TUM School of Computation, Information and Technology
Technische Universität München

Machine Learning for Irregularly-Sampled Time Series

Marin Biloš

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:

Prof. Dr. Matthias Grabmair

Prüfer der Dissertation:

1. Prof. Dr. Stephan Günnemann
2. Prof. Dr. David Kristjanson Duvenaud,
   University of Toronto

Die Dissertation wurde am 13.04.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 23.11.2023 angenommen.

Abstract

In medical, industrial, financial, and many other domains, it is common practice to monitor how measurements change over time. In such settings, the times of measurements are often not uniformly spaced. In this thesis, we consider such irregularly-sampled time series, with the goal of defining a generative model for both the measurements and their arrival times. We further extend our approach to address critical tasks such as time series classification, forecasting future values, and imputing missing data. First, we combine neural density estimation and temporal point processes to define a scalable generative model of the arrival times. Our model offers closed-form maximum likelihood training and straightforward sampling. Next, using the time of the measurement as input, we introduce a model that predicts the class of the measurement along with its associated uncertainty. For data that is continuously observed, we propose a diffusion-based generative model that can also be conditioned on other available information to successfully forecast and impute missing values. Finally, we design a model that learns the solutions to differential equations by imposing certain design constraints on the neural network. Our model naturally handles the continuous time while offering fast evaluation. Therefore, it can be used to model dynamical systems and, when coupled with recurrent neural networks, defines an encoder for irregularly-sampled time series.

Zusammenfassung

In medizinischen, industriellen, finanziellen und vielen anderen Bereichen ist es üblich, zu überwachen, wie sich Messungen im Laufe der Zeit ändern. In solchen Situationen sind die Messzeiten oft nicht gleichmäßig verteilt. In dieser Arbeit betrachten wir solche unregelmäßig erfassten Zeitreihen mit dem Ziel, ein generatives Modell sowohl für die Messungen als auch für deren Ankunftszeiten zu definieren. Wir erweitern unseren Ansatz weiter, um kritische Aufgaben wie die Klassifizierung von Zeitreihen, die Vorhersage zukünftiger Werte und die Imputation fehlender Daten anzugehen. Zunächst kombinieren wir neuronale Dichteschätzung und temporale Punktprozesse, um ein skalierbares generatives Modell der Ankunftszeiten zu definieren. Unser Modell bietet Maximum-Likelihood-Training und unkompliziertes Sampling. Als nächstes führen wir unter Verwendung der Messzeit als Eingabe ein Modell ein, das die Klasse der Messung zusammen mit ihrer zugehörigen Unsicherheit vorhersagt. Für kontinuierlich beobachtete Daten schlagen wir ein diffusionsbasiertes generatives Modell vor, das auch auf andere verfügbare Informationen konditioniert werden kann, um erfolgreich Vorhersagen zu treffen und fehlende Werte zu imputieren. Schließlich entwerfen wir ein Modell, das die Lösungen von Differentialgleichungen lernt, indem es bestimmte Designbeschränkungen auf das neuronale Netzwerk auferlegt. Unser Modell behandelt die kontinuierliche Zeit auf natürliche Weise und ist schnell auszuwerten. Es kann daher zur Modellierung dynamischer Systeme verwendet werden und definiert in Verbindung mit rekurrenten neuronalen Netzwerken einen Encoder für unregelmäßig erfasste Zeitreihen.

Acknowledgments

I would like to thank my supervisor Prof. Stephan Günnemann for providing me an outstanding support through this academic journey. I am grateful for your mentorship and for creating a relaxed atmosphere mixed with a good dose of humor.

Thanks to all the wonderful people at DAML group, I will especially cherish our lunch conversations. I would like to thank my DAML collaborators Oleksandr Shchur, Bertrand Charpentier and Johanna Sommer for being a big part of this thesis.

Special thanks to my collaborators Syama Sundar Rangapuram, Tim Januschowski, Kashif Rasul, Anderson Schneider and Yuriy Nevmyvaka, especially for your support during my internships.

Thanks to Nicholas Gao who helped translate the abstract to German.

Htio bih se zahvaliti obitelji na pruženoj podršci i ljubavi.

Ivana, hvala ti za sve.

Contents

1 Introduction

It is increasingly common to regard data as a commodity, not unlike a typical ore. Sensors capture raw signals by observing their surroundings and the scrappers save the footprints across the online world. This raw material is then refined and sometimes enriched with manual labeling, for it to be fed into algorithmic machines that compress and abstract away, with the aim of producing models. Once the learning is complete, the original data is all discarded but its task-dependent, distilled representation is embedded in the model's parameters. At that point, the model is deployed to perform its designated task with the promise of efficiency and accuracy, beyond what humans could achieve.

The history of machine learning predates modern computing machines; Gauss had been fitting curves to the motions of celestial bodies already in the early 19th century [77]. In the simplest case, the collection of points can be summarized by two parameters, the slope and the offset of the curve—a straight line passing through the points. However, with more complicated data, we have to utilize more advanced approaches. The first step was to make the models non-linear [174] which, in the above example, gives us access to all kinds of different curves, matching the real trajectory more closely. For even more complicated data like images and time series, it is hard to find a simple mapping between input and output. The progress was made by stacking representations into a hierarchy, where each layer abstracts more that the previous, essentially learning the features of data which are then easier to map to the correct output [113]. This is referred to as deep learning, a revolution in machine learning that was led by improvements in both the hardware and the software [194, 207] and which allowed achieving great results in image classification [142], as well as speech recognition [101], language translation [257], time series forecasting [232], and others.

In this thesis, we will consider data that is collected irregularly in continuous time,[1] with missing values, and having varying sequence lengths. This kind of data can be found in many domains, including medical measurements, dynamical systems, stock prices, location data, social interactions and various systems' logs, to name a few.

1.1 Contributions and outline

We aim to answer a research question that naturally imposes itself when we consider applying neural networks to data that has some special structure:

> How to **represent** and **output** irregularly-sampled data using neural networks?

[1] Meaning with varying time lengths between the observations; not having a constant sampling rate.

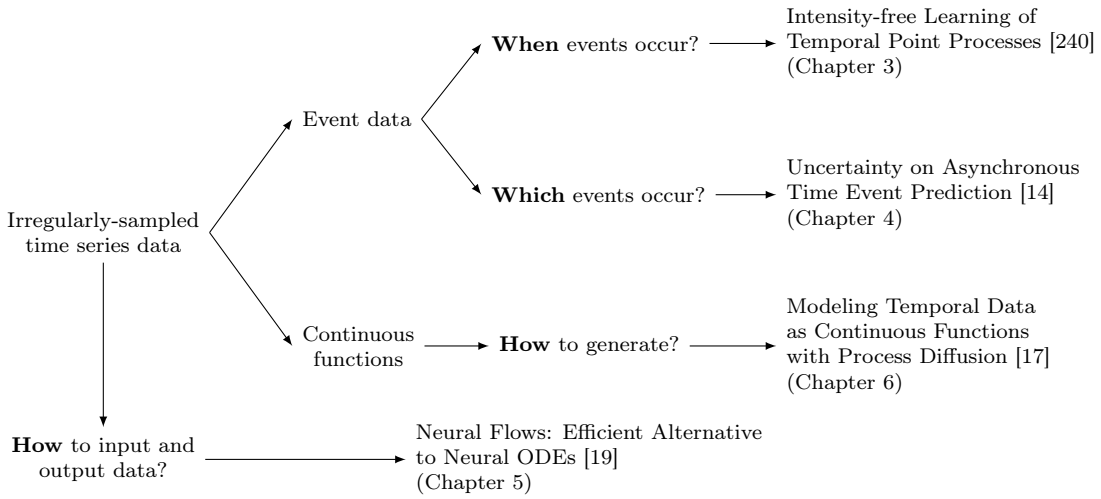


Figure 1.1: Frequently asked questions when encountering irregular time series. Depending on the task we can use different methods that we present throughout this thesis. The above diagram shows the structure of the chapters together with the corresponding papers they are based on. For a full list of publications, see Section 1.2.

When using neural networks, we prefer representing data compactly, with a fixed-sized vector that can be passed on to the downstream task. Compared to the conventional data types such as images or even regular time series, the main challenge here is tackling the nonuniform sampling rate. In this thesis, we will explore models that can handle continuous time and irregularity to produce useful representations of such data.

Figure 1.1 shows some common questions one might ask when dealing with irregularly-sampled time series. The first thing to notice are the two main types of time series that we consider: events and continuous values. For example, some observations are only made at a specific time point and do not persist, like earthquakes or messages between two people. On the other hand, measurements like temperature or velocity are assumed to be continuous in time and they exist even when we do not take a measurement. Based on this categorization we will approach modeling such data in different ways.

Since event data is strictly tied to their observation times, a natural thing to ask is whether we can model the arrival time of new events. A common solution to this problem specifies density over time, that is, a function that tells us how many new events we expect to see on some time interval. The framework we will use to model this is called a temporal point process (TPP). In **Chapter 3** we will see how to combine TPPs with deep learning, to produce more expressive models, along with some other desirable properties.

Time points are rarely observed on their own, we usually have additional information tied to them; for example, who is messaging whom or the location of the earthquake. Therefore, given a history of observations we would like to be able to predict what happens at some future time point. Modern machine learning, in particular deep learning, excels at predictive supervised tasks, however, we often observe overconfidence in prediction, especially when encountered with an out-of-distribution data. We discuss this in more

detail in **Chapter 4**. The way we solve this problem is by designing a model that can assign an amount of certainty to a prediction, including the case when it is uncertain if any event will happen.

