NORMALIZING FLOWS AND CONTINUOUS MULTI-OTSU THRESHOLDING APPLIED TO EMPIRICAL WAVELET TRANSFORMS

A Thesis

Presented to the

Faculty of

San Diego State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Applied Mathematics

with a Concentration in

Dynamical Systems

---

by

Stefan Cline

Spring 2023

SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Thesis of Stefan Cline:

Normalizing Flows and Continuous Multi-Otsu Thresholding

Applied to Empirical Wavelet Transforms

Christopher Curtis, Chair
Department of Mathematics and Statistics

Jérôme Gilles
Department of Mathematics and Statistics

Ashkan Ashrafi
Department of Electrical and Computer Engineering

Approval Date

Copyright © 2023
by
Stefan Cline

DEDICATION

Dedicated to my wife, Anna. Her unwavering love and support made this possible.

Я посвящаю свой тезис моей жене Анне. Этот проект стал возможен только благодаря её нескончаемой любви и поддержке.

ABSTRACT OF THE THESIS

Normalizing Flows and Continuous Multi-Otsu Thresholding
Applied to Empirical Wavelet Transforms
by
Stefan Cline
Master of Science in Applied Mathematics with a Concentration in Dynamical Systems
San Diego State University, 2023

Normalizing flows have proven to be powerful tools for generative machine learning (ML) models. Normalizing flows work by learning a bijective mapping in the form of a neural net (NN) from one space, known as the base which is usually a Gaussian distribution, to another, known as the target distribution which is a dataset with an unknown probability density function (PDF). This process aims to find an analytically tractable PDF for the target dataset. In this work, we use normalizing flows to attempt to learn how to better partition a signal's frequency space representation, thereby providing an alternative approach to boundary point determinations of the Empirical Wavelet Transforms (EWT) algorithm. As such, we study a popular class identification method, known as Otsu's Method (OM), which has found extensive use in EWT research. In conclusion, we remark that a novel technique we develop, which we named Otsu's Normalizing Flows Method (ONFM), produced low reconstruction errors and interesting EWT components for three rich signals when run through the EWT algorithm. This result leads us to conclude that further research in this area could potentially be valuable for signal processing.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

$\mathcal{F}$ This is generally used to describe a forward pass of the NN from the base distribution to the target distribution.

$\mathcal{F}^{-1}$ This is generally used to describe a backwards pass of the NN from the target distribution to the base distribution.

Bijective A one-to-one and onto function. All functions for normalizing flows must be bijective.

EWT Empirical Wavelet Transform.

FT Fourier Transform.

KDE Kernel Density Estimation is the application of kernel smoothing for probability density estimation, i.e., a non-parametric method to estimate the probability density function of a random variable based on kernels as weights.

ML Machine Learning.

NN Neural Network or Neural Net.

Normalizing Flow A neural net that is a composition of bijective transformations that map some base distribution to a target distribution.

ONFM Otsu's Normalizing Flows Method. This is the name of the novel technique developed in this paper.

PDF Probability Density Function.

Pyro The library used that had pre-built bijective functions.

PyTorch The machine learning framework used to create the neural nets throughout this paper.

ACKNOWLEDGMENTS

I would like to thank Dr. Curtis for guiding me through the thesis writing process and his infinite patience. He regularly handed me reasonable and thought provoking topics to explore, which has made this an overall enjoyable and highly valuable experience. I would also like to thank: Dr. Gilles, Dr. Carretero, Dr. Palacios, Dr. George, and Dr. Mahaffy, for being great educators that made me an objectively better mathematician. I would also like to thank Dr. Ashrafi (and Dr. Gilles again) for agreeing to serve on my thesis committee.

I would also like to thank Seth Minor, Christopher Mack, Joseph Diaz, Susana Mungia-Hernandez, and Michael Juybari-Johnson for being great friends and peers during what was objectively one of the most difficult and stressful times of my life. I could not have done it without them.

CHAPTER 1
INTRODUCTION

Here we introduce Empirical Wavelet Transforms (EWT) from signal processing, and discuss Otsu's Method (OM) and how it is used in EWT. From there, we explore the basics of Neural Nets (NN) and Machine Learning (ML). We then turn our attention towards Normalizing Flows and examine a basic example, as well as discuss the limitations of KDEs. Lastly we briefly discuss the Van der Pol Oscillator (VDP) and the Stochastic VDP (SVDP).

1.1 Empirical Wavelet Transforms as a Path to Otsu's Method and Normalizing Flows

The Fourier series and transform have been studied extensively and date back to the early 1800's thanks to the French mathematician, Jean-Baptiste Joseph Fourier [19]. The main idea is that any piecewise continuous function can be represented as an infinite series of sines and cosines. Furthermore, the Fourier transform (FT) extracts frequency components of harmonic signals from a time series signal or function [8]. A more recent (1909) development is the wavelet transform [5]. Here, we see a resemblance to the Fourier transform with an obvious modification [8]

$$\text{Fourier Transform:} \qquad \hat{f}(\xi) = \int_{\mathbb{R}} f(t) e^{-2\pi i \xi t}\, dt \tag{1.1}$$

$$\text{Wavelet Transform:} \qquad W_f(u,s) = \langle f, \psi_{u,s} \rangle = \int_{\mathbb{R}} f(t) \frac{1}{\sqrt{s}} \overline{\psi\left(\frac{t-u}{s}\right)}\, dt \tag{1.2}$$

We do well to remember that $e^{-2\pi i \xi t}$ can be instead written as a combination of an imaginary sine and real cosine. So, in effect, we have replaced the complex Fourier trigonometric expression with a wavelet. In the wavelet transform, the alteration of the wavelet comes from adjusting or scaling s in the mother wavelet expression, $\psi \in \mathbb{C}$. By convolving ψ with the signal the modes of a signal with both temporally localized high frequency behavior as well as lower frequency more persistent behavior can be captured. This is a departure from sines and cosines which exhibit unchanging oscillatory behavior over the whole real line.

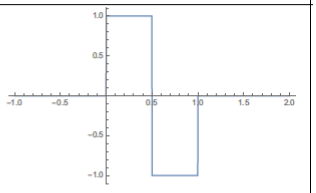
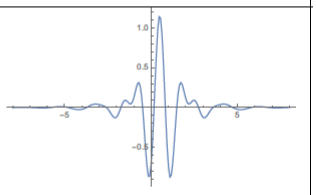
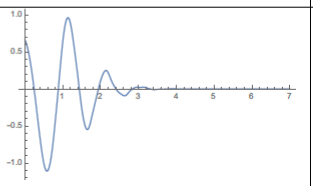
	$\psi(t)$
Haar	
Meyer	
Daubechies ($N = 8$)	

Figure 1.1. Three examples of wavelets [1].

A mother wavelet ψ must satisfy two simple criterion

$$\int_{\mathbb{R}} \psi(t)\ dt = 0$$

$$||\psi(t)||_{L^2} = \left(\int_{\mathbb{R}} |\psi(t)|^2\ dt \right)^{1/2} = 1$$

By scaling the wavelet for discrete steps of the scaling factor s (see Eq. 1.2), the wavelet shape is elongated as shown in Figure (1.2). As a quick example, we show the discretized version of the Wavelet Transform, the Digital Wavelet Transform (DWT), of a chrip signal. Here we will scale a daughter wavelet using the **dyadic DWT**, in the following form (where a^j takes the place of s and $j = \mathbb{N}$)[1]

$$\psi_j[n] = \frac{1}{\sqrt{a^j}} \psi \left[\frac{n}{a^j} \right]$$

Then, doing the circular convolution $(f(t) \circledast \psi_j)[n]$ produces a **scaleogram** as seen in Figure (1.3).

We had at first discussed the possibility of utilizing some kind of ML algorithm to possibly better determine the shape of a wavelet for a given signal or family of signals, but we instead stumbled upon Otsu's Method (OM) [14].

[1]The bracket notation is used to differentiate between discrete and continuous, e.g., continuous $f(t)$ will be written discretely as $f[t]$.

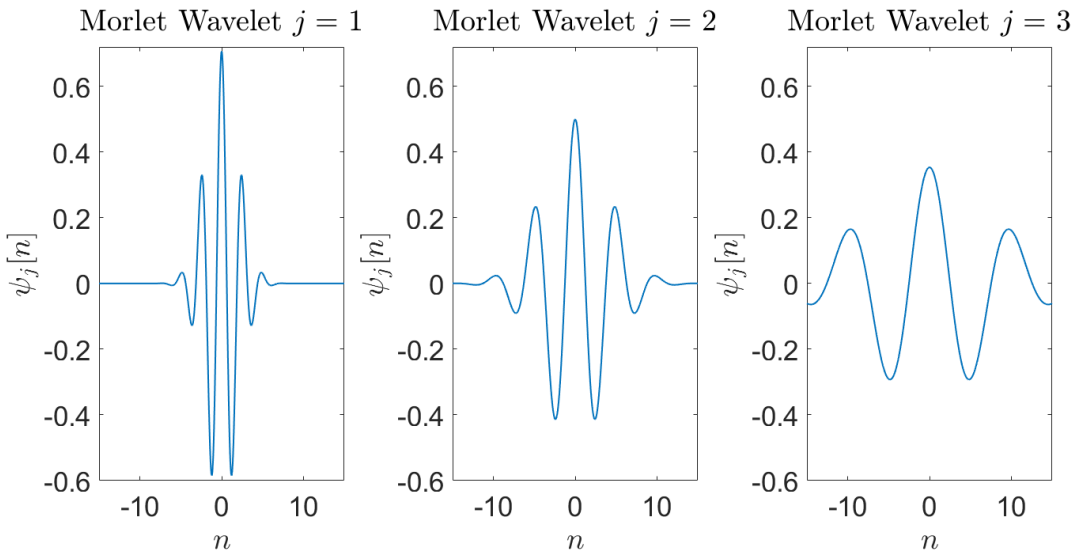


Figure 1.2. The real valued Morlet mother wavelet and two successive elongations by scaling via a^j where $a = 2$.

Oftentimes, one of the main goals of signal processing is to remove unwanted modes or frequencies from a signal to extract its core features. In more simple terms, we desire a filter that removes noise and returns a 'cleaned up' signal. The techniques on how to accomplish this are numerous, each with their own strengths and weaknesses. In the MATLAB repository for Empirical Wavelet Transforms by Dr. Jérôme Gilles, he uses OM to perform this noise filtering which we will explain more thoroughly in Section (1.2.2) [9].

It was this first view of observing a signal as not just some continuous or discontinuous function, but potentially as a *histogram* of data, that sparked an idea by Dr. Christopher Curtis to see if we could improve upon OM via normalizing flows.

Normalizing flows are relatively new (circa. 2017) and are used to map a base distribution (hence the connection to a histogram) to a target distribution in the form of a bijective transformation function [11]. When normalizing flows builds a neural net (NN) this bijective transformation function is tuned by optimizing the parameters of the NN. Broadly speaking,

Target to Base:	$\mathcal{F}(\boldsymbol{s})$,	$\boldsymbol{s} \equiv$ Target Sample
Base to Target:	$\mathcal{F}^{-1}(\boldsymbol{x})$,	$\boldsymbol{x} \equiv$ Base Sample.

A key item to note early on is that $\boldsymbol{x}$ and $\boldsymbol{s}$ are samples from a distribution, they are not an approximation of the Probability Density Function (PDF) (e.g., a Kernel Density Estimate (KDE)) or an explicitly given PDF of their associated dataset. A simple example of the difference is shown in Figure (1.4).

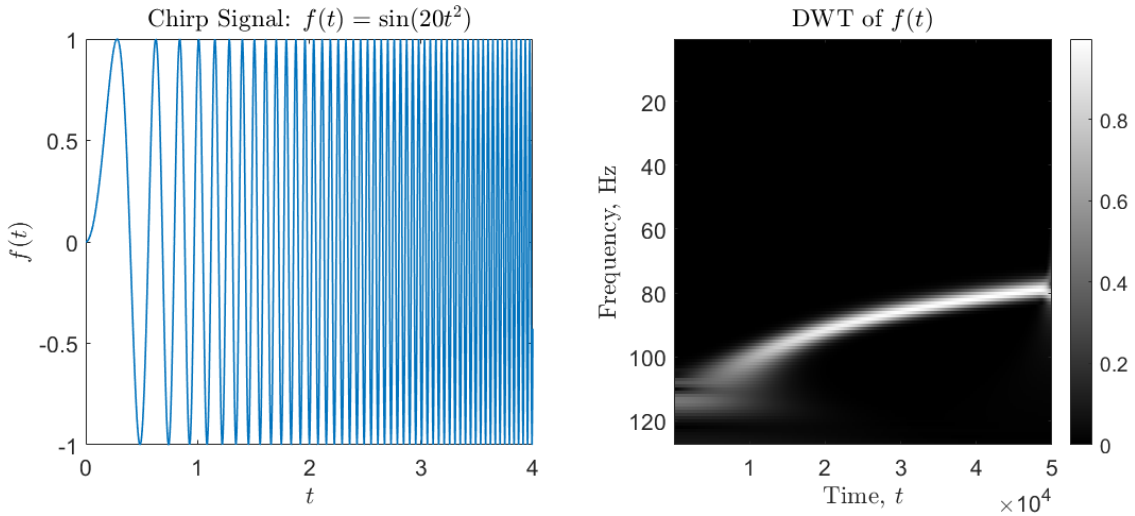


Figure 1.3. Scaleogram of a Morse wavelet (using the default settings for the `cwt` command, and noting that a Morse wavelet is qualitatively similar to a Morlet wavelet) applied to a chirp function using MATLAB's Wavelet Toolbox [1]. The frequency of the chirp starts off low, increasing as time increases. On the scaleogram, we see as time goes on, the bright band starts near the low frequencies, and ends at higher frequencies.

1.2 The Role of Otsu's Method in EWTs

Here we will look first at OM and how it functions, then take a look at how it is used for EWTs.

1.2.1 Otsu's Method

Otsu's Method, developed by the Japanese Mathematician Nobuyuki Otsu in 1979, was primarily envisioned as a foreground and background partitioning algorithm [14], the idea being that for a grayscale image one could find some optimal threshold by which to separate pixels into classes. By doing so images would consume less memory

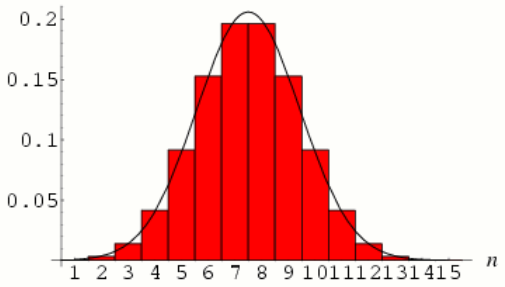


Figure 1.4. The red bars correspond to counting and binning the dataset which is then represented as a histogram [20]. The curve if estimated could be generated utilizing the KDE technique, or if the PDF is known the curve can be the PDF.

as they would be more sparse, and the technique could highlight the critical levels of an image. This could help the key features standout from otherwise background clutter. An example of the technique's intended purpose is shown in Figure (1.5).

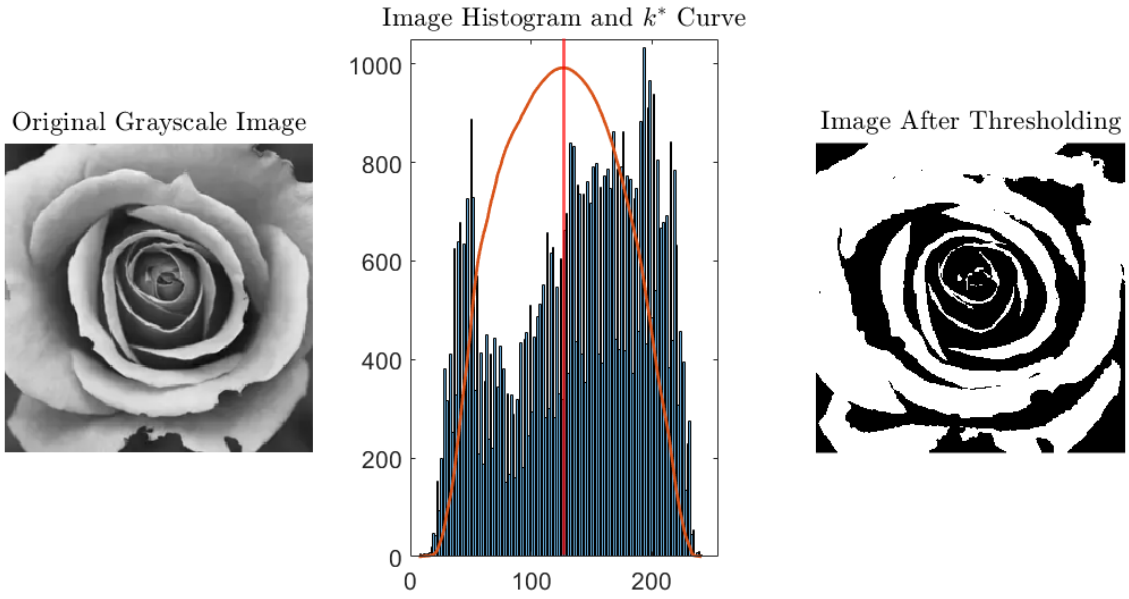


Figure 1.5. Left: The original grayscale image of a rose. Middle: The histogram of the images pixel values. Right: The image after applying the threshold.

The method seeks to maximize between-class variance, which is the orange curve with its threshold intersection shown in the histogram plot of Figure (1.5). If we assume that there are L number of gray levels, such that

$$G_l = [1, 2, ..., L]$$

with N number of pixels and n_p pixels per level then

$$N = n_1 + n_2 + ... + n_L$$

As such, the between-class variance of the case where we assume only two classes (i.e., white and black for an image) $C = \{C_1, C_2\}$ can be described by

$$\sigma_B^2(k) = \frac{[\mu_T \omega(k) - \mu(k)]^2}{\omega(k)[1 - \omega(k)]}$$

where

$$\omega_0 = \Pr(C_0) = \sum_{i=1}^{k} p_i = \omega(k)$$

$$\omega_1 = \Pr(C_1) = \sum_{i=k+1}^{L} p_i = 1 - \omega(k)$$

$$\mu_T = \mu(L) = \sum_{i=1}^{L} i p_i$$

$$\mu(k) = \sum_{i=1}^{k} i p_i$$

Which also allows us to rewrite the above as

$$\sigma_B^2(k) = \frac{[\mu_T \omega(k) - \mu(k)]^2}{\omega_0 \omega_1}$$

Effectively, this algorithm starts at $k = 1$, calculates a value for $\sigma_B^2(k = 1)$, then moves to $k = 2$, calculates $\sigma_B^2(k = 2)$, etc. After all of these calculations are performed, the optimal threshold, k^*, is simply where the inter-class variance is maximized. Alternatively,

$$\sigma_B^2(k^*) = \max_{1 \leq k < L} \sigma_B^2(k)$$

Notice that we do not consider L, the largest pixel value, to be a possible threshold point as that will definitionally make $\omega(k) = \omega_0 = 0$ and thereby give us an undefined value for $\sigma_B^2(k = L)$. Again, in Figure (1.5) this curve is plotted along with the histogram, showing its peak value is the same as the determined threshold level[2].

1.2.2 EWTs and Otsu's Method

The process of performing EWTs is rife with different settings, algorithmic paths, and choices. To avoid the overbearing and frankly unnecessary journey through Dr. Gilles' EWT code, a flow chart overview of it is provided in Appendix B[9].

First, an input signal undergoes a FT by means of the MATLAB `FFT` function or 'Fast Fourier Transform'. This is a discrete function that performs an optimized version of the Digital Fourier Transform (DFT), where we note for $f[n] \in \mathbb{R}$ that $\overline{\hat{f}[k]} = \hat{f}[-k]$

$$\text{DFT:} \qquad \hat{f}[k] = \sum_{n=0}^{N-1} f[n] e^{-2i\pi \frac{nk}{N}}$$

[2]The k^* curve was scaled down by a factor of 2.5 simply to visually compress the graph.

where we have N samples. Given a signal that has been sufficiently sampled in time, the DFT constructs a close approximation to the continuous FT. The signal is then cut in half to remove its mirrored negative side, and the absolute value is taken to turn complex valued terms of $\hat{f}[k]$ into magnitudes.

Second we observe the standard form of a normalized Gaussian distribution and note $|f^*[k]| = |f[k]|$. We also build the Gaussian distribution in frequency space

$$\hat{g}[k;\sigma,\mu] = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(k-\mu)^2}{2\sigma^2}}$$

If we simply select $\mu = 0$ then we have σ to vary. By making $\sigma \ll 1$ we see that $\hat{g}[k;\sigma,0]$ becomes extremely steep effectively returning a delta function. As such, with L and R being the left and right starting and stopping points of the discrete domain

$$(\hat{f} * \hat{g})[k] = \sum_{\tau=L}^{R} \hat{f}[k-\tau]\hat{g}[\tau;\sigma,0]$$

For the steepest case of $\hat{g}[\tau;\sigma,0]$, the discrete function will only be able yield a vector that contains $[0,0,...,0,H,0,...0]$ where $H \equiv$ some height of the function $\hat{g}[k;\sigma,0]$, and as such acts as sliding a heightened identity function across $\hat{f}$ simply returning $\hat{f}$ if H is normalized. As we make σ larger, we slowly smooth out quick peaks and end up with smoother functions as can be seen in Figure (1.6).

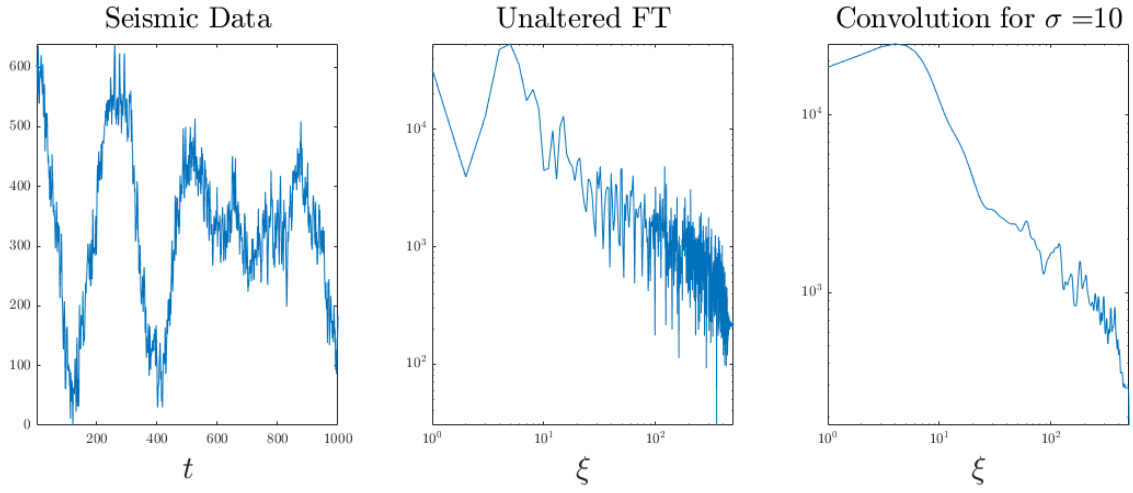


Figure 1.6. Left: A snippet of a signal generated by an earthquake. Middle: The shifted and truncated FFT of the seismic Data. Right: The convolution of that signal at $\sigma = 10$ where we can see some of the features have been smoothed out.

Here is where Otsu and the EWT come together. In Figure (1.7) we can see that for small values of σ there are many local maxima and minima of the curve. As the smoothing via convolutions continues, the total number of peaks and dips becomes smaller and smaller until we are left with an almost entirely smooth function with only a few remaining local minima and maxima. If we save the positions of these peaks per iterated loop as counts of occurrences like one would do with collected data, we can then build a histogram. From this histogram, we then perform OM. The frequencies that survive for a sufficiently long time, now determined by the threshold value of a two class OM, are kept, while the peaks that 'buff out' quickly are removed. Details of the above can be found in Appendix C.

