation". In: (May 2017). arXiv: 1705.07057. URL: https://arxiv.org/abs/1705.07057.

[RM15] Danilo Jimenez Rezende and Shakir Mohamed. "Variational Inference with Normalizing Flows". In: (May 2015). arXiv: 1505.05770. URL: http://arxiv.org/abs/1505.05770.

[Rot+19a] Jonas Rothfuss et al. "Conditional Density Estimation with Neural Networks: Best Practices and Benchmarks". In: (March 2019). arXiv: 1903.00954. URL: http://arxiv.org/abs/1903.00954.

[Rot+19b] Jonas Rothfuss et al. "Noise Regularization for Conditional Density Estimation". In: (July 2019). arXiv: 1907.08982. URL: http://arxiv.org/abs/1907.08982.

[San+18] Shibani Santurkar et al. "How Does Batch Normalization Help Optimization?" In: (May 2018). arXiv: 1805.11604. URL: http://arxiv.org/abs/1805.11604.

[Sil86] B W Silverman. *Density estimation for statistics and data analysis*. Chapman and Hall, 1986, 175 s.

[SP10] Skipper Seabold and Josef Perktold. "statsmodels: Econometric and statistical modeling with python". In: *9th Python in Science Conference*. 2010.

[Sri+14] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pages 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[TT18] Brian L Trippe and Richard E Turner. "Conditional Density Estimation with Bayesian Normalising Flows". In: (February 2018). arXiv: 1802.04908. URL: https://arxiv.org/abs/1802.04908.

[Win+19] Christina Winkler et al. "Learning Likelihoods with Conditional Normalizing Flows". In: (November 2019). arXiv: 1912.00042. URL: http://arxiv.org/abs/1912.00042.

{2.7182818284}ϑωερτυθιοπσδφγηξκλ