In all of the previous cases we had some kind of model that inputs an irregularly-sampled time series and outputs the same type again or represents it as a fixed-sized vector. One way to output irregular data is to assume it follows some function, learn this function, and evaluate it at the given time points. For that, we can use an ordinary differential equation as it describes how given initial points evolve over time. Combining the ODEs with neural networks allows learning the dynamics from data directly. In **Chapter 5** we present an alternative model called neural flows that also learns the dynamic, same as neural ODEs, but without ever specifying the differential equation. Instead, we learn the solutions to differential equations directly with a neural network which allows us to have computationally efficient model while preserving the desired properties of ODEs. To represent the data with a fixed-sized vector we can assume it too evolves over time and gets updated whenever we encounter a new data point. Thus, the model can naturally handle irregularity. We can use such a model to, for example, represent the history on which we condition TPPs.

As mentioned before, some quantities might not be best described as events since the time of measurement is dependent on our sensors and the value can be queried at any time point, like when measuring temperature. Therefore, we assume there is some underlying continuous function from which we sample measurements. In **Chapter 6** we aim to find a model that generates functions as samples, and we make a connection to stochastic processes. One way to generate new data is to start from pure noise and gradually denoise the sample until we reach a sample from the data distribution. The denoising is performed with a neural network which is trained on many clean-noisy data pairs. In our case, we extend this framework to temporal data and present different applications that our model supports, such as imputation and interpolation.

1.2 Own publications

The following is the full list of publications with author's involvement during the PhD:

- Marin Biloš, Bertrand Charpentier, and Stephan Günnemann. "Uncertainty on Asynchronous Time Event Prediction". In: *Neural Information Processing Systems (NeurIPS)*. 2019.
- Nick Harmening, Marin Biloš, and Stephan Günnemann. "Deep Representation Learning and Clustering of Traffic Scenarios". In: *Workshop on AI for Autonomous Driving, International Conference on Machine Learning (ICML)*. 2020.
- Oleksandr Shchur, Marin Biloš, and Stephan Günnemann. "Intensity-Free Learning of Temporal Point Processes". In: *International Conference on Learning Representations (ICLR)*. 2020.

- Oleksandr Shchur, Nicholas Gao, Marin Biloš, and Stephan Günnemann. "Fast and Flexible Temporal Point Processes with Triangular Maps". In: *Neural Information Processing Systems (NeurIPS)*. 2020.
- Marin Biloš and Stephan Günnemann. "Scalable Normalizing Flows for Permutation Invariant Densities". In: *International Conference on Machine Learning (ICML)*. 2021.
- Marin Biloš, Andreea Muşat, and Stephan Günnemann. "Point Processes on Graphs with Normalizing Flows". In: *Workshop on Deep Learning on Graphs, AAAI Conference on Artificial Intelligence*. 2021.
- Marin Biloš, Johanna Sommer, Syama Sundar Rangapuram, Tim Januschowski, and Stephan Günnemann. "Neural Flows: Efficient Alternative to Neural ODEs". In: *Neural Information Processing Systems (NeurIPS)*. 2021.
- Marin Biloš, Emanuel Ramneantu, and Stephan Günnemann. "Irregularly-Sampled Time Series Modeling with Spline Networks". In: *Workshop on the Continuous Time Methods for Machine Learning, International Conference on Machine Learning (ICML)*. 2022.
- Marin Biloš, Andrei Smirdin, and Stephan Günnemann. "Modeling Solutions to Ordinary and Partial Differential Equations with Continuous Initial Value Networks". In: *Workshop on the Continuous Time Methods for Machine Learning, International Conference on Machine Learning (ICML)*. 2022.
- Marin Biloš, Kashif Rasul, Anderson Schneider, Yuriy Nevmyvaka, and Stephan Günnemann. "Modeling Temporal Data as Continuous Functions with Process Diffusion". In: *International Conference on Machine Learning (ICML)*. 2023.

2 Background

In this chapter we provide a background on neural networks (Section 2.1) as well as a short overview of the methods for generative modeling (Section 2.2). The structure in some sections is inspired by [84, 48, 240, 202, 22]; for a more extensive treatment of the topics that follow, the reader is referred to these materials.

2.1 Neural networks

An artificial[1] neural network is, in the broadest sense, any differentiable function f with a set of learnable parameters $\boldsymbol{\theta}$. We denote an input value with $\boldsymbol{x} \in \mathbb{R}^{d_x}$ and the network output with $\boldsymbol{y} \in \mathbb{R}^{d_y}$, $\boldsymbol{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$. Perhaps the simplest example of a neural network is the single-layer feedforward network $\boldsymbol{y} = \sigma(\boldsymbol{W}\boldsymbol{x}) + \boldsymbol{b}$, where $\boldsymbol{W} \in \mathbb{R}^{d_y \times d_x}$, $\boldsymbol{b} \in \mathbb{R}^{d_y}$, and $\sigma : \mathbb{R} \to \mathbb{R}$ is a non-linear function which acts elementwise[2] on an input vector preserving the dimension—also known as the activation function. The learnable parameters in this case constitute of a matrix $\boldsymbol{W}$ and vector $\boldsymbol{b}$. An intermediate value $\boldsymbol{h} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$ is called a hidden or latent value.

Although, in theory, a single hidden layer with an activation function is enough to capture a large family of functions [46], the success of neural networks comes from stacking multiple layers on top of each other. If we alternate between affine transformations and activation functions, we get a multilayer feedforward network:[3]

$$\boldsymbol{y} = \sigma_l(\boldsymbol{W}_l\sigma_{l-1}(\dots\sigma_1(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1)\dots) + \boldsymbol{b}_l), \tag{2.1}$$

with all the weights and biases having compatible dimensions. The learnable parameters are $\boldsymbol{\theta} = \{\boldsymbol{W}_1, \dots, \boldsymbol{W}_l, \boldsymbol{b}_1, \dots, \boldsymbol{b}_l\}$. This is one of the basic building blocks for our models; we cover others, such as recurrent neural networks, later in the chapter.

Given data and an architecture such as Equation 2.1, we want to find the best $\boldsymbol{\theta}$ for the task at hand, for example, regression. To actually find the parameters, we have to specify what we mean by saying that "something is the best fit." In case of linear regression this can be the average distance between the true values y and predicted points $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ which we call a *loss function*. The problem of finding the best parameters is now the problem of *minimizing* the loss, also known as *training*.

For classification one would use a network that outputs a probability that a certain class is the correct one and optimize cross-entropy loss. The network is now constrained

[1] We omit *artificial* in the rest of the text and refer to the models simply as *neural*.

[2] This is not always the case, consider $\text{softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$ which is used to output probabilities in a classification task. It will also be used in Section 2.1.3 as a key element of transformer model.

[3] Also sometimes referred to as a multilayer perceptron and abbreviated as MLP.

to output values between zero and one which can be achieved by using softmax as the final activation. In this example, we can already see how a given task imposes certain constraints that can be directly implemented inside a model. Many of the methods presented in the thesis will follow this pattern.

Finally, to actually learn the optimal values of $\boldsymbol{\theta}$ given a loss function we usually perform gradient descent. First, the gradient of the loss is computed with respect to all the learnable parameters, using automatic differentiation. Second, we update the parameters using the gradient information to minimize the loss in an iterative manner.[4] Loss landscape can be visualized as a surface whose height depends on $\boldsymbol{\theta}$. The gradient then points in the direction of the slope, and the gradient descent will lead us to a minimum, analogous to rolling a ball downhill.

Therefore, it is crucial that the neural network and the loss are built from differentiable operations. This is a requirement that is already satisfied by many functions which allowed designing neural networks with ease, which in turn allows building models that specialize on different, structured data types. For example, recurrent neural networks excel on temporal data (Section 2.1.2). In the following, we will see more examples.

2.1.1 Residual networks and dynamical systems

When we scale the number of layers, the feedforward network (Equation 2.1) suffers from vanishing gradients, that is, the magnitude of the gradient is lower in the first layers and approaches zero, meaning the parameters cannot be updated. This is especially true when using a sigmoid or tanh activation function. A part of the solution is to use a better suited activation function $\sigma(z) = \max(0, z)$, called ReLU. Another improvement is to introduce residual connections that allow the gradients to flow deeper through the network. Let $\boldsymbol{x}_t$ be the output after t layers, then the residual layer is defined as:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + g_t(\boldsymbol{x}_t), \tag{2.2}$$

where $g_t : \mathbb{R}^d \to \mathbb{R}^d$ is a feedforward network. Even if g_t does not pass any gradient information, the skip connection "$+\boldsymbol{x}_t$" will. Stacking multiple residual layers gives us the residual network (ResNet) that is used in image classification and other tasks [100, 273], where we need many layers to extract hierarchical features.

Coincidentally, Equation 2.2 bears a resemblance to a discretization of a dynamical system, which was noticed by several previous works [160, 277]. Instead of viewing $\boldsymbol{x}_t$ as a result of t consecutive layers, we can think of it as a result of some dynamics after t time steps, given a starting point $\boldsymbol{x}_0$ (network input). Further, g_t is seen as being *copied* across all layers, but it now depends on t and not only $\boldsymbol{x}_t$. If g_t is independent of time we call the system *autonomous*. If t remains an integer, we get a discrete system and Equation 2.2 is called a *map*. Letting t be continuous and updating Equation 2.2 in small increments $t \mapsto t + \epsilon$ defines a continuous system that is called a *flow*. We will discuss flows in more detail in Chapter 5.

[4] In practice, it is customary to use existing programming frameworks that implement backpropagation, optimizers and other useful functions. In this thesis we mostly use PyTorch [207], in Chapter 4 we use TensorFlow [173].