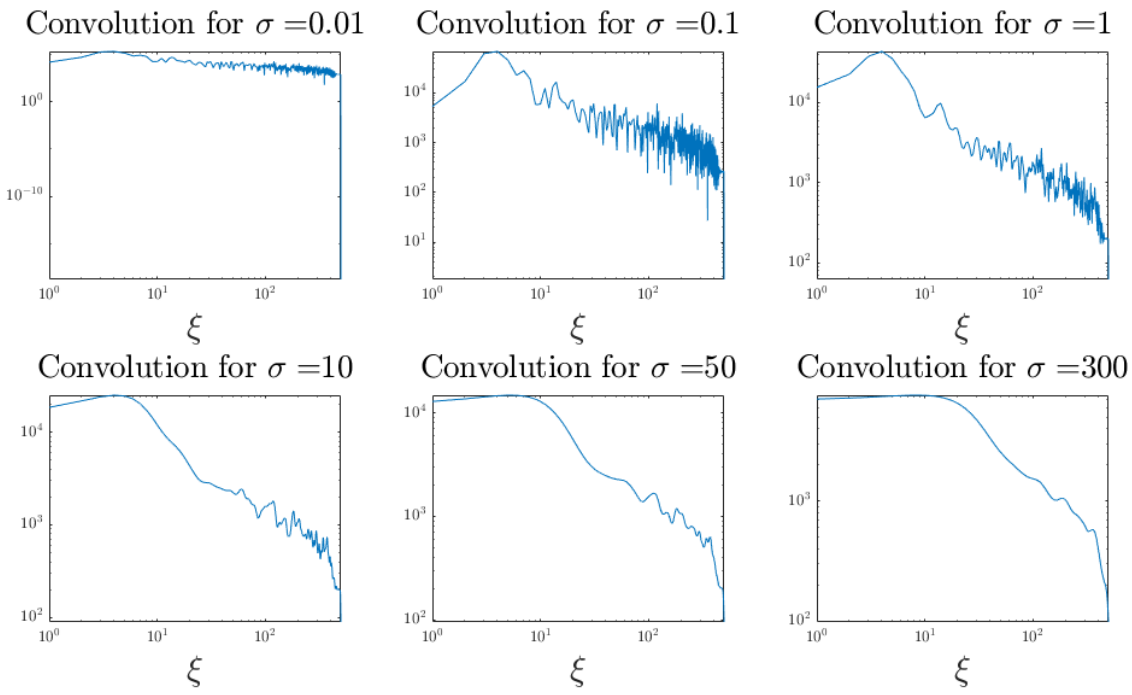


Figure 1.7. The smoothing effect of taking the convolution of the discrete FT with a Gaussian as the value for σ is increased from smaller to larger values. We notice that at $\sigma = 0.01$ it appears to be almost no different than the unaltered FT as seen in Figure (1.6).

Lastly, we make four final notes. First, through some unexplored theory, instead of directly using the function for the Gaussian, a Gaussian kernel in the form of a Modified Bessel function is instead used for small values of σ. We remark however that the principle idea is unchanged, and the above images provide enough motivation for the importance of working further with the Otsu thresholding algorithm. Second, OM, in order to be implemented later, must be altered. Otsu made the assumption that his

algorithm would only be used on images with positive integer pixel values (1 through 255). We instead will want to use it on continuous functions/distributions.

Third we note $\mathcal{F}$ does not preserve k^* that we find from OM which we prove in Section (3.1).

Fourth, in the simple case of $C = \{C_1, C_2\}$, performing Otsu is relatively quick, and thus performing it on the dataset is arbitrary computationally. However, if we wish to say perform a multi-threshold on an arbitrary dataset, we observe that for images or datasets of relatively small sizes (1000^2 pixels), the computational cost of determining the set of $k_M^* = [k_1^*, k_2^*, ..., k_m^*]$ starts to increase rapidly as the number of different combinations to check increases if one uses OM to brute force check the various possibilities. An example of this is provided in Table (1.1). As we can see, for a

Combinatorics Table for m number of k^*		
Number of Thresholds	Combinations	Example $N = 100$
1	$N - 1$	99
2	$\sum_{i=1}^{N-1}[N - i]$	4950
3	$\sum_{i=1}^{N-2}[i(N - 1 - i)]$	161700

Table 1.1. Combinatorics of Multi-Otsu

meaningful number of bins that does not start to simply approach the pixel counts of a discrete image, the number of combinations to check can become unwieldy and scales extremely poorly. As such, we explore using a multi-thresholding algorithm on a continuous function in Section (3.1.1.1).

1.3 Introduction to Neural Nets (NN) and Machine Learning (ML)

In order to get to the point of being able to do normalizing flows, I first needed to understand the basic machinery behind ML and NNs [17, 3]. Let us then examine a simple NN with 3 inputs and 2 outputs which are called **nodes**. Here, our input consisting of three data points, $\vec{h}_1 = [h_{11}, h_{12}, h_{13}]^T$, will then be altered by either **activation functions**, or **transformations** that utilize weights and bias' to produce our two outputs, $\vec{h}_2 = [h_{21}, h_{22}]^T$.

1. Activation Functions: These functions can have several uses, but specific to normalizing flows, they act as a shrinking function that can control numerical

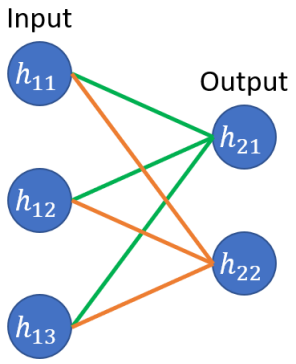


Figure 1.8. Structure of a NN with an input and output layer.

instability (much like the Gram-Schmidt Orthogonalization of eigenvalues while tracking chaotic orbits).

2. Transformations: In Figure (1.8) these are represented by the orange and green lines. An example could be doing a linear transformation, $y = ah_{nm} + b_n$, where each line has some associated weight with a bias per node.

As an example, suppose we first want to constrain our data. We can apply a basic activation function (for this example we will use the bijective Sigmoid), then we will complete a linear mapping. Also suppose $\vec{h}_1 = \{h_{11} = -2, h_{12} = .5, h_{13} = 3\}$, then

$$\begin{aligned} \text{Sigmoid Function} \qquad S(x) &= \frac{1}{1+e^{-x}} \\ S(\vec{h}_1) &= \frac{1}{1+e^{-\vec{h_1}}} \\ S \text{ for Going Through } S(x) \qquad \vec{h}_1^S &= \{0.1192, 0.6225, 0.9526\} \end{aligned}$$

Note that this has the effect of taking $\vec{h}_1 \in \mathbb{R} \to (0, 1)$. Now, we wish to create the values for our output vector, $\vec{h}_2$. Here, if we use a simple linear transformation, we have

$$\begin{aligned} \begin{bmatrix} h_{21} \\ h_{22} \end{bmatrix} &= \underbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}}_{\boldsymbol{A}} \cdot \underbrace{\begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \end{bmatrix}_S}_{\boldsymbol{x}} + \underbrace{\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}}_{\boldsymbol{b}} \\ &= \begin{bmatrix} a_{11}h_{11}^S & a_{12}h_{12}^S & a_{13}h_{13}^S \\ a_{21}h_{11}^S & a_{22}h_{12}^S & a_{23}h_{13}^S \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \end{aligned}$$

Note that each a_{nm} value corresponds to one of the lines between nodes in Figure (1.8). Whatever these values are for h_{21} and h_{22}, is the output of a **forward pass** of the NN. In order for this to learn a better output however, we need a few more pieces. First, how well did these outputs do? We can check these outputs against what we desire

them to be[3]. Suppose that we ended up with

$$\begin{bmatrix} h_{21} \\ h_{22} \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}, \qquad \text{Desired: } \begin{bmatrix} -1 \\ 5 \end{bmatrix}$$

Clearly then we want our first value to decrease and our second value to increase. If we write this as a mean squared error **cost function** or **loss function**, we could see it as ($D \equiv$ desired):

$$C = \left(\begin{bmatrix} h_{21} \\ h_{22} \end{bmatrix}_D - \begin{bmatrix} h_{21} \\ h_{22} \end{bmatrix} \right)^2 = \begin{bmatrix} l_1 \\ l_2 \end{bmatrix}$$

We can use this to determine if the changes we are making to the parameters (in this example our a_{nm} and b_n values) are improving things or if they are making things worse. For clarity we are aiming for $C = [l_1 = 0, l_2 = 0]^T$.

Lastly then, how do we go about improving the values we have for a_{nm} and b_n? If we think of these values as occupying some $nm + n$ dimensional subspace of $\mathbb{R}^{n(m+1)}$, then we have an optimization problem where we seek to *minimize* our loss function, C. Hence, we want to find the negative gradient of our cost function and move to a minimum in our subspace. As such

$$\nabla C = \left[\frac{\partial C}{\partial a_{11}}, \frac{\partial C}{\partial a_{12}}, ..., \frac{\partial C}{\partial b_1}, \frac{\partial C}{\partial a_{21}}, ..., \frac{\partial C}{\partial a_{23}}, \frac{\partial C}{\partial b_2} \right]^T$$

This is simply done by following our equations backwards. Labeling our functions and letting ξ stand for one arbitrary a_{nm} or b_n value

$$\mathcal{L}(\xi) = \boldsymbol{a}\vec{h}_i^S + \boldsymbol{b}$$
$$C(\mathcal{L}) = (\mathcal{L} - \vec{h}_D)^2$$

So we can see the effect of a single parameter ξ on C as a chain rule

$$\frac{\partial C}{\partial \xi} = \frac{\partial C}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial \xi}$$

To be clear, we will perform this for a_{11} and b_1

$$a_{11}: \qquad \frac{\partial C}{\partial a_{11}} = 2 \cdot \left(a_{11} h_{11}^S + a_{12} h_{12}^S + a_{13} h_{13}^S \right) \cdot h_{11}^S$$
$$b_1: \qquad \frac{\partial C}{\partial b_1} = 2 \cdot \left(a_{11} h_{11}^S + a_{12} h_{12}^S + a_{13} h_{13}^S \right) \cdot (1)$$

[3]If we know what these outputs are either by value or label, this is known as **supervised learning** which is not the type of problem normalizing flows are, as normalizing flows are **unsupervised learning**.

Hence, $-\nabla C$ is computed via basic gradient descent. The final step is then to either add or subtract some small step known as the **learning rate** from the parameter, thereby causing it to move 'downhill' in the subspace along that parameter's axis. By repeating this process multiple times, we should expect that the loss will be minimized, thereby getting us closer to the desired outputs. This entire process is called **back propagation**.

The example NN that we have shown so far however is not what we would call a **deep NN**. Deep NNs imply that there are 'hidden layers' between the inputs and the outputs as shown in Figure (1.9). However, the process for determining $-\nabla C$ is the

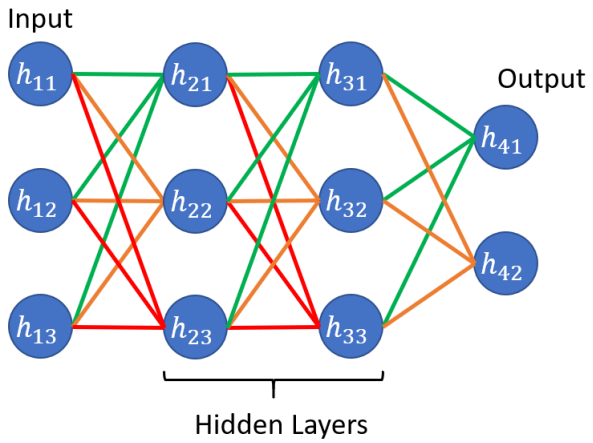


Figure 1.9. An example of a deep NN with hidden layers

same. We just note that with additional layers, a single weight will influence multiple downstream paths, and the calculation of the parameter's overall effect on C is slightly more complicated, but fundamentally unchanged.

1.4 Normalizing Flows

Here we will show some of the underlying theory of normalizing flows and provide a toy example.

1.4.1 Normalizing Flows Theory

A normalizing flow is a transformation of a simple probability distribution (e.g., a standard normal) into a more complex distribution by a sequence of invertible and differentiable mappings [11]. We start by letting $\boldsymbol{Z} \in \mathbb{R}^D$ where $\boldsymbol{Z} \equiv$ random variable with a known density function (e.g., a standard normal Gaussian) $p_Z : \mathbb{R}^D \to \mathbb{R}$. If we then let g be an invertible function and $\boldsymbol{Y} = g(\boldsymbol{Z})$, then utilizing the change of

variables formula, we can find a PDF for $\boldsymbol{Y}$. Namely,

$$p_{\boldsymbol{Y}}(\boldsymbol{y}) = \frac{p_{\boldsymbol{Z}}(f(\boldsymbol{y}))}{|\det Dg(f(\boldsymbol{y}))|}$$

where we have $f^{-1} = g$ and $Df(\boldsymbol{y})$ is the Jacobian of f. The name normalizing flow comes from the fact that the function f (which is also g^{-1}) returns the more complex distribution to the normal distribution. We make the critical note now that f and g are bijective functions in whatever $\mathbb{R}^D$ subspace they exist in.

Now, as is easy to see in practice, single functions with optimized parameters make for a poor model. As such, stringing multiple bijective functions together can improve results rather drastically. Hence, we can have

$$f = f_1 \circ f_2 \circ \ldots \circ f_{N-1} \circ f_N$$
$$g = g_N \circ g_{N-1} \circ \ldots \circ g_2 \circ g_1$$

Which then simply means our determinant Jacobian becomes

$$\det Df(\boldsymbol{y}) = \prod_{i=1}^{N} \det Df_i(\boldsymbol{x}_i)$$

This allows us to see that so long as we maintain that each f_i is bijective, our NN can be given a lot of flexibility in how it manipulates each function's parameters.

As final piece to the basic theory explained above, we show an interesting side note of the preservation of area as it relates to some point $a \in X$ and some point $f(a) \in Z$ in Appendix E. Namely while using Neural Spline Flows (see Section (2.2.3.2.1)) which is the NN model exclusively used in this paper, the following holds

$$\int_{-\infty}^{a} p_Y(y)\ dy = \int_{-\infty}^{f(a)} p_Z(z)\ dz.$$

1.4.2 Normalizing Flows Toy Example

Let us start with a simple Gaussian distribution where we wish to transform it into a 'double humped Gaussian' where we maintain the same area under our respective normalized curves. So we let

$$p_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}, \qquad \text{where } \sigma = 1,\ \mu = 0$$

$$p_Z(z) = \frac{1}{2\sigma\sqrt{2\pi}} e^{-\frac{(z-\mu)^2}{2\sigma^2}} + \frac{1}{2\sigma\sqrt{2\pi}} e^{-\frac{(z+\mu)^2}{2\sigma^2}}, \qquad \text{where } \sigma = 1,\ \mu = 2$$

This then gives us the nice property that both of these functions have a normalized area that is equal to one.

Now, by change of variables (and with a desire to manipulate PDF_Z into PDF_X via some bijective f), from Section (1.4.1) we have

$$p_X(x) = p_Z(z)[f'(z)]^{-1}, \qquad \text{where } x = f(z)$$

Here, because we are only dealing with the 1D case, it may not be readily obvious that the $[f'(z)]^{-1}$ term is actually the one divided by the absolute value of the Jacobian determinant as discussed earlier. If we expand this out we have

$$\begin{aligned}
p_X(x) &= p_Z(z)[f'(z)]^{-1} \\
p_X(f(z)) &= p_Z(z)[f'(z)]^{-1} \\
\frac{1}{\sqrt{2\pi}} e^{\frac{-f(z)^2}{2}} &= [f'(z)]^{-1} \left[\frac{1}{2\sqrt{2\pi}} e^{-\frac{(z-\mu)^2}{2}} + \frac{1}{2\sqrt{2\pi}} e^{-\frac{(z+\mu)^2}{2}} \right] \\
f'(z) &= \frac{1}{\sqrt{2\pi}} e^{\frac{f(z)^2}{2}} \left[\frac{1}{2\sqrt{2\pi}} e^{-\frac{(z-\mu)^2}{2}} + \frac{1}{2\sqrt{2\pi}} e^{-\frac{(z+\mu)^2}{2}} \right]
\end{aligned}$$

This is a rather heinous ODE that can only be solved numerically. Utilizing MATLAB's ODE45 function with the following code

```
FZ = @(z,fz) 1/sqrt(2*pi) *exp(fz^2/2)*(1/2)*(1/sqrt(2*pi)
      *exp(-(z-mu)^2/2)+1/sqrt(2*pi)*exp(-(z+mu)^2/2));
[z,fz] = ode45(FZ,[-7.5,7.5],-0.2);
plot(z,fz,LineWidth=3)
```

we produce a numerical approximation to the exact solution of the desired bijective function $f(z)$ as seen in Figure (1.10). Clearly, even for a rather simple setup with well behaved functions that we know ahead of time, we still are left to wrestle with an unsightly ODE to produce f, our transformation function. As such, the problem of non-linearly distorting even a 1D axis into a different shape (leading to a new PDF) is clearly non-trivial.

The key takeaway is that this task is impossible analytically for an arbitrary dataset as the dataset's PDF is unknown. Therefore, if we are instead able to use a bijective neural net, which then takes the place of f, we can learn PDF_Z by using some PDF_X (in this case a normal Gaussian). This is the function $\mathcal{F}(\boldsymbol{s})$ from Section (1.1). With PDF_Z characterized, we may then have a clearer picture of the nature of our data.

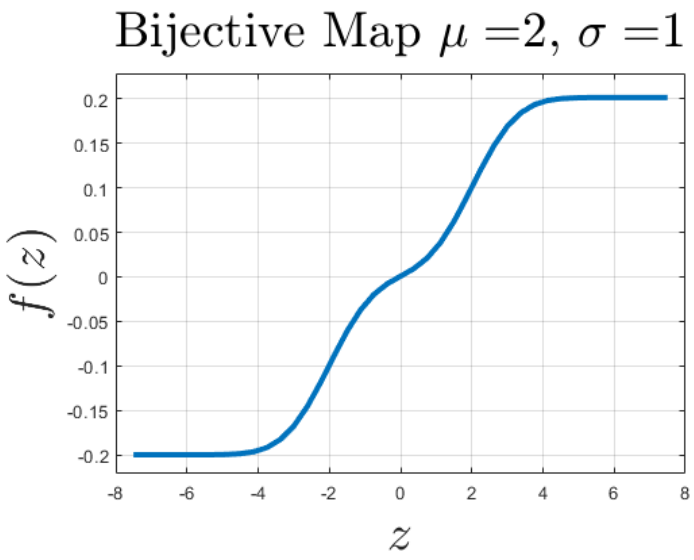


Figure 1.10. Selecting $\mu = 2$ and $\sigma = 1$ and setting an initial condition of $f(t_0) = -0.2$ to roughly allow the function to pass through the origin, we see the transformation that would be required to map a double hump Gaussian to a normal Gaussian ($p_Z(z) \mapsto p_X(x)$).

1.4.3 Limitations of the KDE

We wish to pivot to addressing an obvious concern. Why use normalizing flows as opposed to simply making a KDE of the dataset? Suppose we have a random sample $x_1, ..., x_N$ from a PDF $f_X(x)$ assuming $X \in \mathbb{R}$. The smooth *Parzen* estimate is the typical choice to estimate the value of the PDF f_X at some point x_0

$$\hat{f}_X(x_0) = \frac{1}{N\lambda} \sum_{i=1}^{N} K_\lambda(x_0, x_i)$$

Here, we note that λ is the width and K_λ is the kernel. A common choice is the Gaussian Kernel $K_\lambda(x_0, x) = \phi(|x - x_0|/\lambda)$[10]. In this subtle equation above, there is a rather large assumption being made when producing a KDE; namely, what is the value of λ[4]? If a smaller lambda is chosen, it may produce a tighter and more accurate PDF, or it may over-fit the data. Similarly, a larger λ may overcome over-fitting, but smooth out or remove critical qualitative features of the actual PDF. Also, without additional data or further ways to validate the produced KDE, there is no *a priori* way to determine if one value for λ has produced a better KDE than another. Effectively, the choice of λ is educated guesswork.

As a toy example, we show two combined Gaussians with different standard deviations and means estimated with different λ values in Figure (1.11). When the

[4]This can also be understood as the standard deviation σ.

value for λ is too small, we see over-fitting and unnecessary oscillations in the curve. When λ becomes too large the sharpness of the taller standard deviation is flattened out. It is clear that the best λ value would be somewhere around 0.4 for this example, *but in order to make that assessment we were required to know what the PDF is ahead of time.*

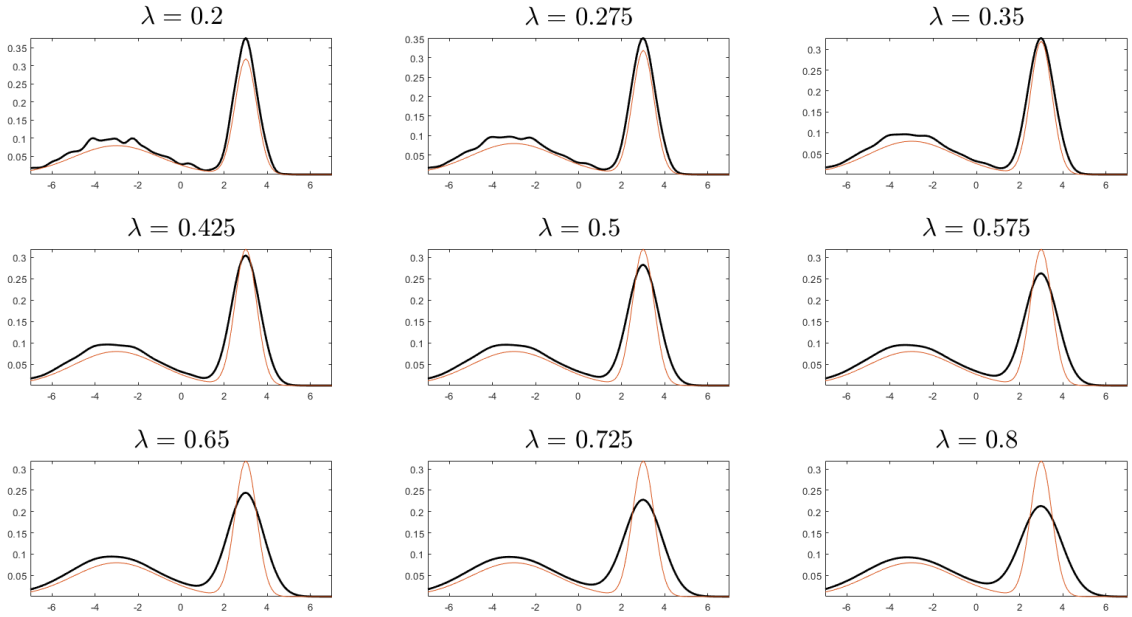


Figure 1.11. A function of two Gaussians $G(x) = g(x; \sigma = 2, \mu = -3) + g(x; \sigma = 0.5, \mu = 3)$ randomly sampled $N = 2000$ being estimated by increasing λ values from 0.2 to 0.8. The KDE uses the Gaussian kernel, K_λ.

Contrasting this with normalizing flows, the normalizing flows approach simply minimizes error for given data points and a respective forward pass thereby honing in on the best transformation function, $x = f(z)$[5]. This removes the aforementioned guesswork surrounding the value for λ. This error minimization, which is covered more extensively in Section (2.2.3.2.2), is done by minimizing the loss function for N number of samples[6]

$$\mathcal{L} = -\sum_{j=1}^{N} [\log (p_X(f(z_j)) + \log(f'(z_j))]$$

Here, only a poor choice of the neural net's structure or composed transformation functions can lead to a 'bad' approximation. The power of this technique is that a bad

[5]This is the same as $\mathcal{F}$ from before. Here I elected to keep the cleaner $f(z)$ notation.

[6]We again use the fact that $p_X(x) = p_Z(z)[f'(z)]^{-1}$ which was mentioned in Section (1.4.2). We make a critical note that we can directly compute $p_X(f(z))$ and $f'(z)$ because p_X is known and f is our NN.

NN will be made obvious by maintaining a high value for the loss function. This is a far more objective and clear metric than trying to have a qualitative feel for the validity of the KDE.