Neural ordinary differential equations. The continuous time case is the one we are mostly interested in. This corresponds to defining an ordinary differential equation (ODE). Stating it more explicitly, let us define the time dependent variable $\boldsymbol{x}(t) \in \mathbb{R}^d$ and the neural network $f : \mathbb{R}^{d+1} \to \mathbb{R}^d$. Then the neural ODE can be written as [35]:

$$\frac{\mathrm{d}\boldsymbol{x}(t)}{\mathrm{d}t} = f(\boldsymbol{x}(t), t). \tag{2.3}$$

Abusing the notation to rearrange the terms we get $\mathrm{d}\boldsymbol{x}(t) = \boldsymbol{x}(t+\epsilon) - \boldsymbol{x}(t) = \mathrm{d}t \cdot f(\boldsymbol{x}(t), t)$ which highlights the similarity to Equation 2.2. Note that, although we interpret t as time, it can represent any change in some scalar quantity, for instance, the distance traveled along some line. Given the starting point $\boldsymbol{x}(t_0)$ at t_0 and the final time point t_1, we can compute $\boldsymbol{x}(t_1)$ with:

$$\boldsymbol{x}(t_1) = \boldsymbol{x}(t_0) + \int_{t_0}^{t_1} f(\boldsymbol{x}(t), t)\,\mathrm{d}t = \mathrm{ODESolve}(f, \boldsymbol{x}(t_0), t_0, t_1), \tag{2.4}$$

where the final right-hand side operation hints that the solution is often obtained by running a numerical solver, as the closed-form solution is usually nonexistent. The simplest numerical solver (Euler method) works similarly to Equation 2.2 as it performs the first-order updates. It is, however, imprecise so more advanced solvers are preferred as they can augment the number of steps and offer more numerical stability [60].

The uniqueness of the obtained solution $\boldsymbol{x}(t_1)$ is covered by the Picard-Lindelöf theorem [43] that states it is sufficient for f to be continuous in t and Lipschitz continuous in $\boldsymbol{x}$. Lipschitz continuity tells us that the function will not *blow up* at some points, by bounding the amount of change in the output given the change in the input. More rigorously, a function f is Lipschitz with constant α if for any x and y, $|f(x) - f(y)| < \alpha|x - y|$.

A function that has a Lipschitz constant $\alpha < 1$ is called a contraction since it is not expanding the space. A useful property of such maps is that they have a unique fixed point, that is, applying f to any input $\boldsymbol{x}$ repeatedly converges to the same output $\boldsymbol{x}^\star = (f \circ \cdots \circ f)(\boldsymbol{x})$, by the Banach fixed-point theorem [8]. If we make g_t in Equation 2.2 a contraction, the whole residual layer becomes invertible which will prove to be useful in Chapter 5 for constructing invertible neural networks.

Given input data $\boldsymbol{x}(t_0)$, t_0 and t_1 and a target value $\boldsymbol{x}(t_1)$ we want to learn the dynamics f. Since f is a neural network and the solver simply invokes f multiple times, adding up the derivatives $\mathrm{d}f$ from t_0 to t_1, the ODESolve call is fully differentiable and we can propagate the gradients through it. However, the solver might require hundreds of evaluations of f, meaning that the full computation graph looks like a very deep ResNet, leading to potential memory issues. A solution to this problem is presented in an adjoint sensitivity method which allows us to get the gradients by solving another ODE backwards in time, without storing all the intermediate values [35, 212]. This works with any ODE solver and has a much lower memory cost since the intermediate solutions are not stored.

In Chapter 5 we will discuss the drawbacks of specifying a dynamic with Equation 2.3 and argue in favor of modeling the solution of the dynamic directly. This will require implementing certain constraints in our network which we will solve by designing modified ResNets that satisfy the desired properties.

2.1.2 Recurrent neural networks

Given a sequence of inputs $(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)})$, a common task is to predict the outputs $(\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(N)})$. What we often want to know is the probability $p(\boldsymbol{y}^{(t)}|\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)})$ of an output $\boldsymbol{y}^{(t)}$ at a step t, given previous inputs. The restriction to only use the first t inputs comes from the natural temporal ordering of the data—we cannot look into the future. The model we consider is a neural network that takes the current input $\boldsymbol{x}^{(t)}$ and all the preceding inputs, and maps them to a vector $\boldsymbol{h}^{(t)}$, called a hidden state, from which we can predict the output. We use the same neural network to obtain hidden states at every step; the final result is the latent representation of a sequence. This network is called a recurrent neural network (RNN) and the update equations are defined as follows:

$$\boldsymbol{z}^{(t)} = \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{b} \tag{2.5}$$

$$\boldsymbol{h}^{(t)} = \tanh\left(\boldsymbol{z}^{(t)}\right) \tag{2.6}$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{V}\boldsymbol{h}^{(t)} \tag{2.7}$$

$$\hat{\boldsymbol{y}}^{(t)} = \operatorname{softmax}\left(\boldsymbol{o}^{(t)}\right) \tag{2.8}$$

where the learnable parameters $\boldsymbol{W}$, $\boldsymbol{U}$ and $\boldsymbol{V}$ are shared at every time step. Here, we specify a classification model with a softmax activation at the end. Since the operations are differentiable, we can train this model with backpropagation. The loss in this case is summed over all of the time steps.

To perform backpropagation we can think of an RNN as a classical neural network with N layers and with inputs and outputs at every layer. This is sometimes referred to as unrolling the RNN. We will again encounter a known issue of trying to propagate the gradient for many layers, or equivalently in our case, far into the past. The equations that update the parameters $\boldsymbol{W}$ and $\boldsymbol{U}$ depend on the gradient of loss with respect to hidden state $\partial\mathcal{L}/\partial\boldsymbol{h}^{(t)}$. If we write out this term for a simple one-dimensional case we get:

$$\frac{\partial\mathcal{L}}{\partial h^{(t)}} = \sum_{s=t}^{N} \frac{\partial\mathcal{L}^{(s)}}{\partial h^{(s)}} \frac{\partial h^{(s)}}{\partial h^{(t)}} = \sum_{s=t}^{N} \frac{\partial\mathcal{L}^{(s)}}{\partial h^{(s)}} \prod_{t\leq k\leq s} \frac{\partial h^{(k)}}{\partial h^{(k-1)}} = \sum_{s=t}^{N} \frac{\partial\mathcal{L}^{(s)}}{\partial h^{(s)}} \prod_{t\leq k\leq s} W\left(1 - \left(h^{(k)}\right)^2\right) \tag{2.9}$$

If W under the product is smaller than 1, the whole term will approach 0 (*vanish*) and if W is larger than 1 it will approach ∞ (*explode*), as $t \to \infty$. That means we cannot keep the gradient information over many steps, which in turn means the network cannot learn long-term dependencies. Pascanu et al. [206] prove that it is sufficient that $\rho < 1$, where ρ is the spectral radius of the recurrent weight matrix $\boldsymbol{W}$, for long term components to vanish and necessary that $\rho > 1$ for them to explode.

The problem of learning long term dependencies can be solved by introducing new architectures, similar to the residual networks (Section 2.1.1). Unlike ResNets, which simply pass the information in an additive way, new RNNs employ a *gating mechanism* that decides when to keep and when to discard the information. This allows the model to keep the hidden state intact for longer periods of time.

Gated recurrent unit (GRU) is one RNN improvement that updates a hidden state given previous state $\boldsymbol{h}^{(t-1)}$ and new input $\boldsymbol{x}^{(t)}$ using gating [40]:

$$\boldsymbol{z}^{(t)} = \sigma(\boldsymbol{W}_z[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}]) \tag{2.10}$$

$$\boldsymbol{r}^{(t)} = \sigma(\boldsymbol{W}_r[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}]) \tag{2.11}$$

$$\tilde{\boldsymbol{h}}^{(t)} = \tanh(\boldsymbol{W}[\boldsymbol{r}^{(t)} \odot \boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}]) \tag{2.12}$$

$$\boldsymbol{h}^{(t)} = (1 - \boldsymbol{z}^{(t)}) \odot \boldsymbol{h}^{(t-1)} + \boldsymbol{z}^{(t)} \odot \tilde{\boldsymbol{h}}^{(t)} \tag{2.13}$$

Equation 2.12 is the same as the update equation in a simple RNN if we think of a vector $(\boldsymbol{r}^{(t)} \odot \boldsymbol{h}^{(t-1)})$ as the previous state. Instead of getting the new state we get a *candidate* $\tilde{\boldsymbol{h}}^{(t)}$. The motivation for the following steps is that not every input should be fully taken into account when updating the hidden state. For that, we calculate the gate $\boldsymbol{z}^{(t)}$ that determines how much information from the candidate state are we going to take, and at the same time, how much information from previous state will be written over. When $\boldsymbol{z}^{(t)} = 0$ the whole previous state is kept and the network has completely discarded the new input (Equation 2.13). On the other hand, when $\boldsymbol{z}^{(t)} = 1$ it forgets everything about the past and only keeps the new state based on the new input. Note that $\boldsymbol{z}^{(t)}$ is a vector so the network can keep and discard per individual dimension of $\boldsymbol{h}^{(t)}$.

A different architecture is the long short-term memory (LSTM) [103] that historically preceded GRU. Instead of having one hidden state, there are two: a cell state vector $\boldsymbol{c}^{(t)}$ and a hidden state $\boldsymbol{h}^{(t)}$ that is also used as the output. Given a previous cell, a hidden state, and a new input, we can update the cell and hidden state as follows:

$$\boldsymbol{f}^{(t)} = \sigma(\boldsymbol{W}_f[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}]) \tag{2.14}$$

$$\boldsymbol{i}^{(t)} = \sigma(\boldsymbol{W}_i[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}]) \tag{2.15}$$

$$\boldsymbol{o}^{(t)} = \sigma(\boldsymbol{W}_o[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}]) \tag{2.16}$$

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \tanh(\boldsymbol{W}[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}]) \tag{2.17}$$

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \odot \tanh(\boldsymbol{c}^{(t)}) \tag{2.18}$$

where $\boldsymbol{f}^{(t)}$, $\boldsymbol{i}^{(t)}$ and $\boldsymbol{o}^{(t)}$ are forget, input and output gates, respectively. Second term on the right-hand side in Equation 2.17 is the familiar update of the hidden state from an RNN. It is multiplied with the input gate and combined with the gated previous state cell similar to Equation 2.13. Finally, we get the output by multiplying the cell state with the output gate. Notice that, since there are multiple gates, we can keep the previous state and the new input, or discard both of them. There are other variants that have different connections. For example, *peephole* LSTM replaces $\boldsymbol{h}^{(t)}$ with $\boldsymbol{c}^{(t)}$ in all equations [78].

We will use GRU and LSTM extensively through the thesis since they are one of the most common ways to encode a time series with a neural network. However, since we are dealing with irregularly-sampled time series, we will have to adapt the models to account for this. For example, in Chapter 5 we present an extension of GRU architecture that evolves the hidden state in continuous time between observations, instead of just using Equation 2.13.

2.1.3 Attention and transformers

As we have seen in Section 2.1.2, RNNs struggle learning to keep the information for many time steps. Such a behavior is undesirable in certain tasks like machine translation where the dependence between the words persists over longer horizons. The first encoder-decoder architectures [257] used one RNN to process the text in the source language and another RNN to output the translation to the target language, word-by-word. However, by the time the decoder would start translating, the first word from the source would be a full sentence away. This turned out to be an issue since the first word from the source is usually the most relevant when starting the translation. Therefore, Bahdanau et al. [6] introduced a way to look-up all the words in the source sentence and pick the most relevant one when generating the currently translated word, called attention.

Attention in neural networks is used to denote a *soft selection* from a set of elements. If we continue with the previous example, the set consists of individual words and we would like to select one of them. The simplest way to implement this is to have a layer that outputs the index of the word we want to select, however, this is not differentiable and therefore is not learnable. An alternative approach is to assign a unique weight value to all the words, and then multiply the vector representations of words by their (normalized) weights and sum everything up. If the weight vector is, for example, $(0.01, 0.99, \ldots, 0)$, we would be essentially selecting the second element, although the final vector would also include some information from other elements as well (note the 0.01 weight).

More formally, if we have a set of elements $(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n)$, $\boldsymbol{v}_i \in \mathbb{R}^d$ and a *query* vector $\boldsymbol{q} \in \mathbb{R}^d$, the output of the attention layer will be:

$$\boldsymbol{y} = \sum_i a_i \boldsymbol{v}_i, \quad \boldsymbol{a} = \text{softmax}(\boldsymbol{v}_1^T \boldsymbol{q}, \ldots, \boldsymbol{v}_n^T \boldsymbol{q}). \tag{2.19}$$

The inner product $\boldsymbol{v}_i^T \boldsymbol{q}$ measures the similarity of each input to the query, whereas the softmax can be seen as a way to normalize the resulting similarity scores. If some value is very close to $\boldsymbol{q}$, it will be preserved in the output. Vaswani et al. [273] abstract this further and introduce three matrices that pack together the vectors: $\boldsymbol{Q} \in \mathbb{R}^{n \times d_k}, \boldsymbol{K} \in \mathbb{R}^{n \times d_k}, \boldsymbol{V} \in \mathbb{R}^{n \times d_v}$, corresponding to queries, keys and values, respectively. The attention is now defined as:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right)\boldsymbol{V}, \tag{2.20}$$

which differs from Equation 2.19 by introducing multiple query vectors and separating the keys from the values. The keys play a role in measuring the similarity and the values correspond to the actual quantity we care about. Coming back to the machine translation example, $\boldsymbol{K}$ and $\boldsymbol{V}$ would both correspond to the source sentence representation and $\boldsymbol{Q}$ would represent the current state of the translated sentence. The separation of keys and values allows us to have one representation for the word that will be translated and one representation that is used for selection of the word with respect to the query.

Multi-head and self attention. Attention layer (Equation 2.20) can be applied to any *set-like* data: words, time series, point clouds, patches of pixels, etc. In practice, it is useful to apply attention in parallel which is referred to as multi-head attention. In the simplest case, we can copy the matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ h-times, transform them with the linear layers, run attention on each of the copies and concatenate back the results. Another useful transformation is when we apply the attention to the set itself, that is, given a set of points $\boldsymbol{X}$, we obtain the query, key and value matrices directly from $\boldsymbol{X}$ with a neural network and collect the result of an attention layer, which will preserve the cardinality. This allows for rich interactions between all the set elements.

Positional encoding. Attention, as proposed in Equation 2.20, does not distinguish between the different orderings of the elements and is, therefore, a permutation equivariant transformation. Although this can be useful in some cases [15], we would usually like to have a notion of ordering, especially when dealing with time series data. Vaswani et al. [273] propose using a positional encoding that takes the element position i and outputs the vector $\boldsymbol{p} = (p_1, \ldots, p_d)$, where d is the dimension of the input data:

$$p_j = \sin\left(\frac{j}{10000^{2j/d}} + j\pi\right). \tag{2.21}$$

Transformer is an architecture that combines all of the previously presented ideas to get an expressive encoder-decoder model. For example, in machine translation the encoder input will be the sentence tokens with the positional encoding added to them. Multiple stacked layers of multi-head (self) attention, layer normalization [4], and feedforward layers, with residual connections between them produce the source language sentence embeddings. The decoder has the same structure, however, the multi-head attention takes the encoder embeddings as keys and values and the currently translated sentence as query. The decoder has to be autoregressive, producing one token at a time, but during training the ground-truth translation is simply shifted to the right by one token and the attention implements masking such that it is impossible to *attend* to the future tokens. That is, the masking is causal and prevents leaking any information from the future when predicting the current token. Thus, the training can be parallelized.

We will use attention and transformers as a building block for a spatio-temporal model in Section 5.2.3. We build an attention-like model to capture distance between points in Section 6.3.2. Finally, we discuss how attention can be reparameterized to allow fast evaluation of its Jacobian for density modeling (see Section 3.4.1 for motivation, and [15] for a more detailed discussion).

2.2 Generative modeling

Most of the questions we ask when dealing with irregular time series (see Figure 1.1) can be answered by finding the appropriate distribution, for example, we rephrase *when* events occur to *which distribution* captures the arrival time of events. For a more concrete example, in regression we can specify the probability of the predicted value for

a given input $p(y|\boldsymbol{x})$. The value y is a scalar and the probability p is usually a normal distribution. However, in real-world, we will encounter other, more complicated cases. In machine translation, the conditional distribution $p(\boldsymbol{y}|\boldsymbol{x})$ is over the vector $\boldsymbol{y}$. Language embedding models, on the other hand, capture the distribution $p(\boldsymbol{x})$, where $\boldsymbol{x}$ is a set of tokens. Models that specify the distribution of the observables are called generative models, in contrast to modeling probability of target given observation which is known as discriminative modeling. In the rest of this section we will take a look at the ways to define a generative model, where the common goal between all of them is to enable generating new samples from the learned distribution.

In our case, we are dealing with temporal data $\{(\boldsymbol{x}_i, t_i)\}_i^n$ and would like to know *when* something happens, and *what* happens at a given time point. That is, we are interested in $p(\boldsymbol{t})$ where the set of time points $\boldsymbol{t} = (t_1, \ldots, t_n)$ is strictly increasing and the number of points is random, following some underlying process. The "what happens" question can be written as $p(\boldsymbol{x}|t)$—for a given time we model the distribution of the observations. Further, we might be interested in forecasting the future values given a history of n observations $\mathcal{H}_n = \{(\boldsymbol{x}_i, t_i)\}_i^n$: $p(\boldsymbol{x}_{n+1}, t_{n+1}|\mathcal{H}_n)$. When dealing with partially observed data we can ask what is the distribution of the missing values given the observed, and so on.

2.2.1 Temporal point processes

In this section we answer the question of *when the event happens* by defining the process that generates time points on an interval $[0, T]$. The number of points we observe on this interval together with their positions are random. This can be naturally described using the framework of temporal point processes (TPP). We first introduce the notion of a counting process $N(t)$ that returns the number of events on the interval $[0, t]$. Let $N(t)$ be a random variable and consider the following distribution:

$$\Pr\{N(t) = n\} = \frac{(\lambda t)^n}{n!} e^{-\lambda t}. \tag{2.22}$$

This is known as a Poisson distribution so we say Equation 2.22 specifies a Poisson process. The expected number of points on $[0, t]$ will be λt and the process is stationary (*homogeneous*) since it is Poisson with the same rate $\lambda \Delta t$ on any of the subsets of the interval with length Δt. The term λ is also known as the *intensity* of a process and is defined as a probability $\Pr\{\text{observing exactly one event on } [t, t + \epsilon)\}$. To generate new points from this process we first draw the number of elements n from a Poisson distribution, and then independently and uniformly sample n points from interval $[0, T]$.

Another way to sample new points is by drawing new arrival times one by one, sequentially in an increasing order. This is also a more natural way to perform sampling as the data was generated sequentially in time. For that, we consider the distribution $\Pr\{N(t) = 0\} = e^{-\lambda t}$, that is, the interval until the first point arrives which is also known as the survival function. Since the original process is stationary and points are independent, this distribution applies everywhere because it does not matter where we place the origin $t = 0$. Therefore, we can conclude the inter-arrival times $\tau_i = t_i - t_{i-1}$, with $t_0 = 0$, are exponentially distributed in the case of homogeneous Poisson process.