Lastly, above we assumed (somewhat by default) that the K_λ function would be a normal Gaussian kernel. However, there are infinitely many kernels one could choose from as seen in Figure (1.12). Without expansion, we remark that it should be clear that different kernel shapes will produce different KDEs. Determining which kernel then is best is again educated guesswork.

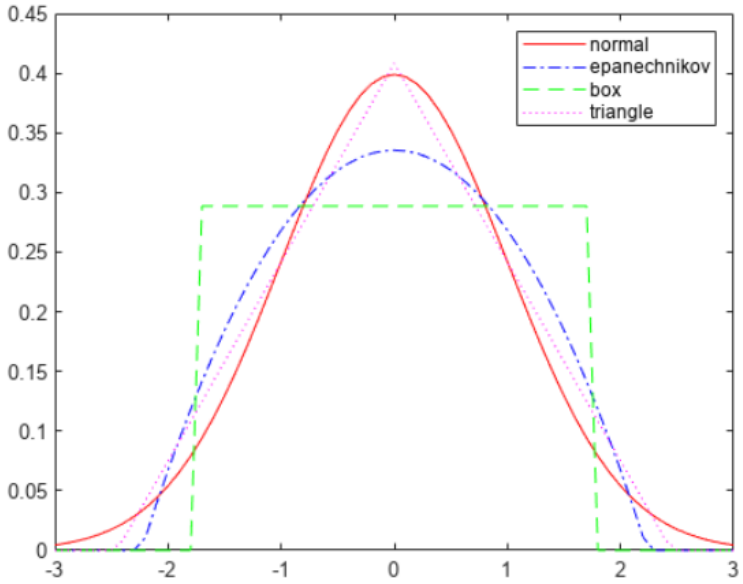


Figure 1.12. Several different examples of kernels (from MATLAB's Kernel Distribution documentation page).

As such, we can safely conclude that the normalizing flows technique, as far as accuracy is concerned, will be a superior technique and is generated completely by the provided data. We make a final note that the strength of the KDE however is its lack of computational overhead (a NN will almost always be more computationally expensive than an optimized algorithm). This is to say that KDEs should not be abandoned, simply that one should understand their limitations.

1.5 The Van der Pol (VDP) and Stochastic VDP (SVDP) Oscillators

To end the introduction sections we briefly discuss the Van der Pol (VDP) Oscillator[7], and we note that the VDP without added noise comes from the ODE

$$\ddot{y} - \mu(1 - y^2)\dot{y} + y = 0$$

By making the necessary substitutions we arrive at an equivalent system of ODEs in the form[8]

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= \mu(1 - x_1^2)x_2 - x_1 \end{aligned}$$

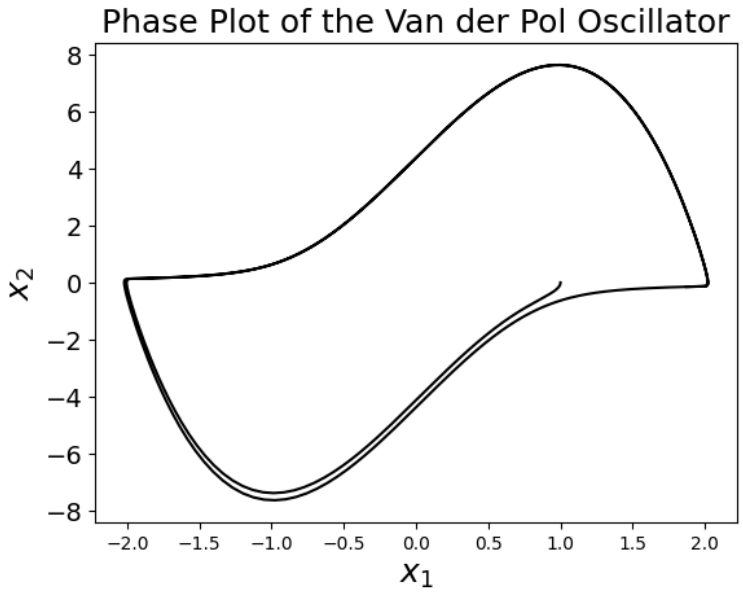


Figure 1.13. An example of the VDP with $\mu = 5$, ICs of $x_1(0) = 1,\ x_2(0) = 0$ and integrating from $t_0 = 0$ to $t_f = 20$. We can see that it smoothly converges to its non-linear limit cycle.

Now, if we wish to add noise to this system of ODEs, and thereby turn the VDP into a stochastic VDP, we then make two additions and our system is changed to

$$d\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \left(\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 0 \\ \mu(1 - x_1^2)x_2 \end{pmatrix} + \begin{pmatrix} 0 \\ \alpha \sin(\omega_d t) \end{pmatrix} \right) dt + \sigma \begin{pmatrix} 1 \\ 1 \end{pmatrix} dB_t$$

[7]The VDP has been well studied and one can find more information about it here[18].

[8]Here we simply let $x_1 = y$ and $x_2 = \dot{y} = \dot{x}_1$. By solving for $\ddot{y}$ then making substitutions we arrive at our system of ODEs.

where we note that the dB_t term is a Brownian motion process[7]. Also, given that we have added Brownian motion to both $\dot{x}_1$ and $\dot{x}_2$, we note σ controls the strength of the noise[9]. We show an example of the SVDP in Figure (1.14).

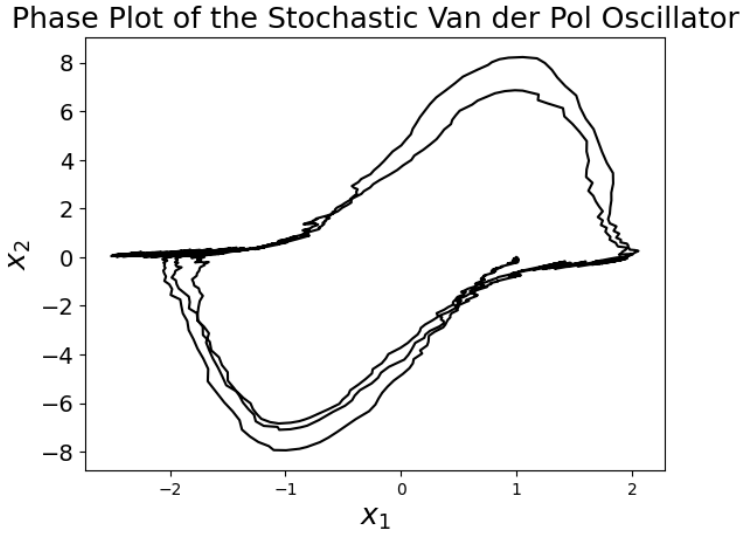


Figure 1.14. An example of the SVDP system of ODEs with $\mu = 5$, an IC of $x_1(0) = 1,\ x_2(0) = 0$ and integrating from $t_0 = 0$ to $t_f = 20$. Here $\alpha = 0.2$, $\omega_d = 8\pi$, and $\sigma = 0.3$.

[9]We note that a full discussion of Brownian motion and stochastic calculus, as well as their respective numerical methods, are out of scope for this thesis. My advisor, Dr. Curtis, handled the implementation of adding Brownian motion to a standard VDP integration scheme via Runge–Kutta method (SDE). The implementation of this process can be seen in the source code provided in Appendix A.

CHAPTER 2
BUILDING THE NORMALIZING FLOW NEURAL NETWORK

We first show the overhead required to simply learn how to use `PyTorch` and ML generally [15]. Then, we move on to thoroughly evaluate two basic examples.

2.1 MNIST Data Set and Learning `PyTorch`

The status quo (which is one I did not stray from) for first learning machine learning seems to be utilizing the MNIST dataset to train an ML model on how to correctly classify handwritten digits. An input of a handwritten number between 0 and 9 is provided, and the model attempts to correctly state what that image's digit is. The data set consists of 70,000 handwritten numbers that are split into two bins. The first is training data which contains 60,000 images, and the second is testing data containing 10,000 images[1].

`PyTorch` is an ML framework based on the library, `Torch`. `PyTorch` utilizes a `Python` interface. There are two types of arrays one can make, and there are two locations for those arrays to be stored where actions can be performed on them. A `PyTorch` tensor can be run off of a GPU or a CPU. If your machine does not have access to a GPU, `PyTorch` will default to running on the CPU. It should be noted however, that `numpy` arrays cannot run on the GPU. As such, in the referenced code there are quite a few instances where a `numpy` array is created, and then switched to a torch tensor, and vice versa because `Python` functions do not work with `Torch` inputs. Running code on a GPU is much faster; while running my own code I recorded a roughly nine times improvement in how long it took to run on a GPU compared to a CPU.

The MNIST ML algorithm I wrote has the following general structure, as do the normalizing flow models.

1. Importing required packages.
2. Shaping and verification of the input data.
    - For the MNIST data set ensuring a matrix was a long column vector.

[1]The MNIST data set was imported via `keras.datasets import mnist`.

- For normalizing flows, this meant creating a properly shaped `torch.tensor` of values randomly sampled from whichever distribution.

3. Creating device agnostic code (code that can run without a GPU if one is not available).
4. Building the training model's structure as a class, and creating an instance of the model. These classes store which kind of transformations to use, and the size and number of hidden layers for deep NNs.
5. Determining the loss function and the choice of optimizer. `PyTorch` offers several optimizers. I chose Adam, a type of Stochastic Gradient Descent (SGD)[2].
6. Shaping and classifying the output data.
    - For the MNIST dataset this meant understanding the value for each output node as a probability of the model's certainty that it was a given digit.
    - For normalizing flows, this was understanding the outputs were a collection of data points that ideally should look like a normal Gaussian distribution.
7. The training loop and loss curve. Here is where successive plots are generated to show both the visual and numerical outputs of the loss.
8. Model visualization and verification. After the model is done training, a final snapshot is shown alongside the loss. From here whatever testing is desired can take place. In the case of normalizing flows, this meant passing data back and forth through $\mathcal{F}$ and $\mathcal{F}^{-1}$ respectively to determine if we had command of the NN, and subsequently approximating the target's PDF.

The code for the MNIST model is provided in Appendix A.

[2]SGD optimizers detect that the loss is not improving and 'randomly kick' the parameter values landing them in a different local part of the subspace, thereby potentially finding a deeper minima to fall into, and thus creating a better model.

2.2 1D Normalizing Flow in PyTorch - Normal Gaussian to Normalized Double Hump Gaussian

The below details follow the general outline that was explained in Section (1.4) and framework from Section (2.1).

2.2.1 Package Imports

The line by line list of packages that were used in the code can be found by looking at the code itself found in Appendix A. However we make a special note. The `Pyro` package was not able to be used without some minor adjustments.

The `Pyro` package was used as it had built in bijective transformations. Some of them assumed the incoming vectors would be `numpy` arrays and would be running on the CPU. This required going through parts of the code line by line and forcing arrays to be sent to the GPU. An example is provided below.

```
def _searchsorted(sorted_sequence, values):
    """
    Searches for which bin an input belongs to (in a way that is
    parallelizable and amenable to autodiff)
    TODO: Replace with torch.searchsorted once it is released
    """
    values = values.to("cuda")
    XX = values[..., None]
    XX = XX.to("cuda")
    return torch.sum(values[..., None] >= sorted_sequence, dim=-1) - 1
```

As such, it was required that any machine that was utilized while running the code, would first need to replace three files which can be found in Appendix A. These changes did not change *what* the `Pyro` functions were doing, but simply allowed them to be run on the GPU[3].

[3]CUDA® (or Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs[13].

2.2.2 Parameter Choices and Pseudo-Data

Here we make a first critical choice. The spline transformation, which is explained in Section (2.2.3.2.1), assumes that a $[-B,-B]\times[B,B]$ box will be generated, where $\mp B$ are the lower and upper respective bounds of the input and output domains. In this case, the output domain is the domain for the double hump Gaussian, which is of the form

$$p_Z(z)=G_{dh}(z)=\frac{1}{2}\left[\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(z-\mu)^2}{2\sigma^2}}+\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(z+\mu)^2}{2\sigma^2}}\right]$$

The $1/2$ in front normalizes G_{dh} and makes $\int_{\mathbb{R}}G_{dh}(z)\,dz=1$. As such, $\mu=2$ and $\sigma=1$ with limits of $\mp B=\mp 7.5$ were chosen[4]. The respective input domain then becomes the domain of the normalized Gaussian.

Here is where a critical feature of the `Pyro` transformations must be understood. *Only data points, not discrete or continuous functions, may be entered as arguments to the* `Pyro` *transformations.* Hence, I was required to generate random data points from $G_{dh}(z)$. I did this by simply using `numpy`'s `np.random.normal(mu,sig,arr)` function. For each 'hump' I generated the same number of data points. This built the pseudo-data set

$$\mathcal{G}_{dh}=\{g_{10},g_{20},g_{11},g_{21},...,g_{1N},g_{2N}\}$$

From there, $\mathcal{G}_{dh}$ was transformed from a `numpy` array to a `torch.tensor`. An example of this can be seen in Figure (2.1) where to better show the separation $\mu=5$.

2.2.3 Normalizing Flow Model and Bijective Transformations

It is worth noting immediately that simply because a transformation is typically bijective, that does not mean its NN transformation is bijective. Observing a simple counterexample that follows a similar setup to Section (1.3) but removes the sigmoid activation function, first let

$$\vec{h}_1=[-1,0,1]$$
$$\Rightarrow \mathbf{A}\vec{h}_1+\boldsymbol{b}=\begin{bmatrix}a_{11}&a_{12}&a_{13}\\a_{21}&a_{22}&a_{23}\end{bmatrix}\vec{h}_1+\begin{bmatrix}b_1\\b_2\end{bmatrix}=\begin{bmatrix}-1&0&2\\1&0&-2\end{bmatrix}\vec{h}_1+\begin{bmatrix}3\\7\end{bmatrix}$$

[4] At these endpoints is $G_{dh}(\pm 7.5)=5.38e-8$.

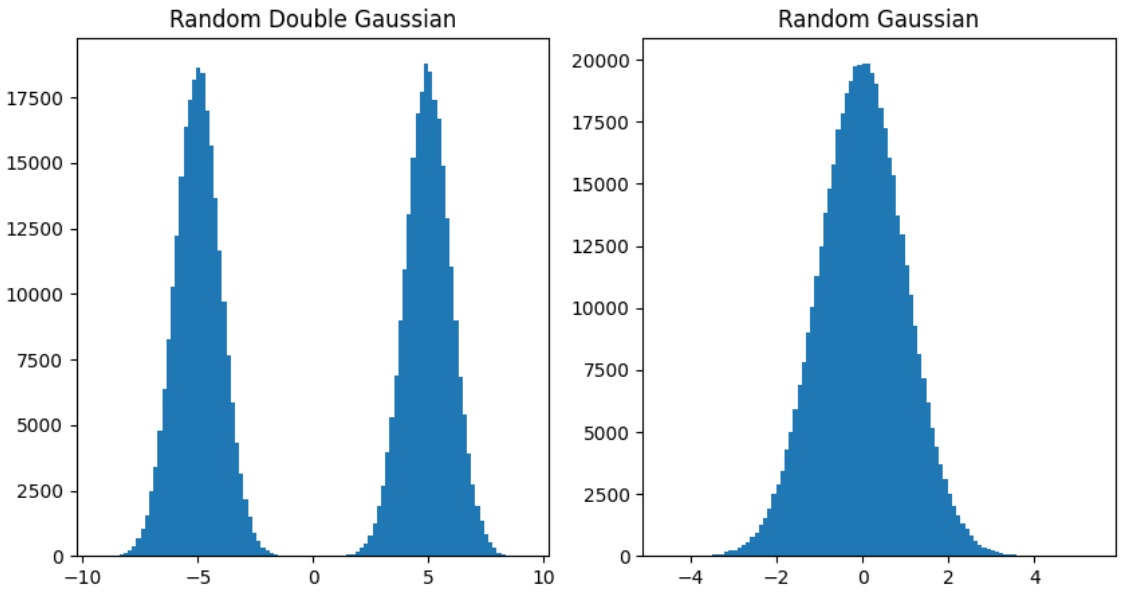


Figure 2.1. Randomly sampled double hump Gaussian and normal Gaussian.

Then, if we compute $\vec{h}_2 = \mathbf{A}\vec{h}_1 + \boldsymbol{b}$ we arrive at

$$-1 + 0 + 2 + 3 = 5 = h_{21}$$
$$-1 - 0 + 2 + 7 = 5 = h_{22}$$

clearly showing us that a standard linear transformation with random initial values for its parameters is not necessarily going to be bijective as we end up with $h_{21} = h_{22}$. We need to ensure that our transformations are monotonically increasing (and therefore analytically invertible) which is not trivial to do. Hence, we sought to utilize the `Pyro` library with ready made bijective NN functions instead of generating them from scratch which is an entire massive effort on its own[5].

2.2.3.1 The Normalizing Flow Model Structure

The `Pyro` tutorial explored multi-dimensional problems, but not problems showing multiple transformations strung together, as such making a deep NN required the unique definition of a model class, which I named `NormFlowModel_<name>` in order to properly store the different combinations of bijective transformations. The reason primarily being that each transformation needed to be uniquely stored in order to store that specific `Pyro` model's parameter values. This allowed for a list of transforms to be

[5]From the `Pyro` library I only used the linear and quadratic spline transformations in the displayed work of this thesis. However, I did make a functioning model using Householder, and Discrete Cosine Transformations as well which are not explored here as the splines were sufficient for our problem sets.

referenced by both the forward and backwards calls. This became essential later in being able to track the input of an individual point through $\mathcal{F}(s_0)$ or $\mathcal{F}^{-1}(x_0)$[6].

Here I pause to make a much needed clarifying note: the naming of `inv_transforms` and `transforms` is unintentionally confusing and it was not clear at the time of developing the code that the `transforms` calls take you from the normal Gaussian back to the target distribution. As such, `transforms`$\equiv \mathcal{F}^{-1}(x)$ and `inv_tranforms`$\equiv \mathcal{F}(s)$.

The outputs of the self-defined class, `NormFlowModel_<use_case>`, which are utilized heavily in the training loop are enumerated below.

1. `NormFlowModel` $\equiv$ a defined instance of the class which is then put on the GPU.
2. `inv_transforms` $\equiv$ a list of all of the transformation functions in their specific order, i.e., $f_1 \circ f_2 \circ ... \circ f_N$.
3. `transforms` $\equiv f_N^{-1} \circ f_{N-1}^{-1} \circ ... \circ f_1^{-1}$.
4. `base_dist` $\equiv$ the normalized Gaussian base distribution with $\sigma = 1$ and $\mu = 0$.
5. `flow_dist` $\equiv$ an object that can accept an array and perform necessary operations on it.

For (2) and (3) above, let sL be a linear spline transformation, and sq be a quadratic spline transformation. Then, a common NN structure used was (i.e., `inv_transforms`)

$$sq1(\mathcal{G}_{dh}) \mapsto sL1 \mapsto sq0 \mapsto sL0 \mapsto \mathcal{O}$$

where $\mathcal{O}$ is the output of the NN, and the goal is to have $\mathcal{O}$ fit a normal Gaussian distribution. If this is achieved, then we can use

$$p_Z(z) = p_X(f(z))[f'(z)]^{-1}$$

to make a good approximation the target's PDF. In the above we note that the output of the first transformation, a quadratic spline ($sq1$) which takes in the dataset $\mathcal{G}_{dh}$, is then what is placed into $sL1$, where the output of $sL1$ is placed into $sq0$, etc. This composite chain continues and eventually produces $\mathcal{O}$. This chaining together of transformations (i.e., functions) fits with the general normalizing flow theory as discussed in Section (1.4.1) and provided me necessary flexibility while dealing with the several examples shown throughout this paper.

[6]We use x_i to specify a point taken from the Gaussian distribution, and s_i from the 'signal' distribution. In this example the single hump Gaussian points are x_i while the signal is the double hump Gaussian, s_i.

As such, the entire process can be boiled down to (where $N \equiv$ normal Gaussian)

$$[\mathcal{F}(\mathcal{G}_{dh}) = \mathcal{O}] \approx \mathcal{G}_N$$

2.2.3.2 Bijective Transformations

2.2.3.2.1 Linear and Quadratic Splines - Neural Spline Flows

The paper Neural Spline flows written by Durkan et al., extensively outlines the quadratic spline process[7] [6]. The general idea is that a set of bijective quadratic equations are strung together as a spline meaning each segment shares its starting and ending points with the next segment, as well as its derivative. The input parameter, `count_bins`, determines how many spline segments will be generated, which is referred to by K in the paper. This provides each spline call with $3K - 1$ parameters, denoted by $\boldsymbol{\theta}$, to adjust. A visual of the above is shown in Figure (2.2) where the term knots refers to the shared points between spline segments.

$$\boldsymbol{\theta}_i = \{\boldsymbol{\theta}_w, \boldsymbol{\theta}_h, \boldsymbol{\theta}_\delta\}, \text{ where } w \equiv \text{ width, } h \equiv \text{ height, and } \delta \equiv \text{ derivative}$$

If we understand the knots and derivatives to be

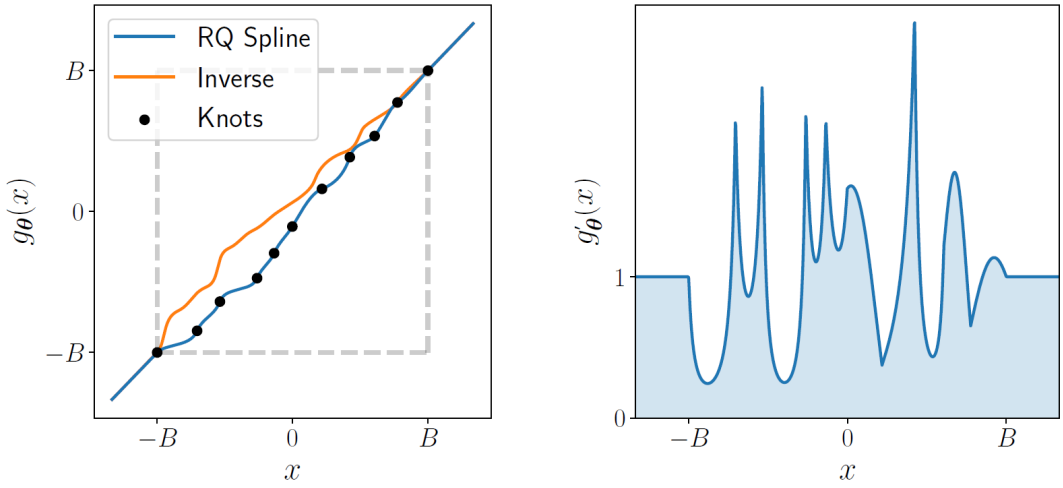


Figure 2.2. The figure presented in the Neural Spline Flows paper showing monotonic rational-quadratic transforms with $K = 10$ bins/segments with $K + 1$ knots and $K - 1$ derivative points (assumed to be 1 outside of the $[-B, -B] \times [B, B]$ box), as well as the spline's derivative [6].

[7] Understanding the quadratic case makes the linear case a trivial extension of the quadratic case and as such we omit a discussion of the linear case here.

$$\mathcal{K} = \left\{ \left(x^{(k)}, y^{(k)}\right) \right\}_{k=0}^{K},\ \left(x^{(0)}, y^{(0)}\right) = (-B, -B) \text{ and } \left(x^{(K)}, y^{(K)}\right) = (B, B)$$

$$\Delta = \left\{ \delta^{(k)} \right\}_{k=1}^{K-1}$$

and also letting per interval (and noting that $x^{(k)} < x^{(k+1)}$ and $y^{(k)} < y^{(k+1)}$)

$$s_k = \frac{y^{k+1} - y^k}{x^{k+1} - x^k}, \qquad \xi(x) = \frac{x - x^k}{x^{k+1} - x^k}$$

allows the expression for the rational-quadratic to be

$$\frac{\alpha^{(k)}(\xi)}{\beta^{(k)}(\xi)} = y^{(k)} + \frac{(y^{(k+1)} - y^{(k)})[s^{(k)}\xi^2 + \delta^{(k)}\xi(1-\xi)]}{s^{(k)} + [\delta^{(k+1)} + \delta^{(k)} - 2s^{(k)}]\xi(1-\xi)}$$

Lastly, we observe that the widths and heights, which become optimizable parameters are

$$w^{(k)} = x^{(k+1)} - x^{(k)}$$
$$h^{(k)} = y^{(k+1)} - y^{(k)}$$

Effectively, the various knots described by points in $\mathbb{R}^2$ are free to be adjusted within the $[-B, -B] \times [B, B]$ box thereby effecting the output of the transformation as directly impacting the knot position changes the general shape of the polynomial.

This rather simple structure has the main advantage that because it is completely generated by quadratic polynomials, its derivative and inverse (using the quadratic formula[REF to Appendix of theirs]) are analytically solvable and simple to compute numerically. Thus, computing the gradient for back propagation is just an exercise in messy but tedious calculus.

2.2.3.2.2 Architecture and Parameter Learning for Neural Spline Flows

The architecture used by the splines is a derivation of a coupling transform which can be seen in Figure (2.3) [6]. The linear function $g_{\boldsymbol{\theta}}(x_i) = \alpha x_i + \beta$ for this architecture is known as an additive or affine transformation [6]. The structure of this affine transformation serves to do two things. One, by shifting the data that is and is not split up during each training epoch, the problem of over-fitting is avoided. Two, it gives an entire NN an ability to learn the best parameters possible for a bijective mapping by not applying the NN to the data that is fed forward, but instead towards generating parameter values in $\boldsymbol{\theta}$. It is further worth mentioning that these NNs can be anything the user chooses (convolutional, recurrent, linear, etc.).

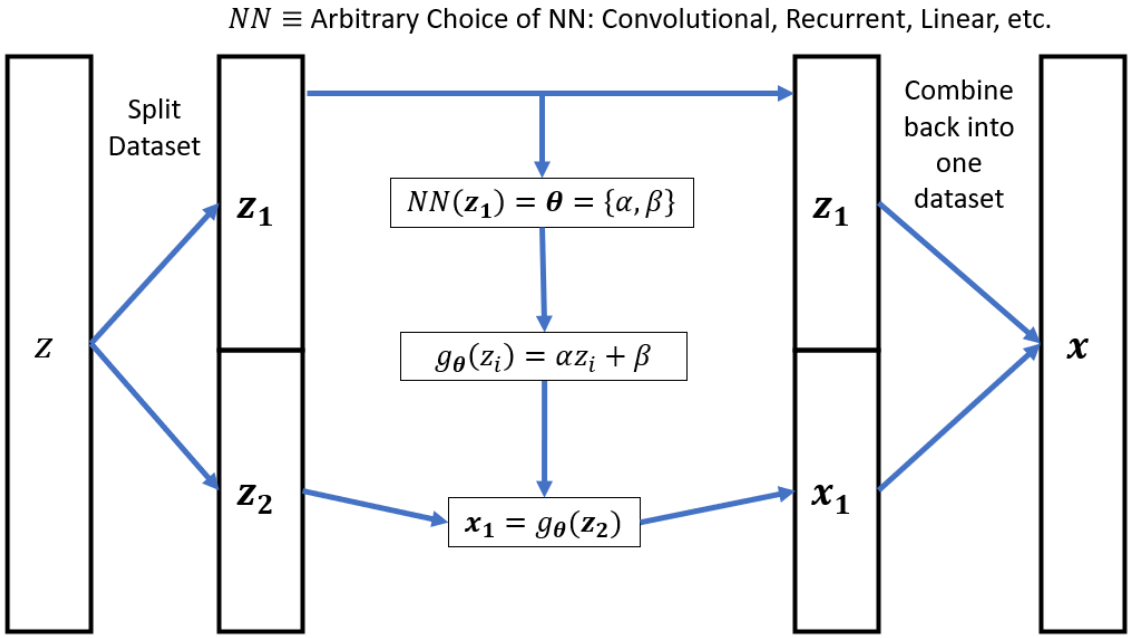


Figure 2.3. The dataset coming from the unknown PDF, z, is fed through a series of operations to become x, which is then compared to a normalized Gaussian distribution. By splitting up the dataset differently each time, overfitting is avoided. NN stands for any arbitrary choice of neural net (a few possible choices being: convolutional, recurrent, linear, etc.). In our current example, one could think of z as $\mathcal{G}_{dh}$.

In the case of Neural Spline Flows, the architecture is updated by making the widths, heights, and derivatives the parameters in $\boldsymbol{\theta}$ to optimize [6]. Those outputs are passed through activation functions (softplus and softmax functions) to remain in the $[-B, -B] \times [B, B]$ box, and then fed forward to be an approximation of $\mathcal{G}_N$. An updated structure can be seen in Figure (2.4), and is written specifically for the double hump Gaussian example being explained here.

Lastly, we explore how the loss function is calculated, and subsequently how parameters are updated. Because our target is simply a normalized Gaussian $G(x; \sigma = 1, \mu = 0)$, we know we can measure how well the data fits $G(x; 1, 0)$ by simply computing the Maximum Likelihood Estimation (MLE) where m is the number of elements in $\mathcal{G}_N$.

$$\arg\max p_X(\mathcal{O}) = \sum_{i=1}^{m} \log(p_X(f_{\boldsymbol{\theta}}(\mathcal{G}_{dh})))$$

But there is a problem. If we simply look at maximizing our MLE (or multiplying it by negative one thereby minimizing it) we will simply end up training the model to shove everything as close to zero as it can. Our bijective models in this

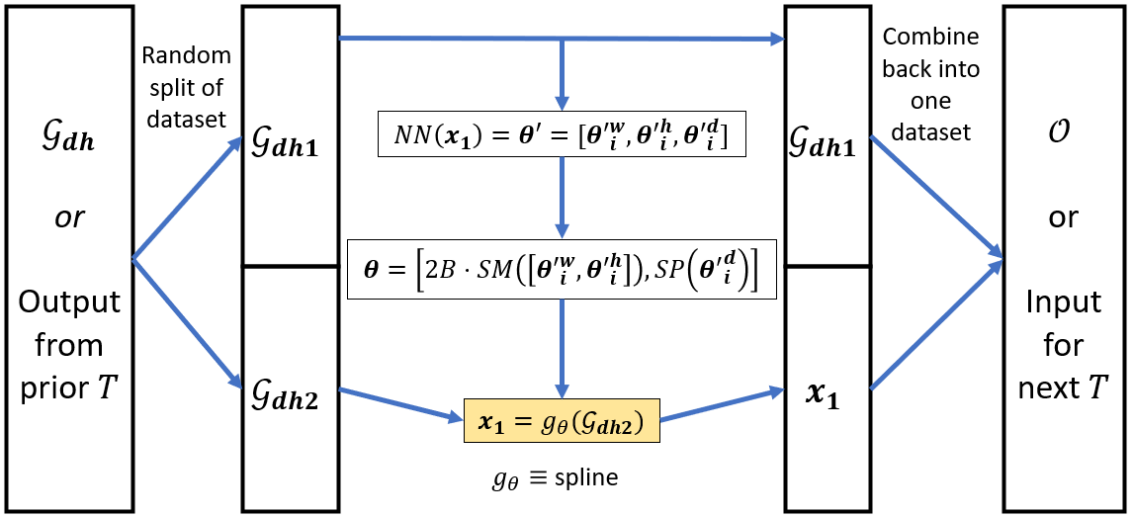


Figure 2.4. The dataset comes either from $\mathcal{G}_{dh}$ or the dataset output from the last transformation (T) that was applied. Again, NN is any arbitrarily chosen NN. The SM and SP functions are softmax and softplus respectively. Critically, the spline as discussed in Section (2.2.3.2.1) now takes the place of the previously linear $g_{\boldsymbol{\theta}}$.

case need to also preserve **probability mass** (and recall p_X is the probability for a normal Gaussian).

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= \log p_X(f_{\boldsymbol{\theta}}(\mathcal{G}_{dh})) + \log|\det Df_{\boldsymbol{\theta}}(\mathcal{G}_{dh})| \\ &= \log p_X(f_{\boldsymbol{\theta}}(\mathcal{G}_{dh})) + \log|f'_{\boldsymbol{\theta}}(\mathcal{G}_{dh})|\end{aligned}$$

If we take a closer look, while assuming we want to keep both probabilities equal to one (again recall $f(z) = x$) then in the 1D case

$$\begin{aligned}\int_{\mathbb{R}} p_Z(z)\, dz &= \int_{\mathbb{R}} p_X(x)\, dx \\ p_Z(z) &= p_X(f(z)) \cdot f'(z), \\ p_Z(z) &= p_X(f(z)) \cdot |f'(z)| \qquad \text{ensuring positive probabilities} \\ \log p_Z(z) &= \log[p_X(f(z))] + \log|f'(z)|\end{aligned}$$

In the above line, we know $f(z)$ is the same as $f_{\boldsymbol{\theta}}(\mathcal{G}_{dh})$ and $f'(z)$ is simply $f'_{\boldsymbol{\theta}}(\mathcal{G}_{dh})$. As such, by ensuring we preserve probability mass we end up at our loss function.

From here, the many robust algorithms available in `PyTorch` upon which `Pyro` is built can compute the back propagation necessary and update the parameters accordingly.

2.2.3.2.3 Householder and Discrete Cosine Transformations

In the code referenced in Appendix A, there are a few examples of instances where both Householder and Discrete Cosine transforms are used in the `norm_flow_model.py` file. If needed in future work, this note will hopefully serve as quick reference to an example of their usage.

2.2.3.3 The Training Loop

The learning rates for the iterations of the training loop were set to be $l_r = \{0.1, 0.01, 0.001\}$. In this way, the main outer loop would run the training loop for some predetermined number of epochs (typically 300), and then transition to the smaller learning rate and repeat the process. A larger learning rate helps to converge to an acceptable loss value more quickly, and once the algorithm is roughly in the correct ballpark of the best minima it can find, it then takes smaller steps for later iterations to find the most precise location for the best minima that it can.

The first variables worth noting are: `samp` which dictates the sampling size from the input array ($\mathcal{G}_{dh}$), and `dataset` which simply converts the `numpy` array ($\mathcal{G}_{dh}$) into a `torch.tensor`

```
samp=torch.Size([signal_data_points.shape[0],])
dataset=torch.Size([signal_data_points, dtype=torch.float).to(device)])
```

After these two items are established, the optimizer is selected. Here, I elected to use the Adam optimizer (a type of Stochastic Gradient Descent (SGD).

Then the training loop starts. First, the gradients of the optimizer are set to zero. Then, the loss is calculated from `loss=-flow_dist.log_prob(dataset).mean()`. Then, back propagation through the NN via `loss.backward()` occurs, and then `optimizer.step()` applies the learning rate to each of the parameters. Lastly, `flow_dist.clear_cache()` prevents overfill of data in the `flow_dist` object.

Appendix D shows the training loop in action, giving an idea of how the bijective map changes over several epochs and training rates.

2.2.3.4 Histogram Verification of Forward and Backwards Pass

In this section, the initial data from the target distribution, `XS`, and a randomly generated sample of normalized Gaussian data points, `XG`, both of the same size, are fed through the NN (where again `inv_transforms` is $f_1 \circ f_2 \circ ... \circ f_N$ and `transforms` is

$f_N^{-1} \circ f_{N-1}^{-1} \circ ... \circ f_1^{-1})$.

From normal Gaussian data points to a double hump Gaussian

```
XG = XG.float()
ForwardData = []
for ii in range(len(transforms)):
      if ii == 0:
            ForwardData = transforms[ii](XG.unsqueeze(1))
      else:
            ForwardData = transforms[ii](ForwardData)
```

From double hump Gaussian data points to a normal Gaussian

```
XS = XS.float()
BackwardData = []
for ii in range(len(inv_transforms)):
      if ii == 0:
            BackwardData = inv_transforms[ii](XS.unsqueeze(1))
      else:
            BackwardData = inv_transforms[ii](BackwardData)
```

Both of these, plotted as a histogram, show they at least qualitatively generate histograms that are representative of the correct respective distributions as can be seen in Figure (2.5). With this verification in place, we can be confident that we will have some success in recreating the target's PDF.

It is important to note before moving on that the histograms can be misleading. The approximated functions themselves should always be produced and analyzed to confirm that they match the solutions analytically.

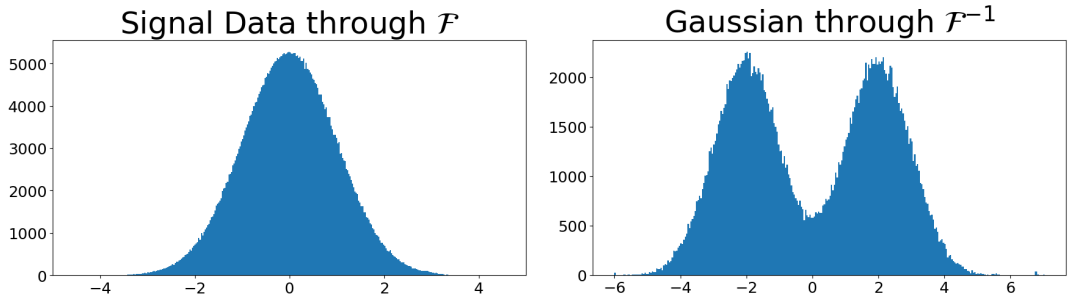


Figure 2.5. Histograms of the data points as they are passed through the NN in both the forward and backward directions.

2.2.3.5 Producing an Approximation of the Target PDF

Here we must be careful as to which values are being used. We recall again that: $x = f(z)$, the correct shape for $p_Z(z)$ analytically is the double humped normalized Gaussian, $p_X(x)$ is a known normal Gaussian PDF, and $f(z)$ is the learned bijective mapping

$$p_Z(z) = p_X(x)[f'(z)]^{-1}$$

Starting with $f(z)$, this is simply passing the z-axis through the `inv_transforms` calls just like was done in Section (2.2.3.4). Second, I simply used the `np.gradient` call to create a function for $f'(x)$. These two functions can be seen in Figure (2.6)[8]. Then,

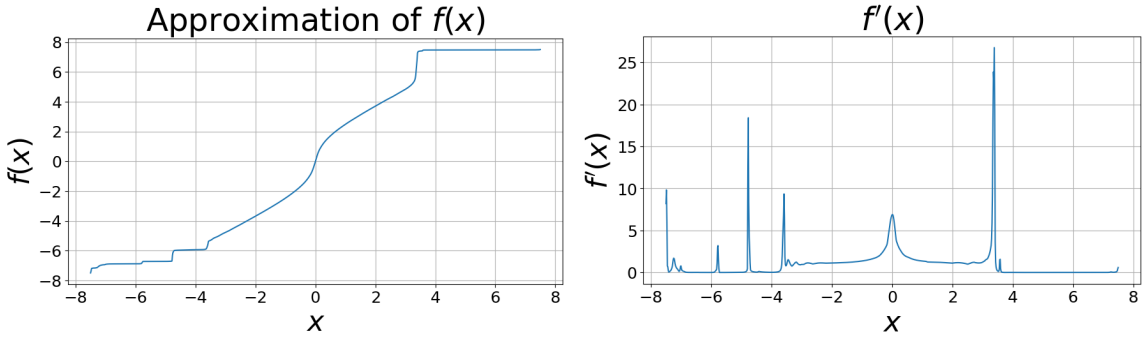


Figure 2.6. Left: The composite bijective function $f(x)$. Right: The first derivative of $f(x)$, calculating this is required to approximate the PDF for the double hump Gaussian.

with $p_X(x)$ and $f'(z)$ already known, we can go ahead and calculate the value for $p_Z(z)$ by multiplying p_X and $[f']^{-1}$ together. Putting this altogether gives the approximation PDF as shown in Figure (2.7). For this Gaussian to double hump Gaussian example, we see a tight fitting to the exact solution.

We note here that this is an obviously much better approximation to the function than what a KDE would be able to achieve. Even by a simple eyeball metric a comparison of Figure (2.7) to Figure (1.11) shows which technique is more accurate.

[8]In Figure (1.10) we see the exact solution and note that our approximation $f(x)$ in Figure (2.6) is qualitatively similar. We note that it struggles further away from zero, as here data points become sparse making it harder for the algorithm to properly learn.

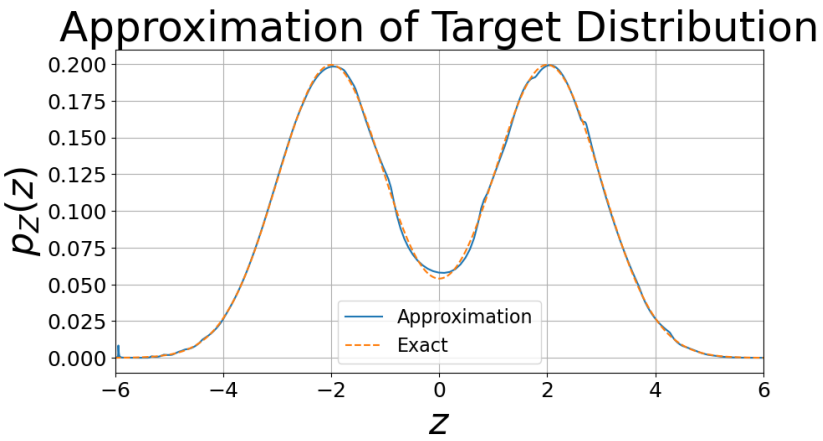


Figure 2.7. The final goal of the normalizing flows process - a function which is an analytical approximation of the target distribution.

2.3 Mapping a Gaussian to a Simple Signal via Normalizing Flows

To avoid repeating much of what was previously discussed in Section (2.2), only the deltas here will be covered as otherwise the process remains the same. We now move on to another example where our target distribution is less simple than a couple of normalized Gaussians. Because the initial application that led us to OM was EWT, it seemed appropriate to test the first 'non-trivial' problem on a Fourier transformed signal. For this I used

$$R[t_n] = \sin(2\pi t_n) + \sin(5\pi t_n) + \sin(7\pi t_n) + N_n$$

where $N_n \equiv$ noise uniformly sampled from $\mathcal{N} \in [0, 2]$.

2.3.1 Pseudo-Data Generation for the Target Distribution

In Figure (2.8) we can see the signal with some added noise. Also, in Figure (2.9) we see $\hat{R}[\omega]$ as the three sharp peaks of the three separate harmonics and the low amplitude jumbled frequencies of the noise.

As stated earlier, the `Pyro` models do not accept PDFs or KDEs. They only accept data samples. Therefore, in order to have an accurate data pool for the target $\hat{R}[\omega]$, I needed to pull samples from $\hat{R}(\omega)$. For this I used the process of **adaptive rejection sampling** [4]. In this process a curve, that can be randomly sampled easily like a Gaussian, is multiplied by some constant M with a chosen σ placing it strictly above the desired distribution curve in such a way that (letting $G(t;\sigma) \equiv$ Gaussian

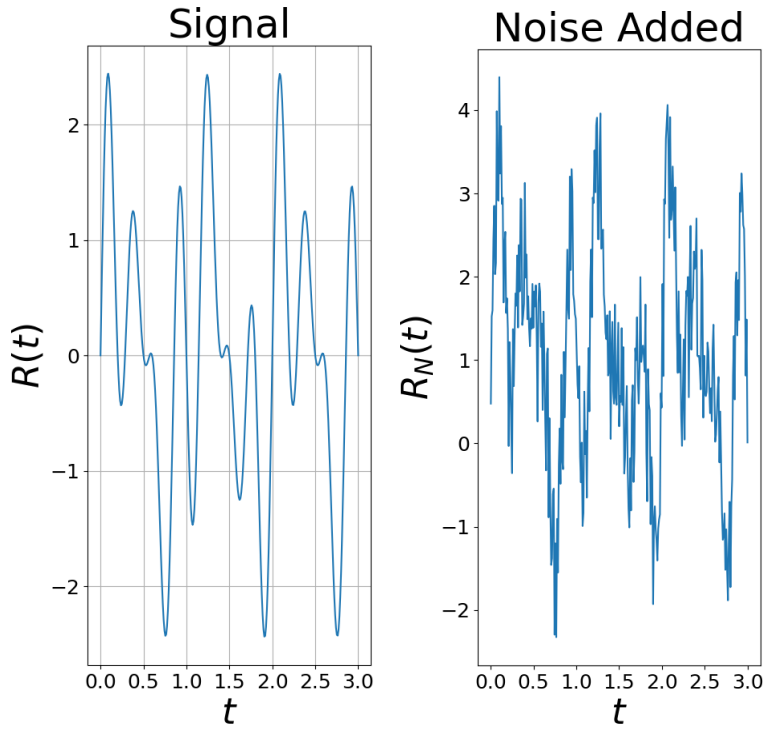


Figure 2.8. Left: Test signal $R[t]$ without noise. Right: Noise added to the test signal.

PDF and $H(t) \equiv$ random curve PDF to be made into a histogram)

$$MG(t;\sigma) > H(t)$$

Then, an element of the histogram is produced from $H(t)$ by first randomly selecting a value from the domain of H, which we will call $s^* \in H$ and then computing its probability, p, with respect to $MG(t;\sigma)$

$$p = \frac{H(s^*)}{MG(s^*;\sigma)}$$

Then, a value, u, is uniformly pulled from $\mathcal{U} = [0,1]$ and compared to p. The dataset $\mathcal{D}_H$ is then built as such

$$\begin{cases} p \geq u, & s^* \in \mathcal{D}_H \\ p < u, & s^* \notin \mathcal{D}_H \end{cases}$$

as a randomly generated set of samples from $H(t)$. In this way, by repeating the process a sufficiently large number of times, a histogram of $H(t)$ is generated. Ideally,

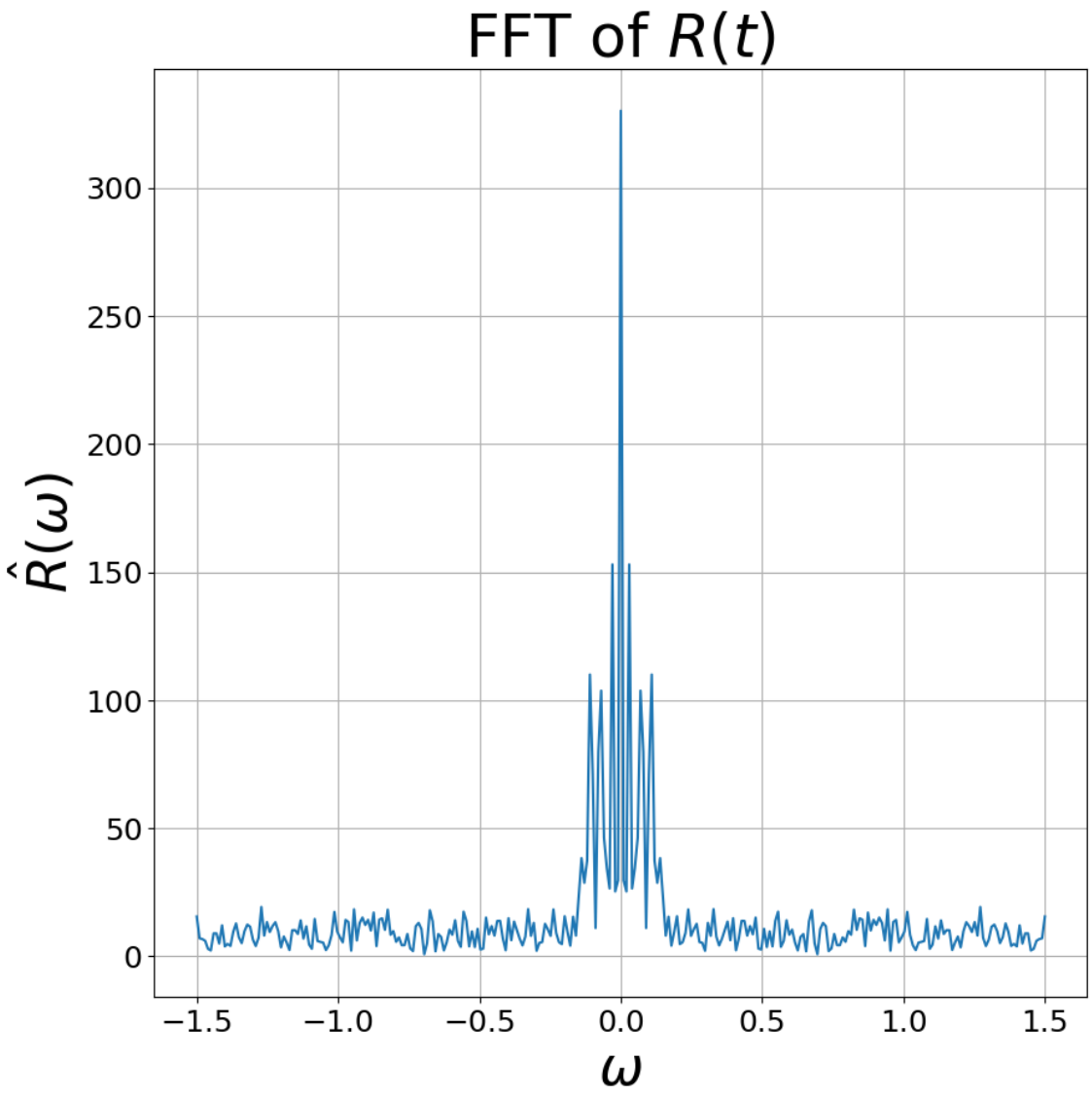


Figure 2.9. The FFT of our three harmonics with noise signal.

the Gaussian that is used should be as close to the signal as possible, or else it will take more attempts to make a sufficiently sized $\mathcal{D}_H$[9].