Having a uniform distribution in time is usually not very realistic. If we consider counting the cars on the highway, we expect different values at different times of the day, while observing the peak numbers during the rush hour. The solution is to make the intensity λ a function that changes with time: $\lambda(t)$. This allows us to define regions with more expected points and regions with less points. The sampling procedure is still the same, the total intensity $\Lambda = \int_0^T \lambda(t)\,\mathrm{d}t$ now acts as the parameter of the Poisson distribution and the locations of points follow the probability density function proportional to $\lambda(t)$, normalized to integrate to 1 on the interval which is easily obtained by dividing with Λ. We can write down the likelihood of the so-called *inhomogeneous* process:

$$p(t_1, \dots, t_n) = \prod_i^n \lambda(t_i) \exp\left(-\int_0^T \lambda(t)\,\mathrm{d}t\right). \tag{2.23}$$

However, this kind of a process still does not capture all the rich interactions we might expect to see in real-world. For example, the occurrence of one event could influence the future events. This is often observed in earthquake data and social networks since sending a message to another person increases the probability of getting a response soon after in a similar way how a single big earthquake is always followed by aftershocks. We can model this behavior by defining a TPP in an autoregressive manner. Let $p_i(t|\mathcal{H}_{i-1})$ denote the conditional probability density function of observing ith event at t, where $\mathcal{H}_{i-1} = \{t_1, \dots, t_{i-1}\}$ is the history of previous observations. Then the TPP can be defined with the sequence of conditional distributions $\{p_1(t), p_2(t|t_1), \dots\}$.

We now define the conditional survival function similarly: $S_i(t|\mathcal{H}_{i-1}) = \Pr\{t_i > t|\mathcal{H}_{i-1}\}$, specifying the probability that a new point arrives after t, having observed $i-1$ points already. Conditional survival and probability density functions are connected by:

$$S_i(t|\mathcal{H}_{i-1}) = 1 - F_i(t|\mathcal{H}_{i-1}) = 1 - \int_{t_{i-1}}^t p_i(s|\mathcal{H}_{i-1})\,\mathrm{d}s, \tag{2.24}$$

where $F_i(t|\mathcal{H}_{i-1})$ is the cumulative distribution function. Now, we define the hazard function as the probability that the first next event happens precisely at t:

$$h_i(t|\mathcal{H}_{i-1}) = p(t|\mathcal{H}_{i-1}, t_i \notin [t_{i-1}, t)) = \frac{p_i(t|\mathcal{H}_{i-1})}{S_i(t|\mathcal{H}_{i-1})}. \tag{2.25}$$

The hazard function can be interpreted as the failure rate in the field of survival analysis since it measures the ratio between the number of failures and the size of the alive population, both at time t. From Equations 2.24 and 2.25 we can derive the following connection: $S_i(t|\mathcal{H}_{i-1}) = \exp(-\int_{t_{i-1}}^t h_i(t|\mathcal{H}_{i-1}))$. From here, we define the *conditional intensity* as a piecewise function by concatenating the hazard functions:

$$\lambda^*(t) = \begin{cases} h_1(t), & 0 < t \le t_1 \\ h_i(t|\mathcal{H}_{i-1}), & t_{i-1} < t \le t_i. \end{cases} \tag{2.26}$$

The conditional intensity is interpreted as the expected rate of events, given the history. Finally, we can write down the likelihood of observing the realization $\{t_1, \dots, t_n\}$ on

interval $[0, T]$ using the conditional intensity:

$$p(t_1, \dots, t_n) = \prod_i^n \lambda^*(t_i) \exp\left(- \int_0^T \lambda^*(t) \, \mathrm{d}t \right). \tag{2.27}$$

Random time change. Since $\lambda^*(t) = \lambda(t)$ holds for a Poisson process, Equations 2.23 and 2.27 look exactly the same, as expected. However, there exists a deeper connection between *any* TPP specified by $\lambda^*(t)$ and a Poisson process. By transforming the time points with $t \mapsto \int_0^t \lambda^*(s) \, \mathrm{d}s$ we obtain a new process which is a Poisson process with unit rate [47, Theorem 7.4.I]. This is similar to the fact that the values of a cumulative density function of any random variable follow uniform distribution. We can use random time change to simulate realizations from TPPs, design scalable neural models and perform anomaly detection [241, 242].

Conventional models. Although we can define a TPP in many different but equivalent ways, conditional intensity is often preferred in literature. From the above definition it is clear that the intensity function only has to be non-negative. Therefore, the requirements are not too strict and we are free to pick from a large family of functions, however, the properties of the process will heavily depend on what kind of function we choose.

In case we want *bursty* behavior, as is the case with earthquake data, we need one event to trigger the next one. This is captured with a *self-exciting* or *Hawkes process* [99]:

$$\lambda^*(t) = \mu(t) + \alpha \sum_{t_j < t} \phi(t - t_j), \tag{2.28}$$

where $\mu(t)$ is some base intensity corresponding to an inhomogeneous Poisson process and ϕ is a kernel function, usually decaying in time. Therefore, the intensity is the highest immediately after the event occurs and decays to a base intensity μ with time. The resulting realization will consist of clusters of points.

On the other hand, if we want to capture the process which tends to have equally spaced time points we can use a *self-correcting process* [112]:

$$\lambda^*(t) = \exp\left(\mu t - \sum_{t_j < t} \alpha \right). \tag{2.29}$$

Unlike Hawkes process, the intensity decreases after each event and then rises with time.

Neural approaches. When we are given data that we know comes from a Hawkes process (Equation 2.28), we can ask the question of finding the optimal μ, α and ϕ that maximize the likelihood in Equation 2.27. As we have seen in Section 2.1, we can use gradient-based optimization for this. The next step towards pure neural models is specifying $\lambda^*(t)$ with a neural network. The model should input the full history $\mathcal{H}_{i-1}$ and the time point t, and output the value of the conditional intensity at t which should be non-negative.

An example architecture could be an RNN (Section 2.1.2) that uses previous time points as inputs and outputs the latent representation of the sequence $\boldsymbol{h} \in \mathbb{R}^h$. Then, a neural network with a non-negative final activation function (for example, softplus: $\sigma(x) = \log(1+\exp(x))$) takes $\boldsymbol{h}$ together with t and outputs $\lambda^*(t)$. The loss is the negative log-likelihood as defined in Equation 2.27. Computing the likelihood is not possible in closed-form due to an integral term which depends on a neural network. Therefore, one solution is to approximate the integral using Monte Carlo sampling and another is to find a way to compute the integral directly. In Chapter 3 we will present an approach that does the latter by specifying the probability density function instead, which allows closed-form likelihood and straightforward sampling.

Marked TPPs. Besides the time of the event, we often care about the type of the event as well. In social networks this could be information containing who is sending a message to whom. We call this additional information *marks*, and denote them with m. Marks are usually categorical features that we can predict by modeling the categorical distribution. Given history, the TPP then describes $p(t, m)$ which can be factorized as $p(t)p(m|t)$, that is, we separately model the TPP $p(t)$ and simply output $p(m|t)$ given the known time t. This straightforward implementation is equivalent to having K different TPPs, for K mark types, obtained by reweighing the base intensity $\lambda^*(t)$ with the categorical distribution $p(m|t)$. That is, for kth mark type we have the following conditional intensity: $\lambda_k^*(t) = p(m = k|t)\lambda^*(t)$. On the other hand, if we decide to factorize the joint distribution such that $p(t, m) = p(m)p(t|m)$, we then model K separate TPPs, one for each mark type and which cannot be expressed in as simple reweighed terms.

Spatial point processes. We can think of spatio-temporal processes as marked TPPs, where the mark is a value from $\mathbb{R}^d$ that usually corresponds to the location of the event. Omitting time, the spatial process specifies the probability of observing the set of points $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ where n is random. Similar to TPPs, we can define the counting process and use a Poisson distribution to model the number of points we observe on a domain $\mathcal{B} \subset \mathbb{R}^d$ (for TPPs this was the interval on $\mathbb{R}$). The realization of a homogeneous Poisson process can be again sampled by first sampling n from a Poisson distribution and then sampling n points uniformly on the domain $\mathcal{B}$. The inhomogeneous process replaces the uniform distribution with a probability density function $p(\boldsymbol{x})$ on $\mathcal{B}$. Since there is no canonical ordering of the points, as we had in the TPP case, we cannot define the conditional intensity function in the same way. Given the observed set of n points, the likelihood can be written as [48]:

$$p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) = n!p(n)\tilde{p}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n), \tag{2.30}$$

where $\tilde{p}$ is the permutation invariant density, that is, the order of points does not matter when evaluating it. We refer to data that lacks ordering as *exchangeable*.

2.2.2 Normalizing flows

In the previous section we saw through the random time change theorem that transforming the points which were sampled from one process returns the points which correspond to another process. This will hold for general distributions as well. Consider the uniform distribution $p(z) = 1$ on $[0, 1]$. Values $f(z) = 2z$ will also follow a uniform distribution, however, on twice the size of the interval and with half the density: $p(f(z)) = 0.5$ on $[0, 2]$. To make a generative model from this, we have to first fix the base distribution (for example, set it to be uniform) and then *learn the transformation* f such that the samples from the base distribution get transformed into samples from the target distribution.

The concept of transforming noise to data is not unheard of, in fact, this is what most generative models do, including generative adversarial networks [85], variational autoencoders (Section 2.2.3), and diffusion models (Section 2.2.4). What is specific to normalizing flows is that f is a one-to-one map which means we can always uniquely go from noise to data and vice versa. In the end, this allows us to compute the likelihood in closed-form using the change of variables formula (Equation 2.32), which is not possible when using the previously mentioned generative models.

First, we define a *Jacobian* matrix that describes local function's behavior, similar to a derivative. Given a function $f : \mathbb{R}^m \to \mathbb{R}^n$, the Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{n \times m}$ is defined as:

$$J_{ij} = \frac{\partial f(\boldsymbol{x})_i}{\partial \boldsymbol{x}_j}. \tag{2.31}$$

It contains all the partial derivatives of a function f and measures the best linear approximation of a function f at a point $\boldsymbol{x}$.

Change of variables. Let $\boldsymbol{z} \in \mathbb{R}^d$ be the sample from a base distribution $\boldsymbol{z} \sim q(\boldsymbol{z})$ and let $\boldsymbol{x} = f(\boldsymbol{z})$ be the result of an invertible and differentiable map (a diffeomorphism) $f : \mathbb{R}^d \to \mathbb{R}^d$. Then, $\boldsymbol{x}$ follows the distribution p which is defined as:

$$\begin{aligned} p(\boldsymbol{x}) &= q(\boldsymbol{z}) \left|\det \boldsymbol{J}_f(\boldsymbol{z})\right|^{-1} \\ &= q(f^{-1}(\boldsymbol{x})) \left|\det \boldsymbol{J}_{f^{-1}}(\boldsymbol{x})\right|, \end{aligned} \tag{2.32}$$

where $\boldsymbol{J}$ is the Jacobian consisting of all the partial derivatives as defined in Equation 2.31. The determinant measures the volume of a linear transformation and the Jacobian is the best linear approximation of f. Then, the right-hand term under the absolute value measures the change of density at a point. That is, if we have a transformation that preserves volume (like rotation or translation), its determinant of Jacobian will be equal to 1 since we do not squash or expand the space. In the introductory example, we had $f(z) = 2z$ meaning that the scaling term is $|\det \boldsymbol{J}_f(\boldsymbol{z})|^{-1} = |\frac{\mathrm{d}2z}{\mathrm{d}z}|^{-1} = 0.5$ as expected. Another useful property of Equation 2.32 is that it goes both ways, that is, if we know the inverse f^{-1}, we can compute the density $p(x)$, given x and $q(z)$.

Inspired by successes of neural networks we would like f and f^{-1} to be expressive functions, however, it is not immediately obvious how to design a neural network that is both expressive and invertible. Further, to maximize the likelihood we need to compute

the determinant of the Jacobian which generally has $O(d^3)$ time cost. This operation is in practice intractable if data is high-dimensional, as is the case with images or time series with many features. To this end, previous works have designed models that can be easily inverted and which have $O(d)$ cost for evaluating Equation 2.32.

Coupling normalizing flow [58, 59] relies on a fact that we can transform one part of data conditioned on another part, while keeping the second part intact. Let us partition the dimensions of $\boldsymbol{z}$ into two disjoint sets: A and B, $A \cup B = \{1, \ldots, d\}$, $A \cap B = \emptyset$. The transformation is then defined as:

$$\begin{aligned} \boldsymbol{x}^{(A)} &= \boldsymbol{z}^{(A)} \exp(\phi(\boldsymbol{z}^{(B)})) + \psi(\boldsymbol{z}^{(B)}) \\ \boldsymbol{x}^{(B)} &= \boldsymbol{z}^{(B)}, \end{aligned} \tag{2.33}$$

where $\phi, \psi : \mathbb{R}^{|A|} \to \mathbb{R}^{|A|}$ are *arbitrary* neural networks. Note that the Jacobian of this transformation is triangular with ones on the diagonal corresponding to the B-indexed values and $\exp(\phi(\boldsymbol{z}^{(B)}))$ on the diagonal for A-indexed values. The determinant of a triangular matrix is the product of the values on the diagonal. This transformation works because we can treat $\boldsymbol{z}^{(B)}$ as constant and then the transformation of $\boldsymbol{x}^{(A)}$ reduces to an affine transform which is easy to invert:

$$\begin{aligned} \boldsymbol{z}^{(A)} &= (\boldsymbol{x}^{(A)} - \psi(\boldsymbol{x}^{(B)})) \exp(-\phi(\boldsymbol{x}^{(B)})) \\ \boldsymbol{z}^{(B)} &= \boldsymbol{x}^{(B)}. \end{aligned} \tag{2.34}$$

Applying only one coupling transformation would keep a large chunk of data unchanged. Luckily, we can treat the resulting distribution $p(\boldsymbol{x})$ as our new starting distribution to apply another normalizing flow on top of it. That is, if we have a composition of invertible functions, the whole transformation is invertible, and if we can compute the determinant of the Jacobian for all of them, the total determinant is their product. Stacking multiple transformations is a common approach to increase the expressivity.

Autoregressive normalizing flow [203, 135] applies a similar idea to coupling transformation but with a more fine grained partition of dimensions. Since every distribution can be factorized as $p(x_1, \ldots, x_d) = p(x_1)p(x_2|x_1) \ldots p(x_d|x_1, \ldots, x_{d-1})$, we can assign an arbitrary ordering to the dimensions of $\boldsymbol{x} \in \mathbb{R}^d$ and model the conditional distributions $p(x_i|\boldsymbol{x}_{<i})$. In the spirit of normalizing flows, we again start with the base distribution $q(\boldsymbol{z})$ and apply the transformation:

$$\begin{aligned} x_1 &= z_1 \\ x_i &= z_i \exp(\phi(\boldsymbol{z}_{<i})) + \psi(\boldsymbol{z}_{<i}), \quad i > 1, \end{aligned} \tag{2.35}$$

where $\phi_i, \psi_i : \mathbb{R}^{i-1} \to \mathbb{R}$ are arbitrary neural networks, as in Equation 2.33. Note that we can compute this transformation in parallel, whereas the inverse is strictly sequential, that is, we have to apply the transformations in a certain order and use the result of a

previous transform to apply the next one:

$$\begin{aligned} z_1 &= x_1 \\ z_2 &= (x_2 - \psi_2(z_{<2})) \exp(-\phi_2(z_{<2})) \\ z_3 &= (x_3 - \psi_3(\boldsymbol{z}_{<3})) \exp(-\phi_3(\boldsymbol{z}_{<3})) \\ &\vdots \\ z_d &= (x_d - \psi_d(\boldsymbol{z}_{<d})) \exp(-\phi_d(\boldsymbol{z}_{<d})). \end{aligned} \tag{2.36}$$

Spline flow [63]. We can again stack multiple layers of autoregressive flows with different orderings of the dimensions. In both the coupling and autoregressive flows we designed the transformation with an affine function. However, we can also find other functions that are invertible and have derivatives that are easy to evaluate. Since the conditional transformations can be viewed as applying a one-dimensional mapping, it is sufficient to find a monotonic function with the desired computational properties. One choice is using splines, which are specified as piecewise polynomials. Therefore, the conditioning neural network takes in $\boldsymbol{z}^{(B)}$ in a coupling flow or $\boldsymbol{z}_{<i}$ in an autoregressive flow and outputs the spline parameters corresponding to the polynomial coefficients. The value x_i is the output of the polynomial evaluated at z_i and the derivative is computed with the polynomial of a lower degree that is easy to find analytically.

Choosing the direction. As we have seen in the case of autoregressive flows, one direction of the flow (Equation 2.35) is faster to compute than the other (Equation 2.36). Let us consider a case where we are given data and want to learn its distribution with a normalizing flow using maximum likelihood, to later perform anomaly detection by evaluating the density on new data. In this case, we only need one direction—the one that computes the likelihood, that is, we only need the inverse transformation f^{-1}. Then, we can parameterize f^{-1} with a function that is faster to evaluate: Equation 2.35. The inverse of this still exists, it is precisely Equation 2.36, and can be now used for sampling.

On the other hand, if we wish to only sample new points and use a loss that is sampling-based, we can parameterize the forward direction. This is something that is quite often used in variational inference (Section 2.2.3) and was the original motivation for designing such computationally *asymmetric* models [135]. Some architectures, such as coupling flows, have the same computational cost for both directions while others do not have closed-form inverse at all [224].[5]

We will use normalizing flows for modeling TPP densities in Chapter 3 and to define an invertible mapping in Chapter 5, as well as to demonstrate an application in spatio-temporal modeling in Section 5.2.3.

Continuous normalizing flows [35] define how the density changes as we change the random variable with an ordinary differential equation (Equation 2.3). Let $\boldsymbol{z}(t_0) \in \mathbb{R}^d$

[5]Note the inverse does exists, we just cannot write it down as a simple formula.

be a random variable following $\boldsymbol{z}(t_0) \sim q(\boldsymbol{z}(t_0))$ and let $\frac{\mathrm{d}\boldsymbol{z}(t)}{\mathrm{d}t} = f(t, \boldsymbol{z}(t))$ define the immediate transformation which, in turn, defines the instantaneous change of variables:

$$\frac{\partial}{\partial t} \log p(\boldsymbol{z}(t)) = -\operatorname{Tr}\left(\frac{\partial f}{\partial \boldsymbol{z}(t)}\right). \tag{2.37}$$

Integrating f from t_0 to t_1 gives a solution $\boldsymbol{x} = \boldsymbol{z}(t_1)$ which follows the distribution:

$$\log p(\boldsymbol{x}) = \log q(\boldsymbol{z}(t_0)) - \int_{t_0}^{t_1} \operatorname{Tr}\left(\frac{\partial f}{\partial \boldsymbol{z}(t)}\right) \mathrm{d}t. \tag{2.38}$$

Function f can be specified with any neural network that gives unique ODE solutions (see Section 2.1.1). Since t_0 and t_1 do not have any assigned meaning to them, they are usually set to 0 and 1, respectively. In contrast to previous flows, here we have an option to use arbitrary architectures $f : \mathbb{R}^d \to \mathbb{R}^d$ with "dense Jacobians" because we avoid computing the determinant and instead evaluate the trace. The inverse can be obtained easily, by integrating in reverse from t_1 to t_0.