For this example, we treated $\hat{R}[\omega]$ as our $H(t)$ and turned it into a continuous function so it could be randomly evaluated for any ω in the frequency domain. This was easily completed with the `scipy.interpolate`'s `PchipInterpolator` function. We avoided using cubic splines as the data was already sufficiently dense and the splines

[9]Here we note that sufficient is subjective and depends completely on the application and problem. In our case of generating normalizing flows that help us be sure we are showing the proof of concept, it was beneficial to generate roughly 5×10^5 samples in $\mathcal{D}_H$.

created unnecessary and frankly inaccurate changes to the signal $\hat{R}(\omega)$ compared to $\hat{R}[\omega]$[10]. An example of this process and its output can be seen in Figure (2.10).

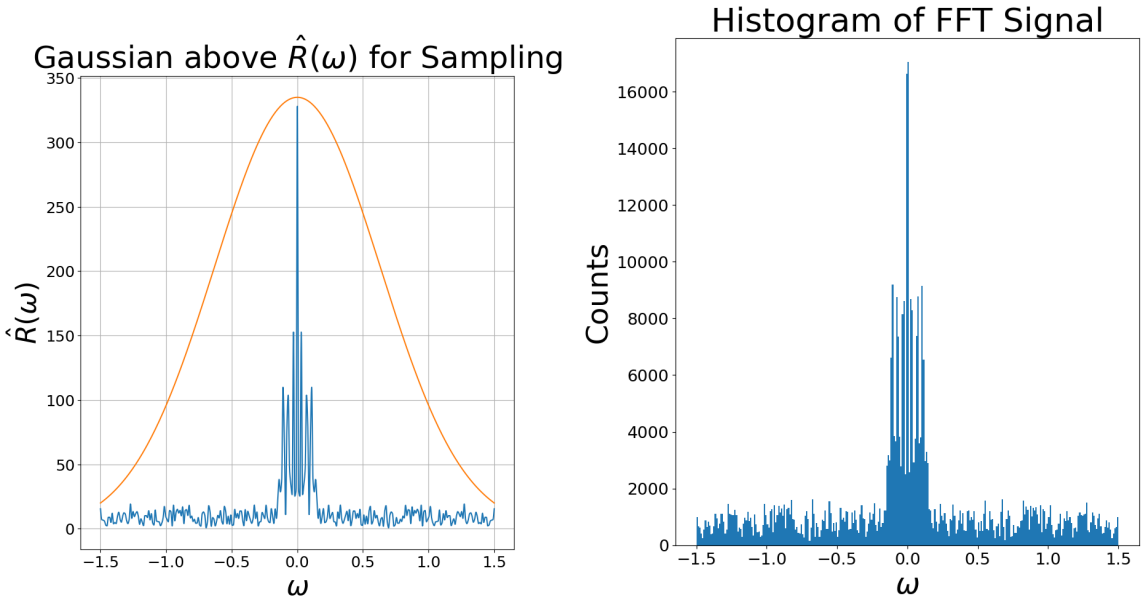


Figure 2.10. Left: The Gaussian strictly above $\hat{R}[\omega]$. Right: A histogram generated by adaptive rejection sampling of $\hat{R}(\omega)$.

2.3.2 Results of Mapping Harmonic FFT to a Normal Gaussian

This dataset is obviously not trivial to map to. The extremely fast peaks and random noise present the algorithm with a challenging task. However, it appears to have done quite well in our respective area of interest, namely, mapping to the three harmonic peaks in the FFT as seen in Figure (2.11).

We now make two interesting notes. First, due to the symmetry of a normal Gaussian, we came to the natural conclusion that a symmetric curve would be easier to map to than a non-symmetric curve. Comparing Figure (2.11) to Figure (2.12), this seems to be a fair assumption. Second, we note that in many of the runs there tends to be a persistent problem that occurs without a clear pattern. Namely, numerical 'spikes' will occur at the tail ends of the approximated distributions. Below in Figure (2.13) we see an example of this. Clearly the main area of concern, namely the harmonic peaks near the center are well mapped to. However there would need to be some kind of manual adjustment made to the plots to avoid these large spikes at the ends.

[10]To avoid any confusion, these splines are not in any way related to the bijective transformation splines that were discussed in Section (2.2.3.2.1).

Lastly, for continuity we show a plot of $f(z)$ and $f'(z)$ in Figure (2.14) to show clearly that this is a task best left to numerical techniques like normalizing flows.

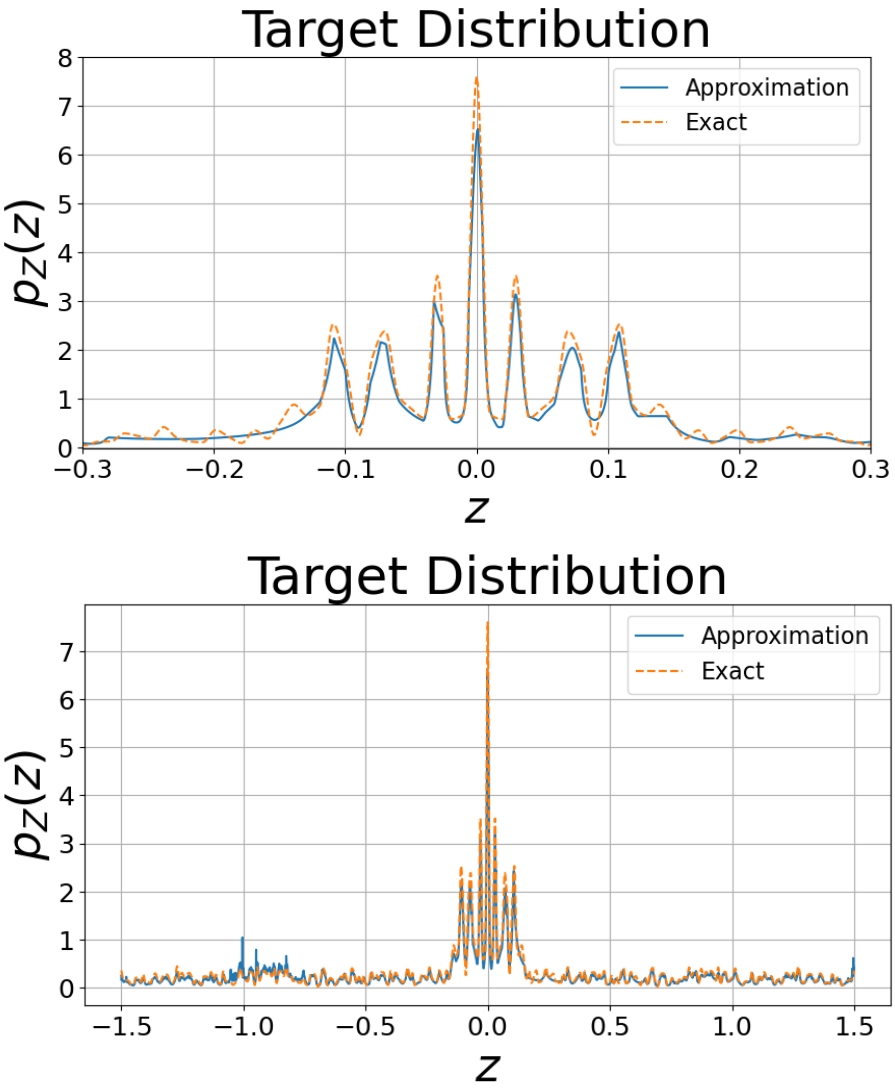


Figure 2.11. The approximation, $p_Z(z)$, in blue, and the exact solution, $\hat{R}(\omega)$, where the histogram data was pulled from in dashed orange. Both plots are of the same information, and the top plot is a zoom in of the central area from $[-0.3, 0.3]$.

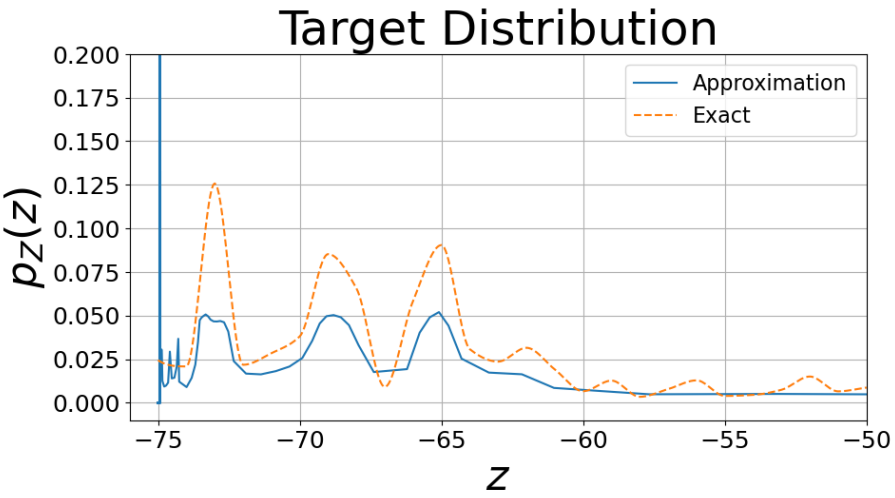


Figure 2.12. Here we see an attempt to map to a non-symmetric domain for otherwise the exact same signal. As is obvious, this seems to be a poor mapping, especially compared to Figure (2.11).

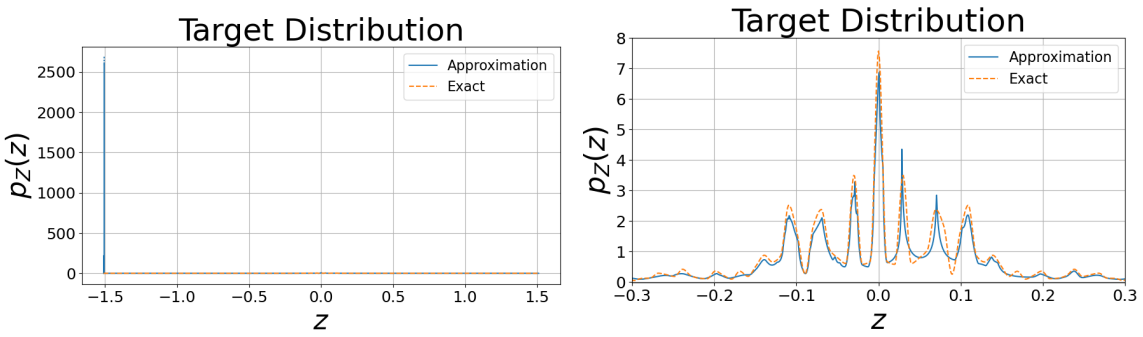


Figure 2.13. These images are from the same plotted arrays. The left image is plotted as is and the right image is zoomed in around the origin and limited to the height of the zero center peak.

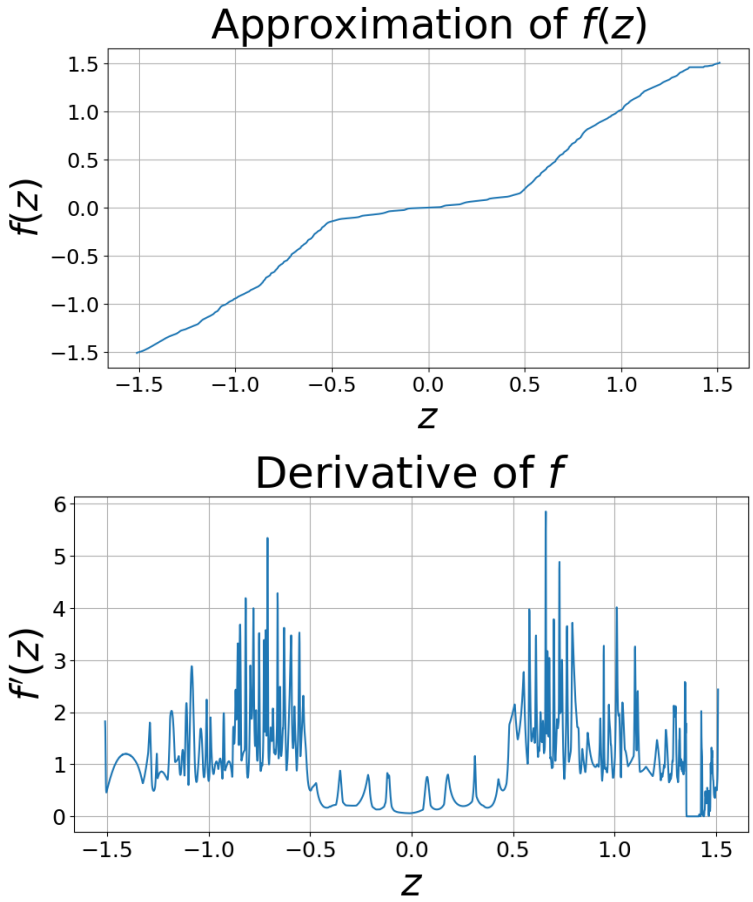


Figure 2.14. The learned $f(z)$ and $f'(z)$ for the three harmonics to normalized Gaussian problem.

CHAPTER 3

RESULTS

We first show a novel technique that Dr. Curtis and I developed and its results applied to a more complicated ECG signal[1]. We then explore how well this technique functions when applied to the Stochastic Van der Pol Oscillator (SVDP) which was introduced in Section (1.5).

3.1 Multi-Otsu Theory and its Application to EWT

3.1.1 Multi-Otsu on a Continuous PDF

We suppose that we are given a scalar data set $\{x_j\}_{j=1}^{N_s}$ where $x_j \sim \mathbb{P}_u$ is sampled from an absolutely continuous distribution $\mathbb{P}_u$ with affiliated density $p_u(x)dx$. We further imagine that the data is well described as a collection of (N_c+1)-*segments* with affiliated *threshholds* $\{k_l\}_{l=0}^{N_c+1}$ with $k_0 = -\infty$ and $k_{N_c+1} = \infty$ such that the l^{th} segment has $N_{c,l}$ members which satisfy one of the inequality:

$$k_l < x_j < k_{l+1}.$$

To determine how best to choose the threshholds $\{k_1\}_{l=1}^{N_c}$, following Otsu, we define the segment probabilities p_l where

$$p_l = \int_{k_l}^{k_{l+1}} p_u(x)dx, \qquad l = 0, \cdots, N_c$$

and the conditional averages μ_l such that

$$\mu_l = \frac{1}{p_l}\int_{k_l}^{k_{l+1}} x p_u(x)dx.$$

We then have the identities/constraints

$$\sum_{l=0}^{N_c} p_l = 1, \qquad \sum_{l=0}^{N_c} \mu_l p_l = \mu_u, \qquad \mu_u = \int_{\mathbb{R}} x p_u(x)dx.$$

Likewise, we can define the conditional variances σ_l^2 so that

$$\sigma_l^2 = \frac{1}{p_l}\int_{k_l}^{k_{l+1}} (x-\mu_l)^2 p_u(x)dx,$$

[1]This is the same ECG signal that can be found in the EWT Toolbox which is downloadable from MATLAB[9].

which has the corresponding constraint that

$$\sum_{l=0}^{N_c} p_l \left(\sigma_l^2 + \mu_l^2\right) = \sigma_u^2 + \mu_u^2, \qquad \sigma_u^2 = \int_{\mathbb{R}} (x - \mu_u)^2 p_u(x) dx.$$

In Otsu, we seek to maximize the *between-group* variance σ_B^2 defined as

$$\begin{aligned}
\sigma_B^2 =& \sum_{l=0}^{N_c} p_l \left(\mu_l - \mu_u\right)^2 \\
=& \sum_{l=0}^{N_c} p_l \mu_l^2 - \mu_u^2 \\
=& \sigma_u^2 - \sum_{l=0}^{N_c} p_l \sigma_l^2 \\
=& \sigma_u^2 - \sigma_W^2,
\end{aligned}$$

where the *in-group* variance σ_W^2 is defined to be

$$\sigma_W^2 = \sum_{l=0}^{N_c} p_l \sigma_l^2.$$

Thus we can see the optimization problem as either one in which we want each segment's average maximally separated from the total distribution average, or we want to minimize the conditionally weighted segment variances, thereby generating well defined clusters or segments.

We can also look at trying to find the critical points of σ_W^2 with respect to k_m which leads to the need to solve the equation

$$\partial_{k_m} \left(\int_{k_{m-1}}^{k_m} \left(x - \mu_{m-1}\right)^2 p_u(x) dx + \int_{k_m}^{k_{m+1}} \left(x - \mu_m\right)^2 p_u(x) dx \right) = 0.$$

We find that, assuming $p_u(k_m) \neq 0$ and $\mu_m \neq \mu_{m-1}$, that we have critical points when

$$k_m = \frac{1}{2}\left(\mu_{m-1} + \mu_m\right) = \frac{1}{2}\left(\frac{\int_{k_{m-1}}^{k_m} x p_u(x) dx}{\int_{k_{m-1}}^{k_m} p_u(x) dx} + \frac{\int_{k_m}^{k_{m+1}} x p_u(x) dx}{\int_{k_m}^{k_{m+1}} p_u(x) dx} \right)$$

This is a properly wicked system of coupled equations for the general case.

That said, if we restrict ourselves to the binary classification problem, then we only have to solve (dropping the subscript on k_1 for brevity) the equation

$$k = G(k),$$

where

$$G(k) = \frac{1}{2}\left(\frac{\int_{-\infty}^{k} xp_u(x)dx}{\int_{-\infty}^{k} p_u(x)dx} + \frac{\int_{k}^{\infty} xp_u(x)dx}{\int_{k}^{\infty} p_u(x)dx}\right).$$

Thus, the partition problem can now be rephrased as the search for fixed points of the function $G(k)$.

To build intuition here, we first show that if $p_u(x)$ is symmetric around its mean, or $p_u(\mu + x) = p_u(\mu - x)$, then $k_1 = \mu$. To see this, we see that using the substitution $\tilde{x} = x - \mu$ that we can readily rewrite this as

$$G(k) = \mu + \frac{1}{2}\tilde{G}(k - \mu),$$

where

$$\tilde{G}(k) = \frac{\int_{-\infty}^{k} xp_u(\mu + x)dx}{\int_{-\infty}^{k} p_u(\mu + x)dx} + \frac{\int_{k}^{\infty} xp_u(\mu + x)dx}{\int_{k}^{\infty} p_u(\mu + x)dx}.$$

Symmetry in p_u around μ then implies that $\tilde{G}(-k) = -\tilde{G}(k)$, so that $\tilde{G}(0) = 0$ and therefore $k = \mu$ is a fixed point of $G(k)$.

On the other hand, if we have asymmetry in our distribution, suppose for $n, m > 2$ that we know

$$p_u(x) \sim \beta x^{-m},\ x \to -\infty,\ p_u(x) \sim \alpha x^{-n},\ x \to \infty.$$

Then we get

$$k_1 \sim \frac{1}{2}\left(\mu_u + \frac{m-1}{m-2}k_1\right),\ k_1 \to -\infty,$$

which is easily solvable so long as the solution is consistent with the assumption that $k \to -\infty$. Likewise, taking the other limit gets us that

$$k_1 \sim \frac{1}{2}\left(\mu_u + \frac{n-1}{n-2}k_1\right),\ k_1 \to \infty,$$

and therefore, since μ_u must have a fixed sign, we see that we can only have one solution or the other depending on circumstances. We note that if p_u decays faster than any polynomial power, we should get the limiting solution $k_1 \sim \mu_u$ so long as $|\mu_u| \gg 1$, thereby making our approximate solution consistent with our asymptotic assumptions. In general then, we see Otsu's method is only ever going to tell us something different than partitioning around the average in strongly skewed circumstances.

That said, with regards to Otsu's method, we should not expect our normalizing flow to also naturally learn where partitions are, at least not without further guidance. To see why not, if we look at the fixed-point equation $k = G(k)$ and then apply the mapping f so that we have

$$z_k = f \circ G \circ f^{-1}(z_k),\ z_k = f(k),$$

then unless G has a symmetry such that

$$f \circ G = G \circ f,$$

then partitions do not immediately map to partitions.

We can even take this a step further by noting that

$$\begin{aligned}
\mu_x &= \int x p_u(x)\ dx \\
&= \int \mathcal{F}^{-1}(z) p_b(z)\ dz \\
&\neq \mathcal{F}^{-1}\left(\int z p_b(z)\ dz \right) \\
&\neq \mathcal{F}^{-1}(\mu_z)
\end{aligned}$$

We can be certain that generally averages will not map to averages. The only way an average will map to an average is if the distributions are symmetric about the average (i.e. $\mathcal{F}(k_g^*) = k_s^*$ for the single and double hump Gaussian example earlier).

3.1.1.1 Root Finding Theory for Multi-Otsu

Assuming that the normalizing flow helps us find better approximations of $p_u(x)$, we still need to figure out where the thresholds are. This problem comes down to solving the root-finding problem

$$G_m(k_{m-1}, k_m, k_{m+1}) - k_m = 0,$$

where

$$G_m(k_{m-1}, k_m, k_{m+1}) = \frac{1}{2}\left(\frac{\int_{k_{m-1}}^{k_m} x p_u(x) dx}{\int_{k_{m-1}}^{k_m} p_u(x) dx} + \frac{\int_{k_m}^{k_{m+1}} x p_u(x) dx}{\int_{k_m}^{k_{m+1}} p_u(x) dx} \right).$$

If we use any root-finding approach based on using a Jacobian, we see that that Jacobian is necessarily a tridiagonal matrix with, for $1 \leq m \leq N_c$, the entries

$$\partial_{k_{m-1}} G_m = \frac{p_u(k_{m-1})}{2p_{m-1}} \left(\mu_{m-1} - k_{m-1} \right),$$

$$\partial_{k_m} G_m = \frac{p_u(k_m)}{2} \left(\frac{k_m - \mu_{m-1}}{p_{m-1}} + \frac{\mu_m - k_m}{p_m} \right) - 1,$$

and

$$\partial_{k_{m+1}} G_m = \frac{p_u(k_{m+1})}{2p_m} \left(k_{m+1} - \mu_m \right),$$

where we keep in mind that $k_0 = -\infty$ and $k_{N_c+1} = \infty$. Numerical quadrature schemes via `SciPy`'s 'quad' and root finding via `SciPy`'s optimize routines should take care of the rest. An example of the code that performs this root finding is provided in Appendix A.

3.1.1.2 Applying Multi-Otsu to EWT

Further details of this section can be found in Appendix G, but a broad overview is presented here. Assume a signal $S[t] \in \mathbb{R}$. We wish to find n number of critical boundary points for $|\hat{S}[\omega]| \in [0, \pi]$ as is done in EWT[2]. We make a note and recall that in EWT boundaries are currently determined by using the previously discussed Gaussian scalespace method in Section (1.2.2).

We propose a novel technique to calculate EWT's boundaries, which we will call the Otsu Normalizing Flow Method (ONFM).