The bottleneck of this approach is evaluating the diagonal elements of the Jacobian. Since we are encouraged to use arbitrary neural networks, the Jacobian has to be computed using automatic differentiation. However, due to the way autodiff is implemented [207], it has to be computed separately for each dimension resulting in an overall $O(d^2)$ cost. To add on this, we have to perform this same computation at each solver step. An alternative is performing the estimation of the trace [109]; given a random variable $\boldsymbol{\epsilon} \in \mathbb{R}^d$ with mean zero and identity covariance it follows that:

$$\operatorname{Tr}(\boldsymbol{J}) = \mathbb{E}_{\boldsymbol{\epsilon}}[\boldsymbol{\epsilon}^T \boldsymbol{J} \boldsymbol{\epsilon}]. \tag{2.39}$$

Although reverse-mode autodiff seemingly hindered the computation of the Jacobian, it now enables computing the term $\boldsymbol{\epsilon}^T \boldsymbol{J}$, since modern frameworks excel when dealing with vector-Jacobian products. The cost falls down to $O(d)$. However, it gives us a noisy estimate so when we want to exactly compute the density (during testing, for example), we again need to use the more expensive Jacobian computation.

In Section 3.4.1 we propose an application for continuous normalizing flows in modeling spatial point processes since CNFs can easily implement permutation invariant densities. We alleviate the issue of computing the Jacobian by decoupling the transformations in such a way that we can easily compute the diagonal elements exactly and with $O(d)$ cost.

Universal approximation of normalizing flows is answering the question of whether a generative model defined with the transformation f can capture any distribution $p(\boldsymbol{x})$ given a base distribution $p(\boldsymbol{z})$. Since for a pair of well-behaved distributions $p(\boldsymbol{x})$ and $p(\boldsymbol{z})$ there always exists a diffeomorphism between the two [202], the question of universality becomes whether the specific transformation can approximate any invertible function. For example, autoregressive flows are universal approximators if they can represent any conditional CDF which is easy to achieve with arbitrary neural networks but not so with a single affine flow layer (Equation 2.35). In general, most normalizing flows answer affirmatively to the question of universal approximation [264, 265], however, this does not give any guarantees into how they will behave in practice.

2.2.3 Variational inference

A common way to specify data generation is with a latent variable model, a two step process where we first sample a latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$ and then sample data conditioned on it $\boldsymbol{x} \sim p(\boldsymbol{x}|\boldsymbol{z})$. For example, we can first pick the content of an image and then draw the image based on this. Given data, we are often interested in uncovering (*inferring*) the latent variable, that is, we want to find a distribution $p(\boldsymbol{z}|\boldsymbol{x})$. In time series data, one might be interested in finding the states that the system is in, for example, server logs can correspond to normal operations or anomalous behavior [241].

The main issue is the intractability of the true posterior distribution $p(\boldsymbol{z}|\boldsymbol{x})$ which is why we have to find a way to approximate it. One established way is to sample from the posterior using Markov chain Monte Carlo (MCMC) and use the empirical distribution of the samples [227]. An alternative is to approximate the posterior with some distribution $q(\boldsymbol{z})$, called a variational distribution, that we learn together with the model likelihood $p(\boldsymbol{x}|\boldsymbol{z})$. Although MCMC is an exact method, it is computationally expensive so variational inference is often the preferred choice when dealing with large datasets, as is the case in modern machine learning.

We predefine a family of distributions $\mathcal{Q}$, where each $q_{\boldsymbol{\vartheta}} \in \mathcal{Q}$ is parameterized with a set of learnable parameters $\boldsymbol{\vartheta}$. The aim is to find an optimal $q^\star_{\boldsymbol{\vartheta}} \in \mathcal{Q}$ that minimizes the Kullback-Leibler divergence to the true posterior, that is, minimizes the distance between the two distributions:

$$\begin{aligned} q^\star_{\boldsymbol{\vartheta}} &= \arg\min_{q_{\boldsymbol{\vartheta}} \in \mathcal{Q}} D_{\mathrm{KL}}(q_{\boldsymbol{\vartheta}}(\boldsymbol{z}) \| p(\boldsymbol{z}|\boldsymbol{x})) \\ &= \arg\min_{q_{\boldsymbol{\vartheta}} \in \mathcal{Q}} \int q_{\boldsymbol{\vartheta}}(\boldsymbol{z}) \log \frac{q_{\boldsymbol{\vartheta}}(\boldsymbol{z})}{p(\boldsymbol{z}|\boldsymbol{x})} \, \mathrm{d}\boldsymbol{z} \\ &= \arg\min_{q_{\boldsymbol{\vartheta}} \in \mathcal{Q}} \mathbb{E}_{q_{\boldsymbol{\vartheta}}}[\log q_{\boldsymbol{\vartheta}}(\boldsymbol{z})] - \mathbb{E}_{q_{\boldsymbol{\vartheta}}}[\log p(\boldsymbol{z}|\boldsymbol{x})]. \end{aligned} \tag{2.40}$$

As we can see, the goal is to approximate the posterior by learning it. By choosing different families $\mathcal{Q}$ we trade-off the expressive power and the approximation capabilities. For example, we can use a normal distribution or a normalizing flow for the approximation.

ELBO. Again, we do not have acess to $p(\boldsymbol{z}|\boldsymbol{x})$ but we can expand the KL divergence:

$$D_{\mathrm{KL}}(q_{\boldsymbol{\vartheta}}(\boldsymbol{z}) \| p(\boldsymbol{z}|\boldsymbol{x})) = \underbrace{\mathbb{E}_{q_{\boldsymbol{\vartheta}}}[\log q_{\boldsymbol{\vartheta}}(\boldsymbol{z})] - \mathbb{E}_{q_{\boldsymbol{\vartheta}}}[\log p(\boldsymbol{z}, \boldsymbol{x})]}_{-\mathcal{L}(q)} + \log p(\boldsymbol{x}), \tag{2.41}$$

where $\log p(\boldsymbol{x})$, called *the evidence*, does not depend on q so it escapes the expectation. Since the KL divergence is always non-negative, the first two terms, which we denote by $\mathcal{L}$, form an evidence lower bound (ELBO). That means that regardless of which q we choose, the ELBO will always be lower or equal to $\log p(\boldsymbol{x})$, while the gap corresponds exactly to the KL divergence between q and the true posterior. Optimizing ELBO finds the best q that reduces this gap. Further expanding the ELBO:

$$\begin{aligned} \mathcal{L}(q) &= \mathbb{E}_{q_{\boldsymbol{\vartheta}}}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})] + \mathbb{E}_{q_{\boldsymbol{\vartheta}}}[\log p(\boldsymbol{z})] - \mathbb{E}_{q_{\boldsymbol{\vartheta}}}[\log q_{\boldsymbol{\vartheta}}(\boldsymbol{z})] \\ &= \mathbb{E}_{q_{\boldsymbol{\vartheta}}}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})] - D_{\mathrm{KL}}(q_{\boldsymbol{\vartheta}}(\boldsymbol{z}) \| p(\boldsymbol{z})) \end{aligned} \tag{2.42}$$

shows that we want to maximize the likelihood and minimize the divergence between the variational distribution $q_{\boldsymbol{\vartheta}}(\boldsymbol{z})$ and the latent prior $p(\boldsymbol{z})$. To recap, given data $\boldsymbol{x}$ and having picked the parameterization of the model $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$, the prior $p(\boldsymbol{z})$, and the approximate posterior $q_{\boldsymbol{\vartheta}}(\boldsymbol{z})$, we can learn the parameters $\boldsymbol{\theta}$ and $\boldsymbol{\vartheta}$ by maximizing the ELBO (Equation 2.42).

Variational autoencoder. Performing the inference on the whole dataset $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ gives us a distribution over random variables $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_N$. It is common to simplify the learning of q by using the so-called mean-field assumption which keeps the individual latent variable distributions independent $q(\boldsymbol{z}_1, \ldots, \boldsymbol{z}_N) = \prod_i q_i(\boldsymbol{z}_i)$. However, we still need to learn the parameters of all N distributions and, additionally, this formulation forces us to retrain every time we encounter a new data point $\boldsymbol{x}_{N+1}$.

The solution is to use *amortized* inference where we specify the variational distribution conditioned on data $q_{\boldsymbol{\vartheta}}(\boldsymbol{z}|\boldsymbol{x})$. For example, we might use a neural network that outputs the mean and the variance of a normal distribution $q_{\boldsymbol{\vartheta}}(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\mu(\boldsymbol{x}), \sigma(\boldsymbol{x}))$. Together with a network that defines the distribution $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$, this forms a variational autoencoder (VAE). We can again learn the parameters of the encoder $q_{\boldsymbol{\vartheta}}(\boldsymbol{z}|\boldsymbol{x})$ and the decoder $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ by maximizing the same ELBO objective (Equation 2.42).

Since we usually learn by backpropagation and gradient descent, we encounter another hurdle—the gradient of the expectation $\nabla_{\boldsymbol{\vartheta}} \mathbb{E}_{q_{\boldsymbol{\vartheta}}}[f_{\boldsymbol{\theta}}(\boldsymbol{z})]$ is not the same as the expectation of the gradient $\mathbb{E}_{q_{\boldsymbol{\vartheta}}}[\nabla_{\boldsymbol{\vartheta}} f_{\boldsymbol{\theta}}(\boldsymbol{z})]$. Luckily, if we find a way to sample from some fixed distribution and reparameterize sampling from q in terms of this distribution, we can mitigate all the issues. As a concrete example, consider parameterizing q as a normal distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}\boldsymbol{I})$, where $\boldsymbol{\vartheta} = \{\boldsymbol{\mu}, \boldsymbol{\sigma}\}$ are learnable. We can sample from q using:

1. $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$,
2. $\boldsymbol{z} = T_{\boldsymbol{\vartheta}}(\boldsymbol{\epsilon}) = \boldsymbol{\mu} + \sqrt{\boldsymbol{\sigma}}\boldsymbol{\epsilon}$.