1. We take some signal or time series data $S[t]$, and retrieve $\hat{S}[\omega]$.
2. Similarly to EWT, we then rescale ω to be $\omega \in [-\pi, \pi]$.
3. Samples are gathered using adaptive rejection sampling as done in Section (2.3.1) for $[0, \pi]$ and duplicated to create a symmetric dataset.
4. We complete a normalizing flows NN and get an approximation $\hat{S}_A[\omega] \approx \hat{S}[\omega]$.
5. We interpolate $\hat{S}_A$ to create a continuous function[3].
6. Letting $\bar{\omega} \in [0, \pi]$ we find the n number of breaks in the continuous curve $\hat{S}_A(\bar{\omega})$ using the Multi-Otsu method as explained in Section (3.1.1) with a twist - starting with higher frequencies, some percent of the total area under the interpolated curve is removed.
7. These breaks are then fed into the EWT Toolbox code as a substitute for other existing methods (which includes the Gaussian scalespace method).
8. The best reconstruction error produced is observed and judged against the output of EWT's Gaussian scalespace method.

3.1.2 Examples of Applying ONFM

3.1.2.1 ONFM on an ECG Signal

Determining the number of modes for this signal was first done by running the original EWT code on the ECG signal to initially remove the burden of determining the number of required boundaries[4]. As such, we were given the number of breaks that would be required as 46. We followed the process as outlined in Section (3.1.1.2) and Appendix G with slight alterations being that we removed none of the signal (perc=100) and $k = 46$, effectively eliminating those for loops.

[2]In EWT and in this technique the domain of the `FFT` is adjusted to be from zero to π before any of the EWT or neural net algorithms are applied.

[3]The `scipy` library used has three options: linear, quadratic, and cubic. All were tested simultaneously with linear providing the best results.

[4]This is listed as `sig4` in the Toolbox tests example set.

During the reconstruction of the signal unaltered EWT[5] produced an error of $E = 4.9960e^{-16}$. Using the breaks as determined by ONFM yielded a similar result of $E = 7.2164e^{-16}$. The numerical values of the breakpoints used can be found in Appendix F.

The EWT components however were drastically different qualitatively. From the unaltered EWT case, there were five main modes, and the rest of the 42 modes captured the tiny noise in the signal. While looking at the components produced by ONFM however, almost every mode appeared to have defined character and shape. Many appeared be groups of duplicate qualitative wavelets. This could hint at less breaks being required if using ONFM. These images can be seen in Appendix F.

3.1.2.2 ONFM Applied to the SVDP

Initial conditions for all SVDP runs were $x_1(0) = 1$ and $x_2(0) = 0$.

3.1.2.2.1 Low Noise SVDP

We wanted to test how well ONFM might work on a stochastic dynamical system. As such, we used the SVDP as outlined in Section (1.5). We set: $t_0 = 0,\ t_f = 20,\ \sigma = 0.05,\ \mu = 1.1$, and $\alpha = 1.0$. These parameters yielded a SVDP that is was still recognizable as a VDP Oscillator as can be seen in Figure (3.1).

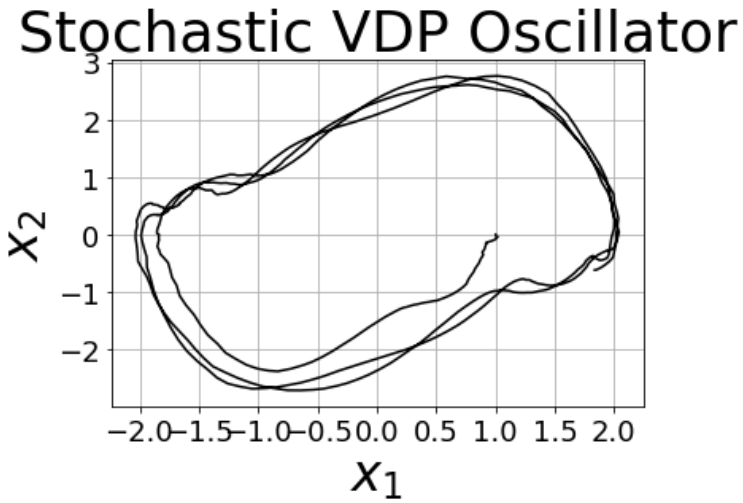


Figure 3.1. A low noise SVDP that was used in ONFM.

[5]Unaltered here means the following settings in the EWT code were chosen: `params.SamplingRate = -1; params.wavname = 'littlewood-paley'; params.globtrend = 'none'; params.degree=6; params.reg = 'none'; params.lengthFilter = 10; params.sigmaFilter = 1.5; params.detect = 'scalespace'; params.typeDetect='otsu'; params.N = 4; params.completion = 0; params.InitBounds = [2 25]; params.log=0; subresf=1; InitBounds = params.InitBounds.`

Following the algorithm as outlined in Appendix G, we produced the results as can be seen in Table (3.1). We make a critical note that this algorithm had to be run independently for the x_1 and x_2 datasets. Namely, the data was generated, saved, and then imported to be used by the NN to ensure both x_1 and x_2 datasets came from the same instance.

Comparing Unaltered EWT vs EWT with ONFM (low noise)		
Item	Unaltered EWT	EWT w/ ONFM
Number of Boundaries for: x_1	14	2
Reconstruction Error for: x_1	2.2204e-15	1.3323e-15
Number of Boundaries for: x_2	15	2
Reconstruction Error for: x_2	2.2204e-15	1.3323e-15

Table 3.1. The results of running the ONFM algorithm and replacing the boundaries values in the `EWT1D` algorithm for the low noise SVDP.

In both cases the results are comparable and our break-points at least allow for the ability to reconstruct the signal accurately.

We now highlight the EWT components that make up this signal through both approaches. First, the x_1 and x_2 time series data can be seen in Figure (3.2). Next, we

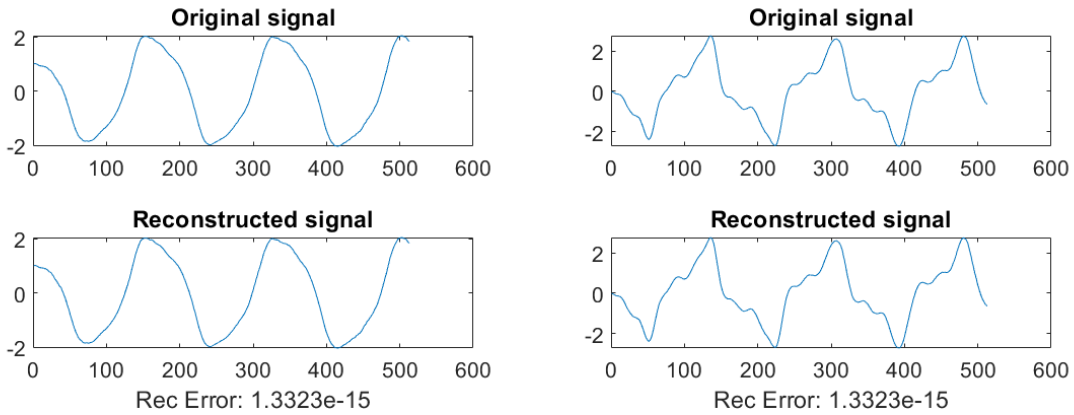


Figure 3.2. The $x_1(t)$ (left) and $x_2(t)$ (right) plots for the low noise SVDP. Both of these plots are the reconstructions done by ONFM.

see the EWT components produced by ONFM for both $x_1(t)$ and $x_2(t)$ in Figure (3.3).

The $x_1(t)$ EWT components seem to effectively allow for one main EWT component to almost completely be able to reconstruct the signal, and the next two components appear to fill in the small amount of noise. Curiously however, three unique modes appear to reconstruct the $x_2(t)$ signal without any EWT components with small 'noise fillers'. We are also missing any main mode that appears as if it were a best fit line to the signal as is present for the x_1 case. Next, to avoid clutter, the EWT components

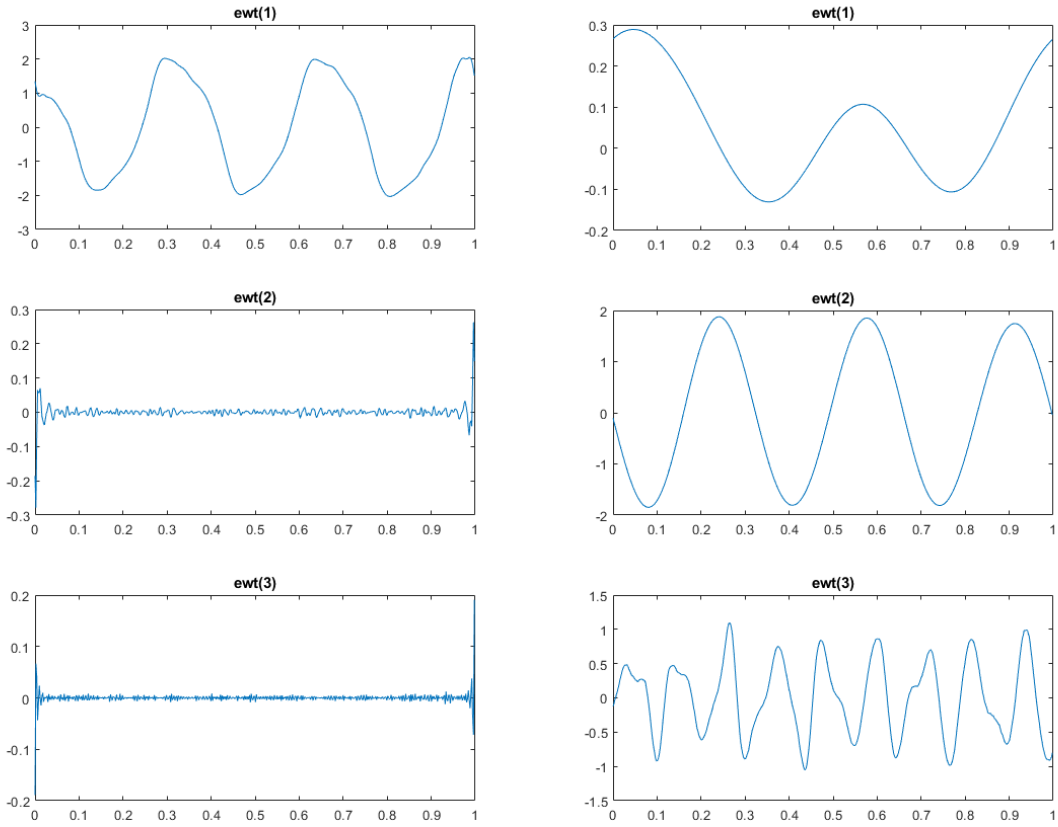


Figure 3.3. The low noise EWT components from the break-points produced by ONFM. Left: x_1. Right: x_2.

produced by running EWT unaltered can be found in Appendix H.

3.1.2.2.2 High Noise SVDP

We then tested how well ONFM might work on a nosier stochastic dynamical system. As such, we again used the SVDP as outlined in Section (1.5) and set:

$t_0 = 0,\ t_f = 6,\ \sigma = 2,\ \mu = 1.0$, and $\alpha = 20$. These parameters yielded an SVDP that looked like chaotic nonsense and can be seen in Figure (3.4).

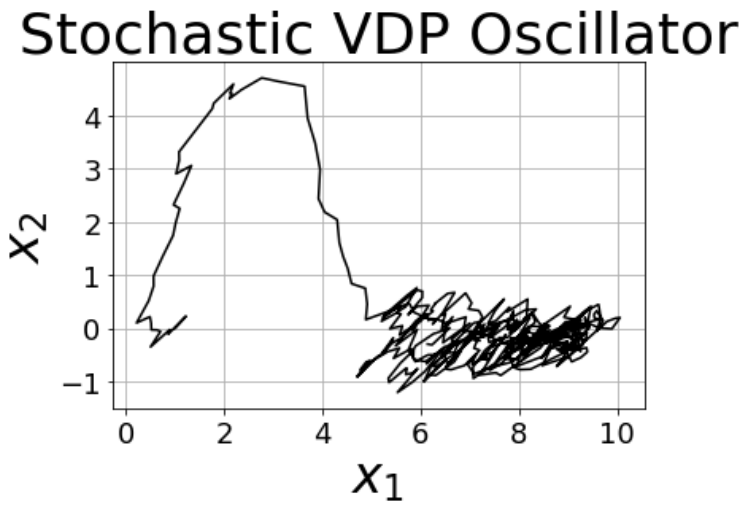


Figure 3.4. The high noise SVDP $x_1(t)$ plotted with $x_2(t)$.

We again plot the EWT components as well as the reconstructions for both $x_1(t)$ and $x_2(t)$. In Figure (3.5) we see the reconstructions. In Figure (3.6) we see the respective components. Similarly to the low noise SVDP, we show the EWT components from the unaltered EWT in Appendix H. Lastly, Table (3.2) shows the number of breaks and reconstruction errors.

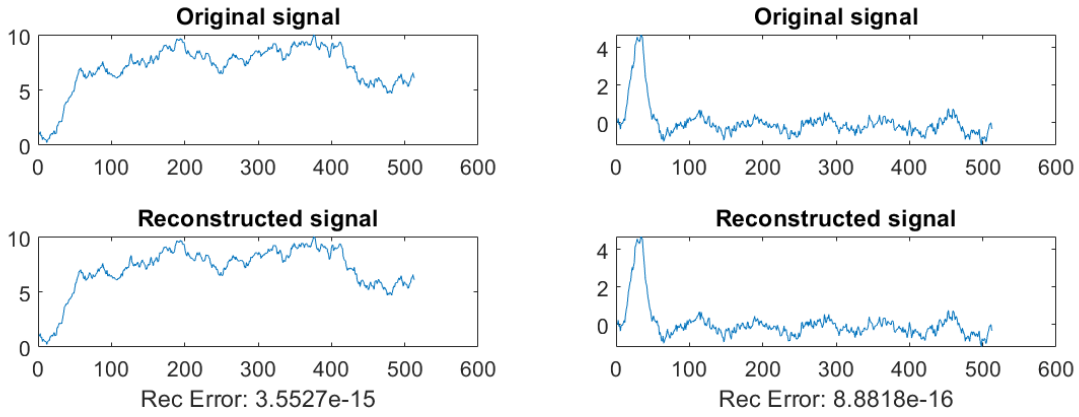


Figure 3.5. The high noise SVDP reconstructions for $x_1(t)$ (left) and $x_2(t)$ (right).

We leave analysis of these results to be done by an SME in the area of EWT.

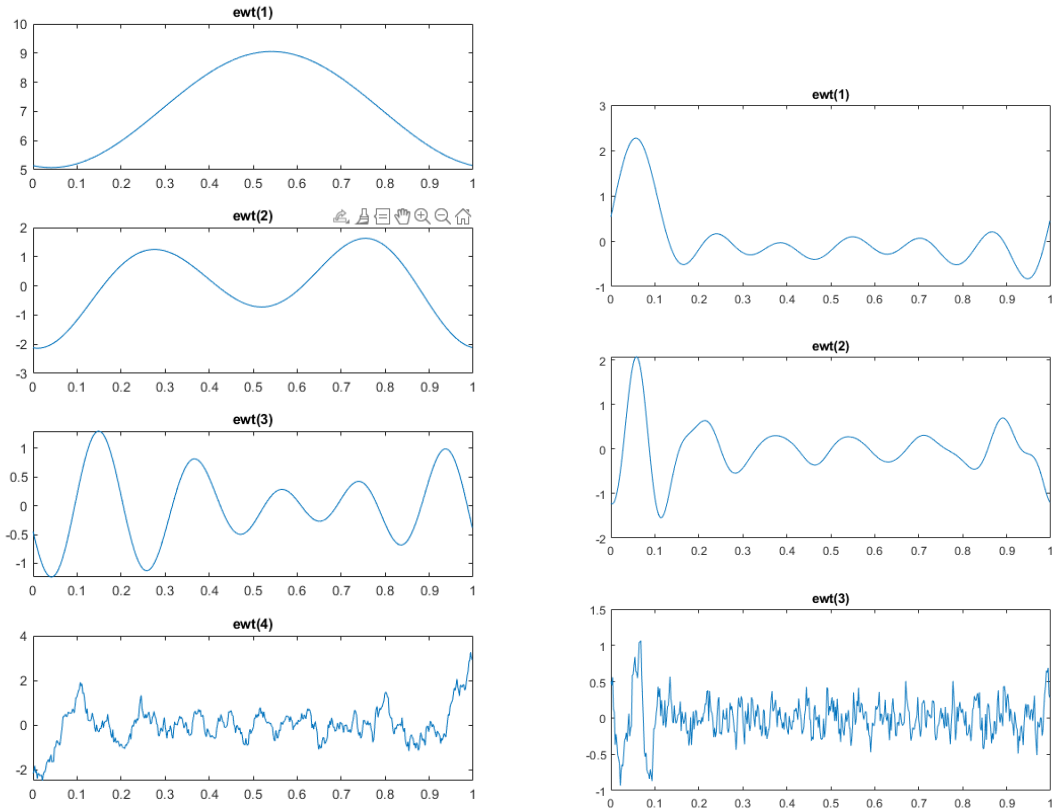


Figure 3.6. The high noise SVDP EWT components for $x_1(t)$ (left) and $x_2(t)$ (right).

Comparing Unaltered EWT vs EWT with ONFM (high noise)		
Item	Unaltered EWT	EWT w/ ONFM
Number of Boundaries for: x_1	20	3
Reconstruction Error for: x_1	7.1054e-15	3.5527e-15
Number of Boundaries for: x_2	18	2
Reconstruction Error for: x_2	3.5527e-15	8.8818e-16

Table 3.2. The results of running the ONFM algorithm and replacing the boundaries values in the `EWT1D` algorithm for the low noise SVDP.

CHAPTER 4
DISCUSSION AND FUTURE WORK

4.1 Discussion

It is yet to be determined if the ONFM technique developed in this thesis will be useful for EWT. Regardless, the sound theory and excellent implementation of Neural Spline Flows (NSF) from Section (2.2.3.2.1) in the `Pyro` library is a powerful tool for data analytics, and allowed us to produce boundaries for EWT from highly non-linear and non-trivial problem sets. It is also remarkable that by composing only linear and quadratic spline transformations via NSF, acceptable target PDFs were able to be generated for a histogram as messy as what was produced via adaptive rejection sampling for an FFT. As a final note on NSF and normalizing flows generally, this process could be applied to any problem in $\mathbb{R}^n$ and not just the 1D cases as were explored here, leaving it wide open for any number of applications.

4.2 Future Work

4.2.1 Neural Spline Flow Instability

There were times when using NSF lead to numerical instability or simply a poor model, opening three lines of potential future exploration.

First, I noticed a pattern that NSF seemed to struggle more when the domain of the inputs was not tightly centered and of higher values. For example, during the process of evaluating an FFT as a histogram for EWT, I had forgotten to shrink the domain to be within $[-\pi, \pi]$. As such, I effectively had the same data qualitatively, but on a domain of $[-512, 512]$. With this spread, no matter how small the learning rate was, I could not get even a mildly acceptable recreation of the target distribution. It would be worth determining if the polynomials used in NSF become unstable for higher values. I therefore hypothesize that a NSF NN has a harder time tuning parameters for a polynomial between larger input values as opposed to smaller ones. If this is true it is a critical gap worth addressing.

Second, the spikes generated by NSF, as can be seen in Figure (2.13), were a constant source of headaches that would require simply running the entire model again. By producing a new model with more epochs in the training loop a new target PDF without problematic spikes would sometimes be created. If not, I would repeat this

process several times until the undesirable features were removed. This process was obviously quite time consuming and tedious. In a practical application these could easily be discovered and removed by some clever techniques (e.g., a linear spline between points at the foothills of the large spikes), but this does not remove the possibility that there is some fundamental issue with the basic concept or theory behind NSF that should be examined.

Third, the `Pyro` package contains a host of other bijective transform options. Two of which mentioned here were the Discrete Cosine Transform (DCT) and the Householder Transform (HT). While exploring various model types, I did notice a slight qualitative difference in the outputs when these two transforms were used. However, without having explored the theory behind them, the reasons for these deltas were largely a mystery. A simple but impactful step would be to study these two transforms to evaluate how they could be useful for some future application.

4.2.2 Different Choice of Base Distribution

All of the work that has been done starts out with a normalized Gaussian distribution. However, while completing the toy example of mapping to a double humped Gaussian, the algorithm struggled the most at zero where the Gaussian distribution needed to be manipulated the most qualitatively to become the dip between the two Gaussian peaks in the target distribution as opposed to being the hump in the base. Therefore there is an entire rich area to explore of which distribution types might work better for different problem sets as the base distribution. I would guess that if some pool of data is largely bimodal, then utilizing a double humped Gaussian could reduce training time and be easier for the NN to manipulate into an acceptable form as opposed to having to find a way to heavily manipulate the peak of the Gaussian which I noticed to be a more difficult task. To further expand on this point, while doing the single to double hump Gaussian toy problem, I noticed the algorithm had an extremely difficult time if $\pm\mu$ was sufficiently large, effectively creating a nearly zero valued gap between the two humps. This forces a nearly vertical increase in intervals of $\mathcal{F}$ which can cause numerical instabilities resulting in `NaN` and `Inf` parameter values that would crash the code. By choosing a different base distribution, this could possibly be avoided.

BIBLIOGRAPHY

[1] J. G. (2023). Gilles, Jerome. Empirical Wavelet Transforms (https://www.mathworks.com/matlabcentral/fileexchange/42141-empirical-wavelet-transforms), MATLAB Central File Exchange. Retrieved June 8, 2023.

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

[3] D. Bourke. Daniel Bourke, Daniel Bourke, Youtube.com. *Learn PyTorch for deep learning in a day. Literally.* 23 Jul. 2022. https://github.com/mrdbourke/pytorch-deep-learning.

[4] G. Casella, C. P. Robert, and M. T. Wells, *Generalized accept-reject sampling schemes*, Lecture Notes-Monograph Series, (2004), pp. 342–347.

[5] L. Debnath, *Brief historical introduction to wavelet transforms*, International Journal of Mathematical Education in Science and Technology, 29 (1998), pp. 677–688.

[6] C. Durkan, *Neural Spline Flows*, 33rd Conference on Neural Information Processing Systems, (2019).

[7] J.-F. L. Gall, *Brownian Motion, Martingales, and Stochastic Calculus*, Springer Publishing Company, Incorporated, 2018.

[8] J. Gilles. Course Booklet for M668-Fourier Analysis. *Dr. Jérôme Gilles* Jan. 2013, 2023. San Diego State University Department of Mathematics and Statistics.

[9] J. Gilles. MATLAB Empirical Wavelet Transforms. *Dr. Jérôme Gilles*12 Aug. 2022.

[10] T. Hastie, *The Elements of Statistical Learning Data Mining, Inference, and Prediction*, Lecture Notes in Mathematics, Springer, Standford, California, 2 ed., 2008.

[11] I. Kobyzev, S. J. Prince, and M. A. Brubaker, *Normalizing flows: An introduction and review of current methods*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 43 (2021), pp. 3964–3979.

[12] Y. LeCun and C. Cortes, *MNIST handwritten digit database*, (2010).

[13] NVIDIA, P. Vingelmann, and F. H. Fitzek, *Cuda, release: 10.2.89*, 2020.

[14] N. Otsu, *A threshold selection method from gray-level histograms*, IEEE Transactions on Systems, Man, and Cybernetics, 9 (1979), pp. 62–66.

[15] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *Pytorch: An imperative style, high-performance deep learning library*, in Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 2019, pp. 8024–8035.

[16] D. Phan, N. Pradhan, and M. Jankowiak, *Composable effects for flexible and accelerated probabilistic programming in numpyro*, arXiv preprint arXiv:1912.11554, (2019).

[17] G. Sanderson. Grant Sanderson, 3Blue1Brown, Youtube.com. *Video Series: Neural networks.* Oct. 5, 2017. https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

[18] S. H. Strogatz, *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry and Engineering*, Westview Press, 2000.

[19] M. E. Valentinuzzi. Institute of Electrical and Electronics Engineers. *Highlights in the History of the Fourier Transform.* Jan. 25, 2016. https://www.embs.org/pulse/articles/highlights-in-the-history-of-the-fourier-transform/.

[20] E. W. Weisstein. Weisstein, Eric W. "Normal Distribution". *From MathWorld–A Wolfram Web Resource.* 24 Mar. 2023, https://mathworld.wolfram.com/NormalDistribution.html.

APPENDIX A
SOURCE CODE

SOURCE CODE

Here is a link to the GitHub repository that contains the files for this project. If the link does not work, feel free to contact the thesis author, Stefan Cline at: "stefan.e.cline@gmail.com".

URL to the code repository for this thesis:
https://github.com/StefanCline/Coding_Examples/tree/main/Normalizing%20Flows%20Thesis

Herein one can find:

1. The Normalizing Flows 1D examples.
   (a) Supporting `Python` files and `.pkl` data files.
   (b) By changing `sig_name` one can run: SVDP, ECG, DHG (double hump Gaussian), Harmonics, Stochastic Linear, and `sig2` (EWT Example).
   (c) The required files that will adjust the `Pyro` source files that prevent the code from running on a GPU.
2. The MNIST digit identification example.
3. The MATLAB files that were used to then compare results with/against EWT.

APPENDIX B
FLOW CHART OF EWT CODE

FLOW CHART OF EWT CODE

A link is provided to the flow chart of the Empirical Wavelet Transform code by Dr. Jérôme Gilles which can be found here

https://www.mathworks.com/matlabcentral/fileexchange/42141-empirical-wavelet-transforms.

Note that this flow chart does not exhaustively map out the EWT code, but provides a sufficient overview for the purposes of this thesis.

APPENDIX C
OM and EWT

OM and EWT

This algorithm outlines the process as described in the referenced paragraph. We note that the Gaussian Kernel in the 1D signal case is the typical

$$\hat{g}(\xi) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-\xi^2}{2\sigma^2}\right)$$

Algorithm 1 Histogram Development for OM for EWT

Require: $0 < \sigma \ll 1$, $f(t) \in \mathbb{C}$

Ensure: $\sigma \ll 1$, $P = \emptyset$

$\hat{g} \equiv$ Gaussian Kernel ▷ in Frequency Space

$f \mapsto |\hat{f}|$ ▷ shifting and keeping only the positive half

while $\sigma \leq \sigma_f$ **do** ▷ $\sigma_f \equiv$ final sigma value

$C(\xi) = (\hat{f} * \hat{g}_\sigma)(\xi)$

$P_\sigma = \text{Inv}[\text{maxima}(C)]$ ▷ Domain value, ξ that maps to each specific $C(\xi)$

$P = P \cup P_\sigma$ ▷ keep all duplicates

$\sigma = \sigma + \Delta\sigma$ ▷ $\Delta\sigma$ is a small step value

end while

$k^* = \Omega(P)$ ▷ $\Omega \equiv$ 2 Class OM

if $\xi > k^*$ **then**

$f(\xi) = 0$

end if

With the high frequency values of the function f removed in the second class, the signal is denoised and the OM process can be seen as a **high pass filter**.

APPENDIX D

Single to Double Hump Gaussian Training Loop Images

Single to Double Hump Gaussian Training Loop Images

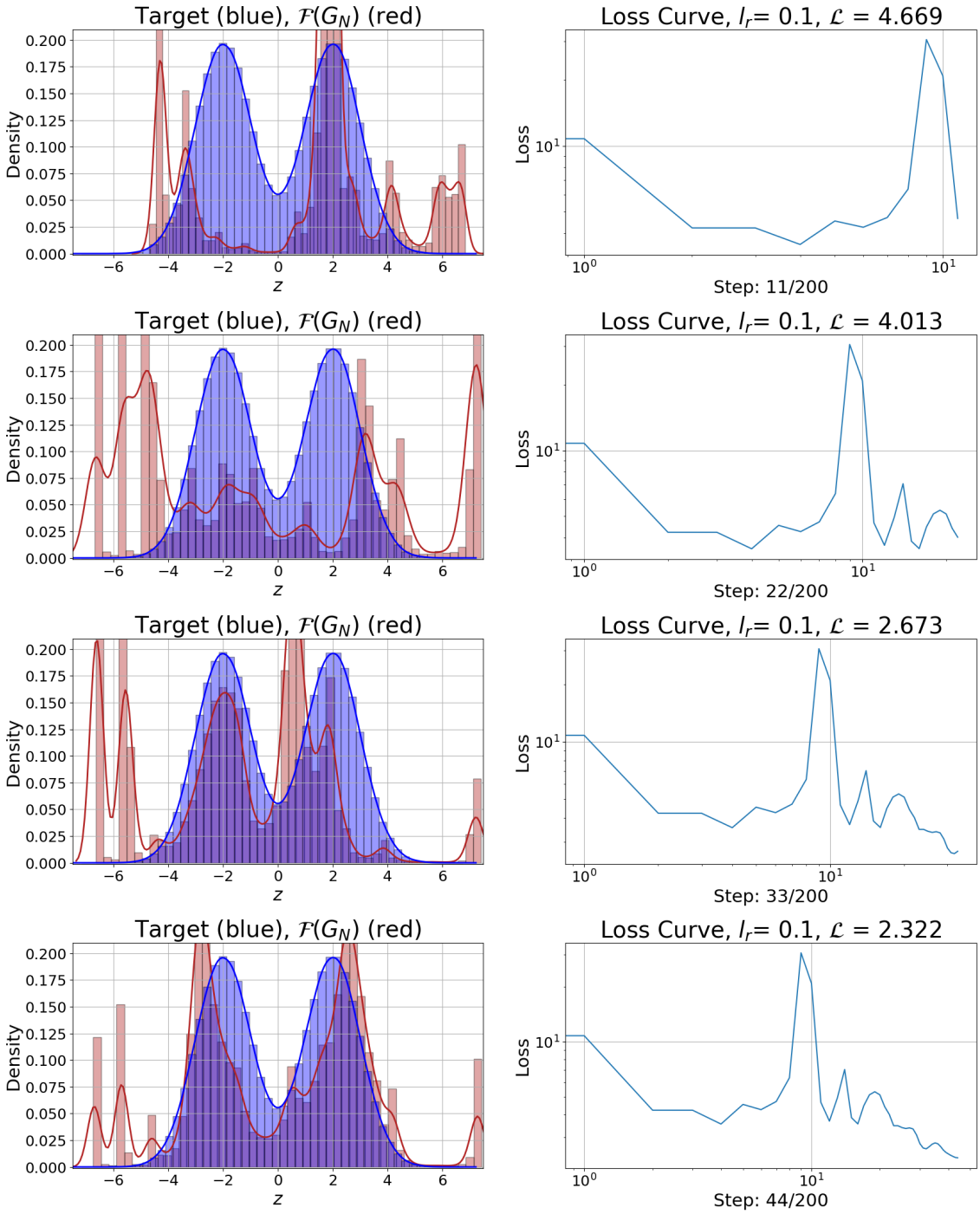
The curves and histograms shown in the plots are from the `seaborn.distplot` function call. Here the options selected were (note `seaborn` is imported as `sns`)

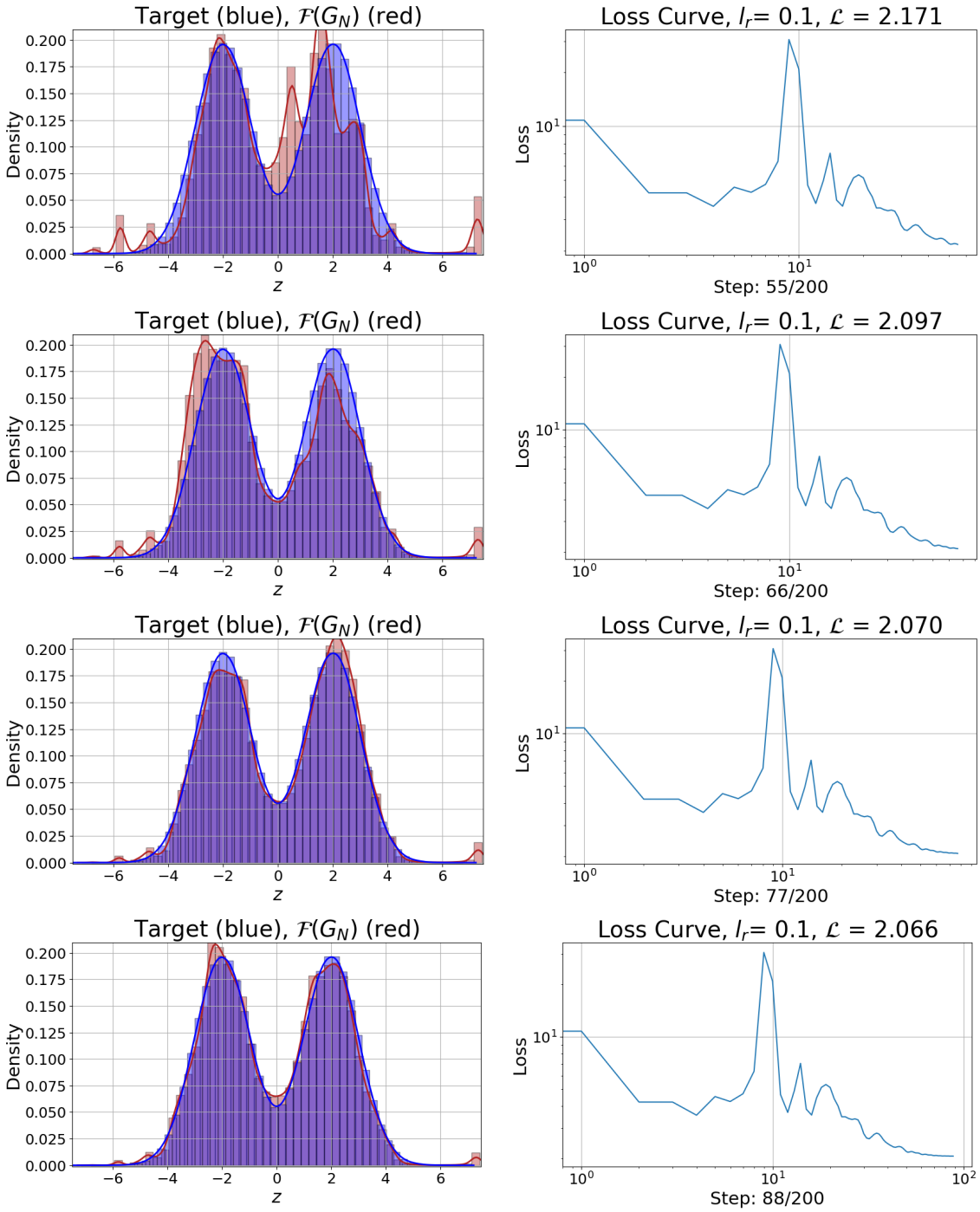
Approximation: `sns.distplot(forward_flow.squeeze(), hist=True, kde=True, bins=None, color='firebrick', hist_kws='edgecolor':'black', kde_kws='linewidth': 2, label='flow')`

Target: `sns.distplot(np.asarray(ps), hist=True, kde=True, bins=None, color='blue', hist_kws='edgecolor':'black', kde_kws='linewidth': 2, label='flow')`

It is critical to note that the curves shown in the above plots are not the approximated PDF but a KDE generated using a normal Gaussian kernel. This is an option I elected to leave on in the `seaborn` plots (`kde=True`) because though obviously not the approximated target PDF using normalizing flows, it is easier to get a rough idea of how well the NN is doing at that step by looking at the KDE than the histogram.

For approximations of the target PDF, i.e., the double humped Gaussian as a numerically approximated function, plots found in Section (2.2.3.5) display this.


Target (blue), $\mathcal{F}(G_N)$ (red)
Density
z
Loss Curve, l_r= 0.1, $\mathcal{L}$ = 4.669
Loss
Step: 11/200
Target (blue), $\mathcal{F}(G_N)$ (red)
Density
z
Loss Curve, l_r= 0.1, $\mathcal{L}$ = 4.013
Loss
Step: 22/200
Target (blue), $\mathcal{F}(G_N)$ (red)
Density
z
Loss Curve, l_r= 0.1, $\mathcal{L}$ = 2.673
Loss
Step: 33/200
Target (blue), $\mathcal{F}(G_N)$ (red)
Density
z
Loss Curve, l_r= 0.1, $\mathcal{L}$ = 2.322
Loss
Step: 44/200


Target (blue), $\mathcal{F}(G_N)$ (red)
Density
z
Loss Curve, l_r= 0.1, $\mathcal{L}$ = 2.171
Loss
Step: 55/200
Target (blue), $\mathcal{F}(G_N)$ (red)
Density
z
Loss Curve, l_r= 0.1, $\mathcal{L}$ = 2.097
Loss
Step: 66/200
Target (blue), $\mathcal{F}(G_N)$ (red)
Density
z
Loss Curve, l_r= 0.1, $\mathcal{L}$ = 2.070
Loss
Step: 77/200
Target (blue), $\mathcal{F}(G_N)$ (red)
Density
z
Loss Curve, l_r= 0.1, $\mathcal{L}$ = 2.066
Loss
Step: 88/200

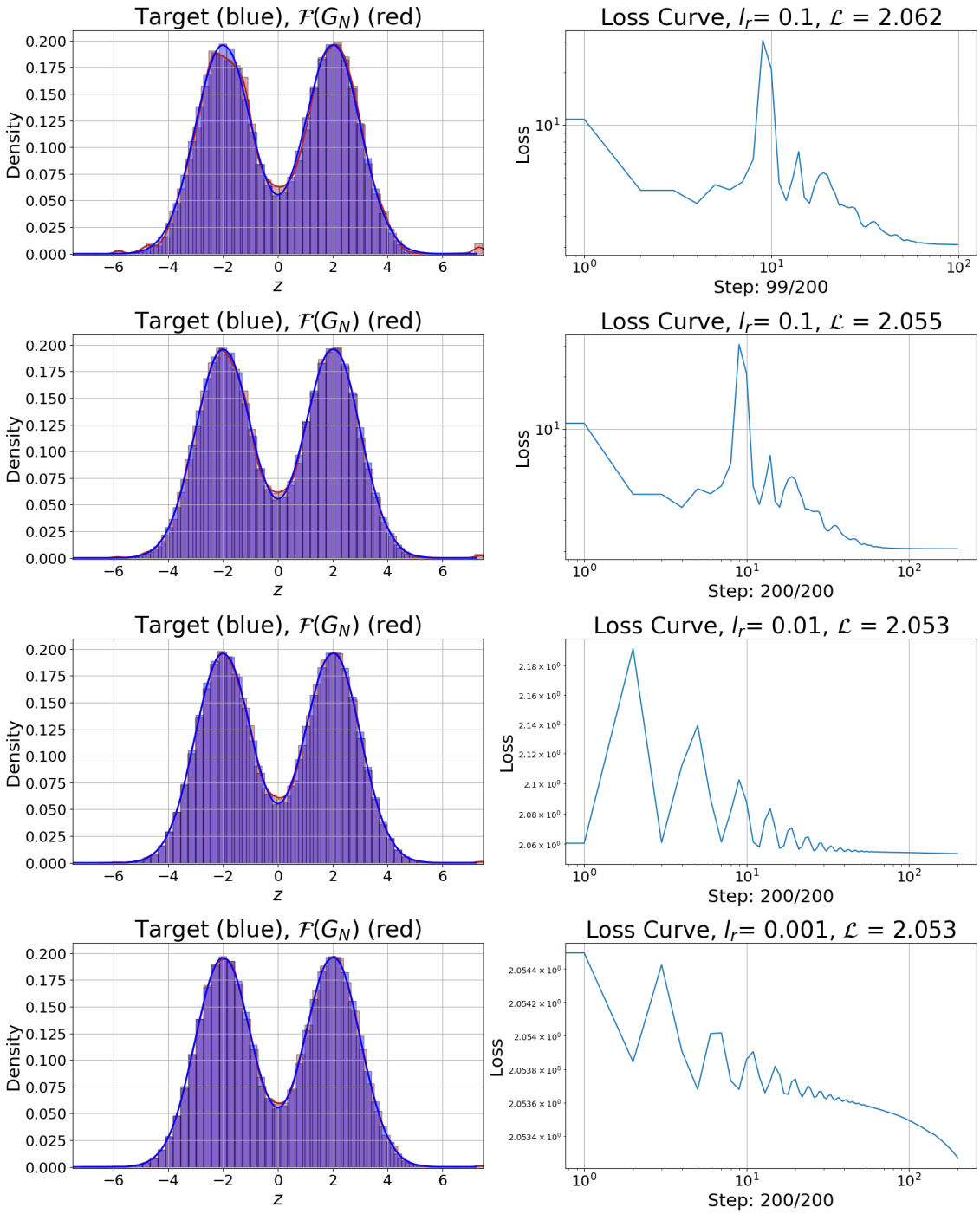


Figure D.1. Showing the evolution of the output of the NN as it trains. The images are show in order of first the learning rate (recall that $l_r = [0.1, 0.01, 0.001]$), and second the epoch (or step).

APPENDIX E

Area Preservation Through the Bijective Mapping

Area Preservation Through the Bijective Mapping

We wish to show that if we have some bijective function while only using Neural Spline Flows, that if we take the integral under one PDF at a point a, that that point when mapped forward through the bijective Neural Net, $f(a)$, will result in having the same area for the new probability function. Graphically we show an example below where we are have that $A = B$. Here we denote the Gaussian as $p_Z(z)$ and the

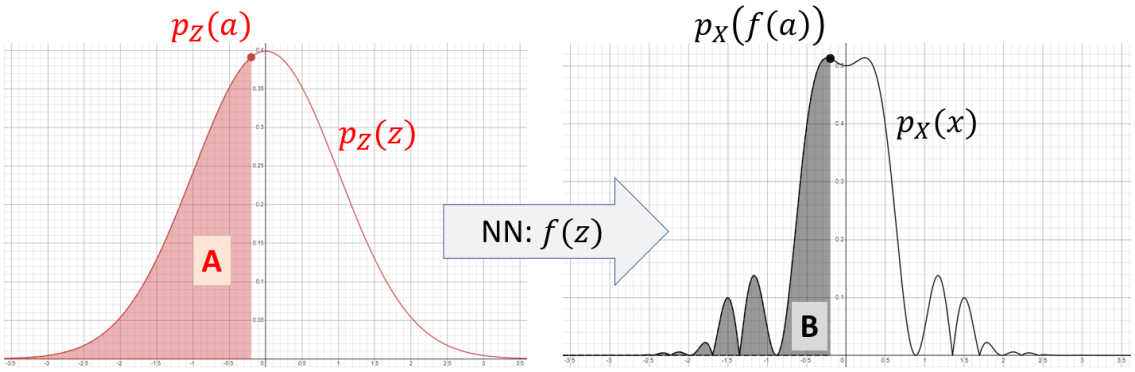


Figure E.1. An example of what we wish to show given a Gaussian and some arbitrary curve with a total area equal to one.

arbitrary PDF as $p_X(x)$ and we note that

$$p_X(x) = p_Z(z)f'(x) \tag{E.1}$$
$$= p_Z(f(x)) \cdot f'(x) \tag{E.2}$$

Starting with the assumption that we preserve probability mass between the two functions

$$\int_{\mathbb{R}} p_X(x)\ dx = \int_{\mathbb{R}} p_Z(z)\ dz$$

We will now go ahead and arbitrarily split up the integral into two pieces at some point a and $f(a)$ respectively.

$$\underbrace{\int_{-\infty}^{a} p_X(x)\ dx}_{(1)} + \int_{a}^{+\infty} p_X(x)\ dx = \underbrace{\int_{-\infty}^{f(a)} p_Z(z)\ dz}_{(2)} + \int_{f(a)}^{+\infty} p_Z(z)\ dz$$

Now, we wish to show that (1) and (2) are equal. Substituting using E.2 gives

$$\int_{-\infty}^{a} p_Z(f(x)) \cdot f'(x)\ dx = \int_{-\infty}^{f(a)} p_Z(z)\ dz$$

Because we know that the Gaussian is p_Z we can expand the last line to be[1]

$$\int_{-\infty}^{a} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-f^2(x)/2\sigma^2} \cdot f'(x)\ dx = \int_{-\infty}^{f(a)} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-z^2/2\sigma^2}\ dz$$

$$\int_{-\infty}^{a} e^{-f^2(x)/2\sigma^2} \cdot f'(x)\ dx = \int_{-\infty}^{f(a)} e^{-z^2/2\sigma^2}\ dz$$

$$\frac{\sqrt{2\pi}\sigma}{2} \operatorname{erf}\left(\frac{\sqrt{2}f(x)}{2\sigma}\right)\Bigg|_{-\infty}^{a} = \frac{\sqrt{2\pi}\sigma}{2} \operatorname{erf}\left(\frac{\sqrt{2}z}{2\sigma}\right)\Bigg|_{-\infty}^{f(a)}$$

$$\operatorname{erf}\left(\frac{\sqrt{2}f(x)}{2\sigma}\right)\Bigg|_{-\infty}^{a} = \operatorname{erf}\left(\frac{\sqrt{2}z}{2\sigma}\right)\Bigg|_{-\infty}^{f(a)}$$

$$\operatorname{erf}\left(\frac{\sqrt{2}f(a)}{2\sigma}\right) - \operatorname{erf}\left(\frac{\sqrt{2}f(-\infty)}{2\sigma}\right) = \operatorname{erf}\left(\frac{\sqrt{2}f(a)}{2\sigma}\right) - \operatorname{erf}\left(\frac{\sqrt{2}(-\infty))}{2\sigma}\right)$$

Knowing this should be in the limit as f tends to negative infinity and as some value tends to negative infinity, we then arrive at this statement

$$\operatorname{erf}\left(\frac{\sqrt{2}f(-\infty)}{2\sigma}\right) = \operatorname{erf}\left(\frac{\sqrt{2}(-\infty))}{2\sigma}\right) \tag{E.3}$$

Now, because we are using Neural Spline Flows, we know that outside of the $[-B, -B] \times [B, B]$ box, the derivative is set to a constant of one. As such, this implies that outside of the box, we simply have that $f(y) = y$. Hence, we know that

$$\lim_{b \to -\infty} f(b) = -\infty$$

Therefore, the two error functions on both the LHS and RHS of Eq.(E.3) both tend to -1 in the limit meaning the two expressions are equal.

Lastly, it is then trivial by extension that

$$\int_{a}^{+\infty} p_X(x)\ dx = \int_{f(a)}^{+\infty} p_Z(z)\ dz$$

also holds.

As such, we have shown that the main property preserved by $a \mapsto f(a)$ is that the area under the respective curves remains the same.

[1]The erf integral results were produced by MAPLE and thus the details are not shown here.

APPENDIX F

Boundaries used for the ECG Signal Reconstruction in the EWT Toolbox Code

Boundaries used for the ECG Signal Reconstruction in the EWT Toolbox Code

The following boundary values were inserted into the `EWT1D.m` file, thereby overriding the outputs of the boundaries that would have otherwise been produced during the test example for `signal='sig4'`.

[0.45340616; 0.07097941; 1.18459965; 0.83451871; 0.7893087; 1.460094; 1.03008774; 2.27397038; 2.53616895; 3.00593969; 1.50160673; 0.57332442; 0.17914847; 2.75229442; 1.55980544; 2.11541548; 2.3415545; 0.83995257; 2.69866077; 1.55311697; 2.78620032; 2.74840419; 2.4133053; 0.45359548; 1.06531227; 0.90271438; 1.75018342; 1.32748553; 1.08675937; 2.80111874; 0.84116928; 1.74437401; 0.54826974; 0.59846512; 0.93400286; 2.03225008; 1.18588059; 0.50230717; 1.6809052; 3.14113614; 1.43372607; 1.20818402; 1.82726773; 1.48517724; 0.11069132; 0.72297747]

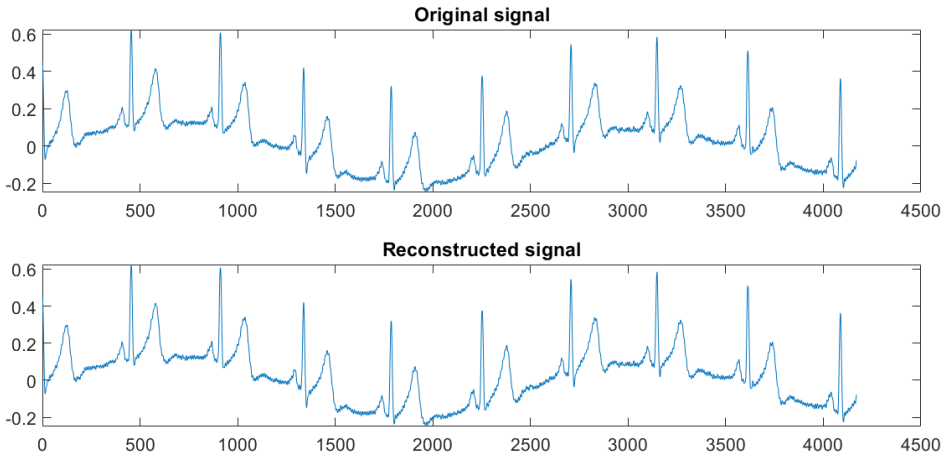


Figure F.1. The original signal and its reconstruction via EWT. Note that both reconstructions are identical at this scale, and thus we only present it one for brevity.

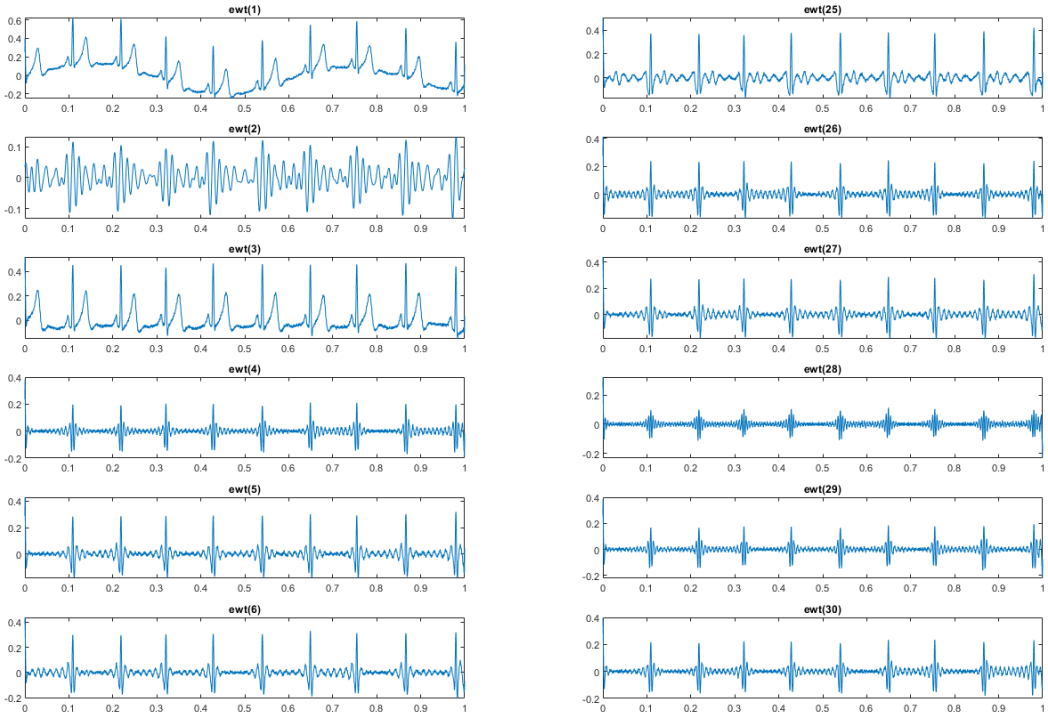


Figure F.2. The EWT components generated using the the boundaries from ONFM, which are listed at the beginning of Appendix F. As discussed in Section (3.1.2.1), many of these modes appear to be duplicates, and as such we only show 12 of the 47 EWT components here.

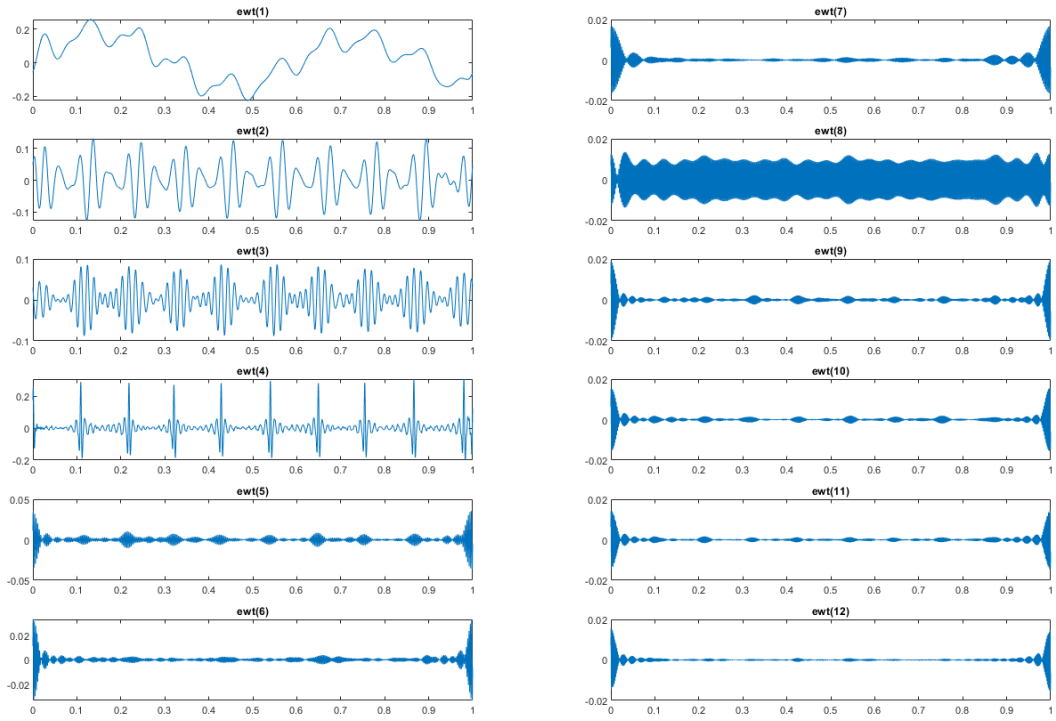


Figure F.3. The EWT components generated using EWT without any alterations. Here, there appear to be four main modes with the rest of the components that capture the noise of the signal. As such, we only again show 12 of the 47 components.

APPENDIX G
ONFM Algorithm

ONFM Algorithm

This algorithm outlines the process as described in Section (3.1.1.2). Below we shorthand the process of completing: a Normalizing Flow between a histogram of data and a standard normal Gaussian distribution to NF, and the process of performing a multi-Otsu thresholding algorithm to OM_n where n is the number of break points.

Algorithm 2 Otsu Normalizing Flow Method

Require: $S[t] \in \mathbb{R}$

Take FFT: $S[t] \to \hat{S}[\omega]$

Rescale $\omega \mapsto \bar{\omega}$ ▷ $\bar{\omega} \in [-\pi, \pi]$

Build $\hat{H}_R = \hat{S}[\bar{\omega}_R]$ ▷ adaptive rejection sampling on $[0, \pi]$

Copy $\hat{H}_R$ reflectively about $0 \Rightarrow \hat{H}_L$

$\hat{H} = \hat{H}_R \cup \hat{H}_L$

$NF(\hat{H}) = \hat{S}_A[\bar{\omega}] \approx \hat{S}[\bar{\omega}]$

Require: $0 < p_s < p_f < 100,\ \mathbf{B} = \emptyset$

for Interp_Type in ['linear', 'quad', 'cubic'] **do**

Interpolate $\hat{S}[\bar{\omega}]$ using Interp_Type

Determine $A = \int_0^\pi \hat{S}_A \, d\bar{\omega}$

for k in [2,n] **do** ▷ n is arbitrarily chosen and $n \geq 2$

for perc in $[p_s, p_f]$ **do** ▷ Percent of area being kept

Find x s.t. $\int_0^x \hat{S}_A \, d\bar{\omega}/A = \text{perc}$

$\vec{b} = MO_k(\hat{S}_A \in [0, x])$

$\mathbf{B} = \mathbf{B} \cup \vec{b}$ ▷ As a new row in $\mathbf{B}$

end for

end for

end for

Import $\mathbf{B}$ into MATLAB

Set recon $\gg 1$

for r in R **do** ▷ R = number of rows in $\mathbf{B}$

Replace `boundaries` in `EWT1D.m` with $\mathbf{B}_r$

if Reconstruction Error < recon **then**

recon = Reconstruction Error ▷ Save row in $\mathbf{B}$

end if

end for

Best reconstruction error determines best break points, and images of the reconstruction as well as the reconstruction value are produced.

APPENDIX H

SVDP EWT Components for the Low and High Signal Reconstructions from Running EWT Unaltered

SVDP EWT Components for the Low and High Signal Reconstructions from Running EWT Unaltered

The below figures are the EWT components from the unaltered EWT shown: low noise x_1, low noise x_2, high noise x_1, and high noise x_2 from Section (3.1.2.2).

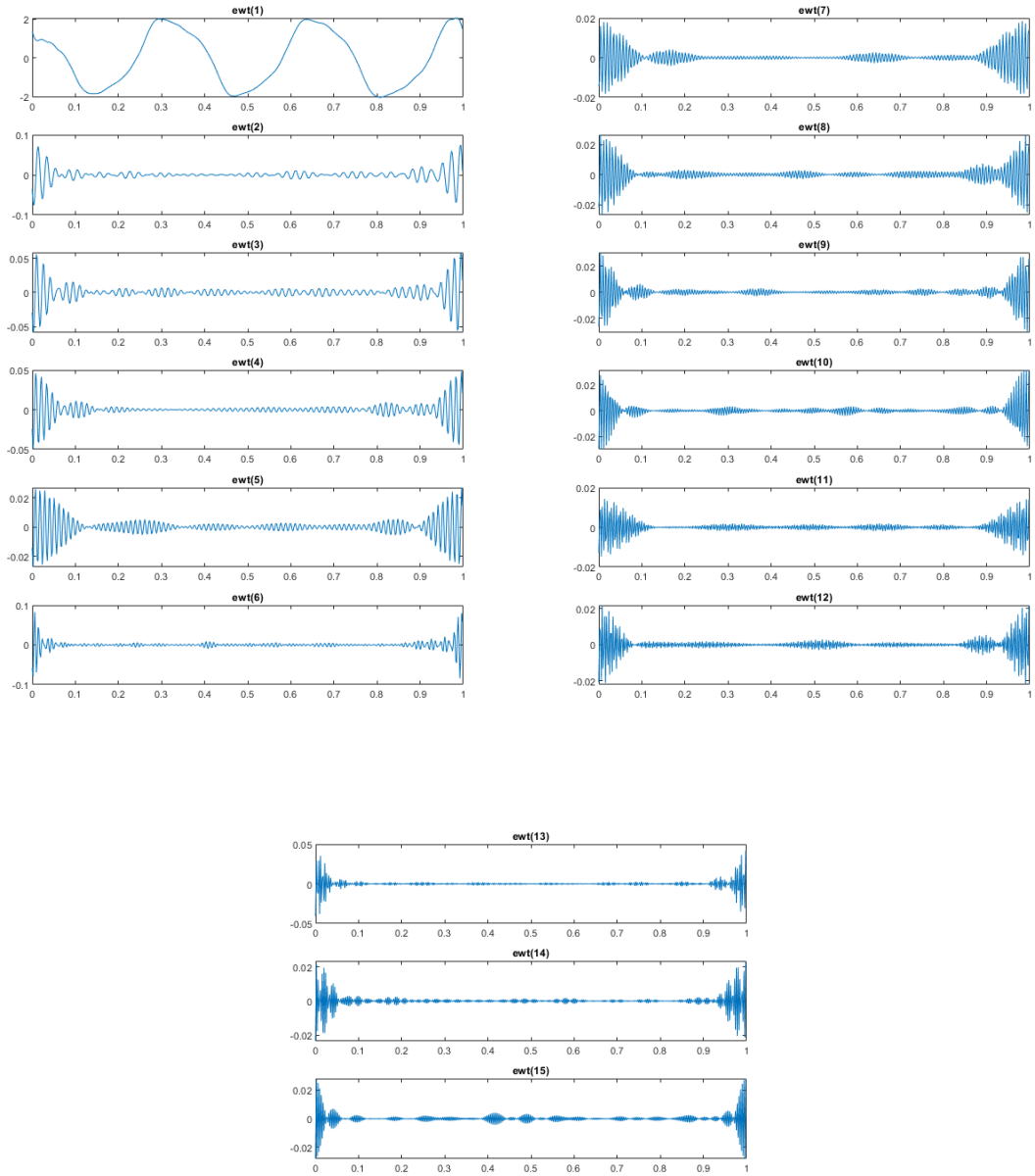


Figure H.1. Low noise SVDP x_1 EWT components.

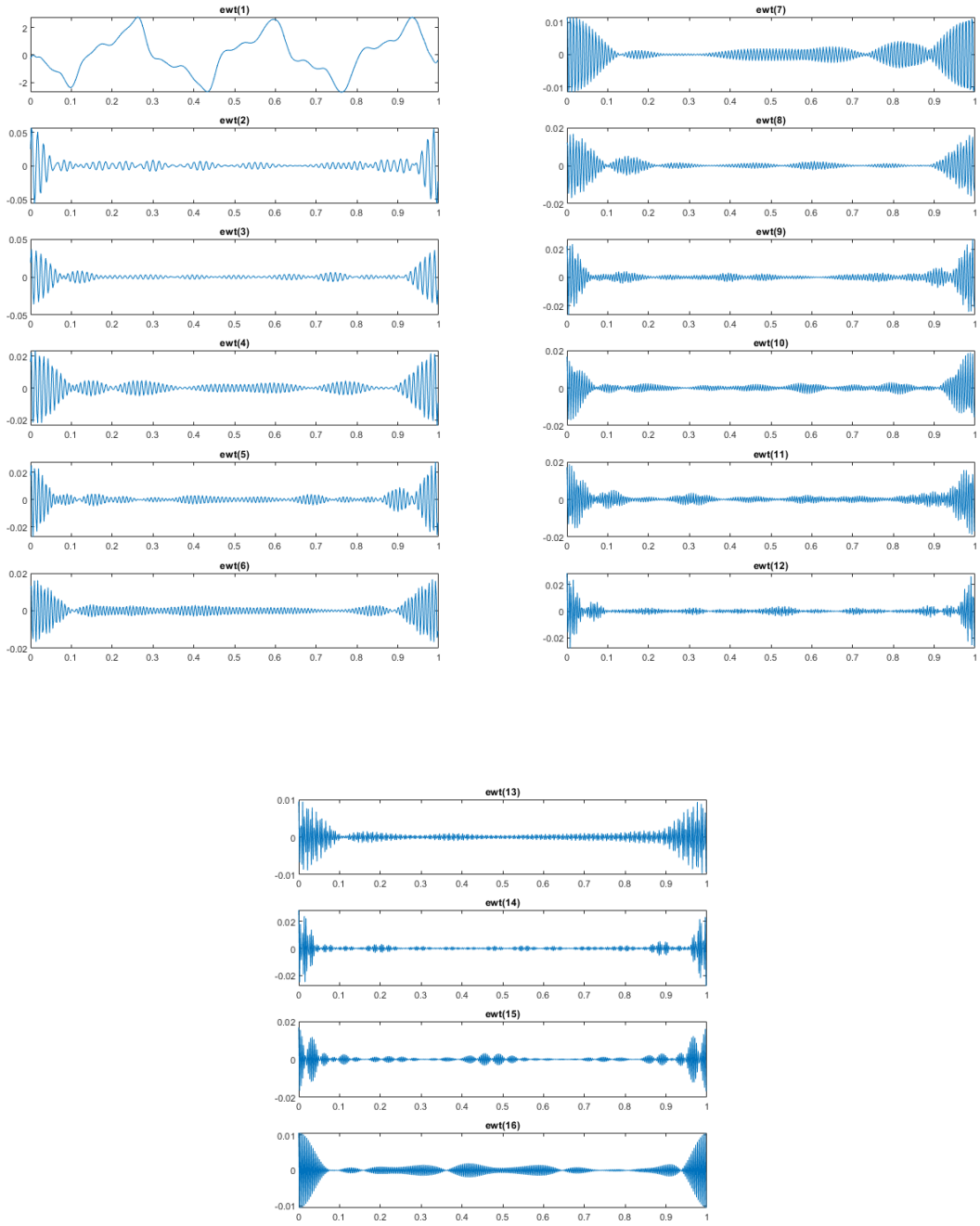


Figure H.2. Low noise SVDP x_2 EWT components.

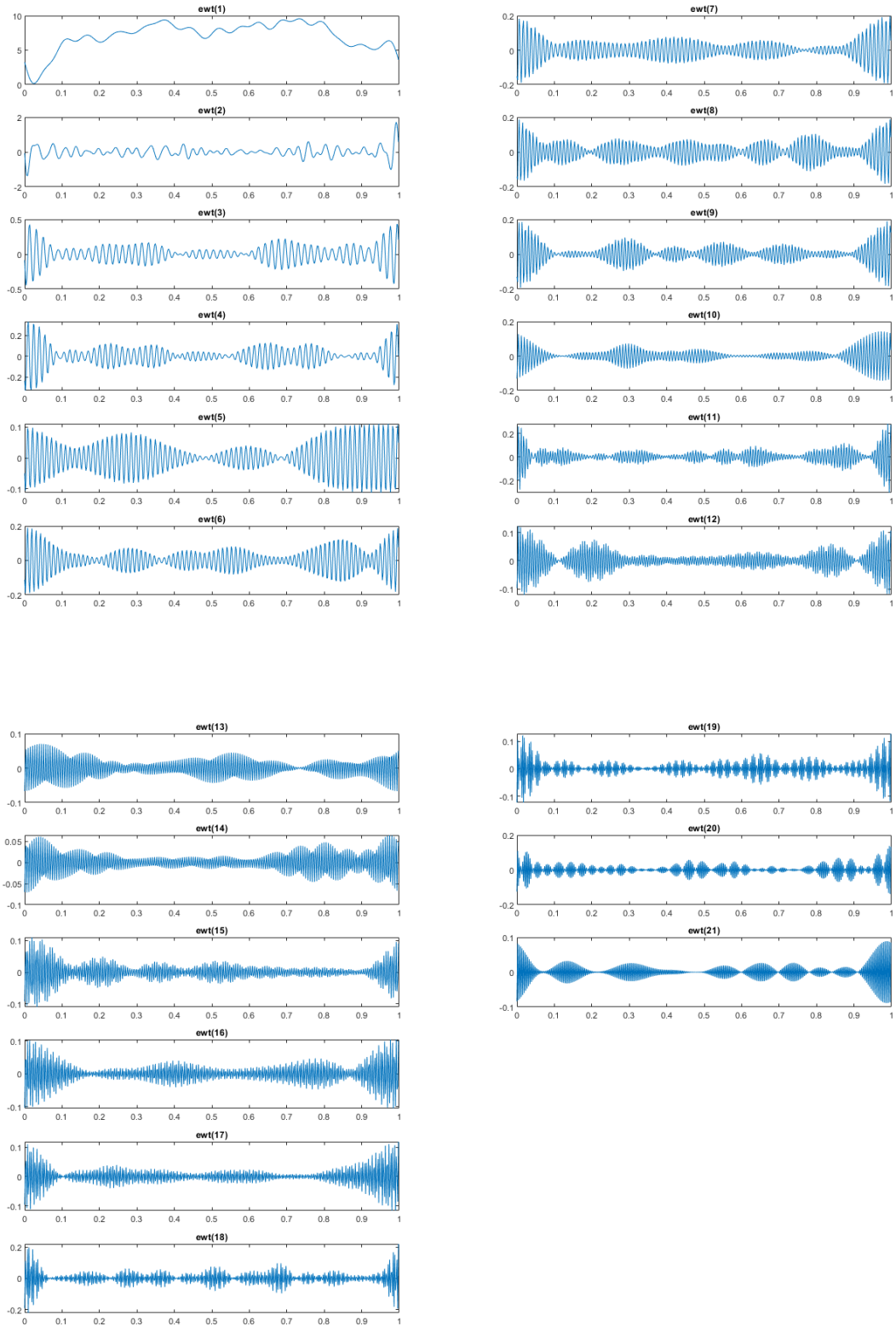


Figure H.3. High noise SVDP x_1 EWT components.

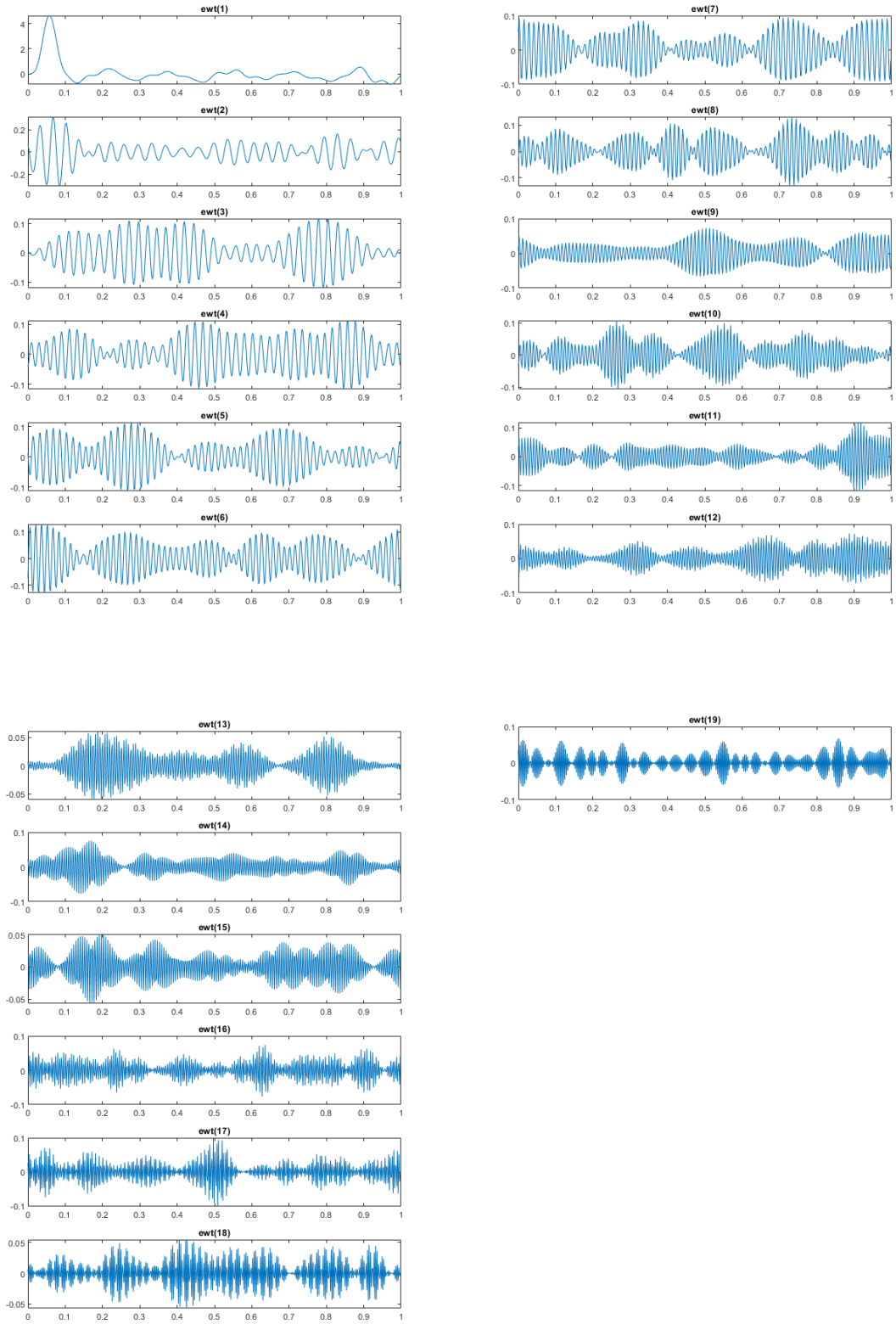


Figure H.4. High noise SVDP x_2 EWT components.