Then, $\mathbb{E}_{\boldsymbol{z} \sim q_{\boldsymbol{\vartheta}}}[f_{\boldsymbol{\theta}}(\boldsymbol{z})]$ is the same as $\mathbb{E}_{\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})}[f_{\boldsymbol{\theta}}(T_{\boldsymbol{\vartheta}}(\boldsymbol{\epsilon}))]$ and finally:

$$\nabla_{\boldsymbol{\vartheta}} \mathbb{E}_{q_{\boldsymbol{\vartheta}}}[f_{\boldsymbol{\theta}}(\boldsymbol{z})] = \mathbb{E}_{\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})}[\nabla_{\boldsymbol{\vartheta}} f_{\boldsymbol{\theta}}(T_{\boldsymbol{\vartheta}}(\boldsymbol{\epsilon}))].$$

The above procedure is known as the *reparameterization trick* which allows us to take gradients with sampling-based objectives.

2.2.4 Generative diffusion

So far we have generated new data by drawing from a known distribution and *denoised* it to get a data sample using neural networks (Sections 2.2.2 and 2.2.3). That is, we had a single deterministic transformation (neural network) between noise and data $\boldsymbol{z} \mapsto \boldsymbol{x}$ or noise and data distribution $\boldsymbol{z} \mapsto p(\boldsymbol{x})$. Instead, in this section, we will approach denoising as an iterative process where samples are refined over many steps starting with pure noise and ending with target distribution—we will now have a sequence of latent variables which converge to data distribution $\boldsymbol{z}_N \to \boldsymbol{z}_{N-1} \to \cdots \to \boldsymbol{z}_1 \to \boldsymbol{x}$. To do so, we will first see how it is possible to sample new points by maximizing the likelihood using the score, similar to how we perform gradient descent to minimize the loss.

Score is defined as the gradient of the log-density: $s(\boldsymbol{x}) = \nabla_{\boldsymbol{x}} \log p(\boldsymbol{x})$. Similarly to gradient-based optimization, we can imagine starting at some random point $\boldsymbol{x}$ which has low probability and "optimizing" $s(\boldsymbol{x})$ using gradient ascent to reach a high probability point. This gives us a way to sample new points from the data distribution. At first, it is not clear how could learning $\nabla_{\boldsymbol{x}} \log p(\boldsymbol{x})$ be easier, or even possible, if we cannot learn $\log p(\boldsymbol{x})$ directly. To see why, let us first consider an unnormalized distribution $\bar{p}(\boldsymbol{x}) = Zp(\boldsymbol{x})$, $Z = \int \bar{p}(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}$. Then, the score for both of these functions (p and $\bar{p}$) is the same. Finally, if we parameterize the unnormalized density $\bar{p}_{\boldsymbol{\theta}}(\boldsymbol{x})$; for instance, as an energy-based model [150], our goal becomes optimizing the score-matching loss [111]:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2} \int p(\boldsymbol{x}) \| s_{\boldsymbol{\theta}}(\boldsymbol{x}) - s(\boldsymbol{x}) \|^2 \, \mathrm{d}\boldsymbol{x}, \tag{2.43}$$

where $s_{\boldsymbol{\theta}}(\boldsymbol{x})$ is model's score, and $s(\boldsymbol{x})$ is the true score. This is the same as minimizing the Fisher divergence. Note that if we only care about generating new samples we do not have to specify $\bar{p}_{\boldsymbol{\theta}}$, rather we can parameterize $s_{\boldsymbol{\theta}}$ directly. If $\mathcal{L}(\boldsymbol{\theta}) = 0$, the distributions are equal and the model maximizes the likelihood. Again, $s(\boldsymbol{x})$ is inaccessible but Hyvärinen [111] show that Equation 2.43 can be rewritten as:

$$\mathcal{L}(\boldsymbol{\theta}) = \int p(\boldsymbol{x}) \left(\mathrm{Tr}(\nabla_{\boldsymbol{x}} s_{\boldsymbol{\theta}}(\boldsymbol{x})) + \frac{1}{2} \| s_{\boldsymbol{\theta}}(\boldsymbol{x}) \|^2 \right) \mathrm{d}\boldsymbol{x}, \tag{2.44}$$

where they apply the partial integration trick which removes the true score function from the equation. Although we now have a way to perform score-matching using only the model's predicted score together with the samples from p (using true data to get a Monte Carlo estimate), evaluating Equation 2.44 is expensive due to the Hessian term. One way to get around this is to use sliced score matching [251], which is using a similar trick to approximate the Hessian as continuous normalizing flows did to approximate the Jacobian (Section 2.2.2). Finally, after we have obtained the trained score model $s_{\boldsymbol{\theta}}(\boldsymbol{x})$, we can sample from $p(\boldsymbol{x})$ using Langevin dynamics [204], starting from random $\boldsymbol{x}_N$ and applying the following update N times until we reach $\boldsymbol{x}_0 \sim p(\boldsymbol{x})$:

$$\boldsymbol{x}_{n-1} = \boldsymbol{x}_n + \delta_n \nabla_{\boldsymbol{x}_n} s_{\boldsymbol{\theta}}(\boldsymbol{x}_n) + \sqrt{2\delta_n} \boldsymbol{\epsilon} \tag{2.45}$$

where δ_n is step size and $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. This is similar to gradient ascent but we also add a small amount of noise at each step. The above update is actually an Euler discretization of the Langevin stochastic differential equation:

$$\mathrm{d}\boldsymbol{x}_t = \nabla_{\boldsymbol{x}_t} \log p(\boldsymbol{x}_t) \, \mathrm{d}t + \sqrt{2} \, \mathrm{d}W_t. \tag{2.46}$$

In general, for an SDE defined with $\mathrm{d}\boldsymbol{x} = f \, \mathrm{d}t + g \, \mathrm{d}W_t$, we say that f is a drift function (with the same interpretation as in ordinary differential equations) and g is the diffusion function, while $\mathrm{d}W_t$ denotes adding infinitesimal amount of noise to each $\mathrm{d}t$-sized step. The term W_t is a standard Brownian motion (Wiener process) which is a stochastic process starting at zero, with independent increments, each with mean zero and variance equal to the time difference: $W_t - W_s \sim \mathcal{N}(0, t - s)$. Since SDE is not deterministic, its solutions will follow some distribution that we denote with $q(\boldsymbol{x}, t)$. It can be shown that $q(\boldsymbol{x}, t)$ of Equation 2.46 will become stationary and equal $p(\boldsymbol{x})$ as $t \to \infty$ [189, 278].

Noise perturbations. Another issue occurs, besides computation complexity, that is less obvious—when we want to sample from our model we will almost surely start at a low density region and climb towards the high density sample. But our model has not seen many (or any) examples with low density that would correspond to pure noise. This can be seen directly from Equation 2.43 which weighs these terms down with $p(\boldsymbol{x})$ so learning the correct score on low density regions is hard. As a result, the learned score in some locations will point in random directions and model samples will be of low quality. One solution is to learn with multiple noise perturbations, as presented in the following.

We start with a clean data point $\boldsymbol{x}$ and index it as $\boldsymbol{x}_0$. We add a small amount of noise to $\boldsymbol{x}_0$ to produce $\boldsymbol{x}_1$, for example, $\boldsymbol{x}_1 = \boldsymbol{x}_0 + \epsilon$, $\epsilon \sim \mathcal{N}(0, 10^{-4})$. We continue doing this for N steps, increasing the noise size, to obtain progressively noisier values $\boldsymbol{x}_2, \boldsymbol{x}_3, \ldots, \boldsymbol{x}_N$, where the last one is pure noise, containing no information about the original data. We can now model the *conditional* score $s_{\boldsymbol{\theta}}(\boldsymbol{x}, n)$, which depends on the noising step n. Since the final value $\boldsymbol{x}_N$ comes from the known distribution, we can sample from it. The learned score $s_{\boldsymbol{\theta}}(\boldsymbol{x}, N)$ has good coverage compared to the previous approach, thus, the gradient is correct and not just a random direction in space.

The usefulness of this approach can be seen when we start sampling, going from $\boldsymbol{x}_N$ to $\boldsymbol{x}_0$. We first sample initial noise value from some fixed distribution, for example uniform, and perform the Langevin dynamics to obtain $\boldsymbol{x}_N$. We then repeat this for $\boldsymbol{x}_{N-1}$ using $\boldsymbol{x}_N$ as the initial sample. The score is well defined since there is significant overlap between $p(\boldsymbol{x}_N)$ and $p(\boldsymbol{x}_{N-1})$. We continue for $n = N-2, \ldots, 2, 1$ to reach the sample from the data distribution $\boldsymbol{x}_0$. The total number of steps N should be large enough that the consecutive distributions are similar, the first step should be close to the true data distribution while the last should be close to pure noise from which we can sample.

Note that previously we sampled from $p(\boldsymbol{x})$ using a procedure that iteratively obtains high density points $\boldsymbol{x}_N, \boldsymbol{x}_{N-1}, \ldots$; however, all of $\boldsymbol{x}_n$ belonged to the same distribution and we evaluated their score with respect to that distribution. Now, we have N distributions and sample from each independently, where the result of sampling from nth distribution acts a starting point for sampling from $(n-1)$th distribution.

Stochastic differential equation noising can be seen as a generalization of the above approach when the number of noising steps approaches infinity. In particular, we define the noising process with a predetermined SDE:

$$\mathrm{d}\boldsymbol{x} = f(\boldsymbol{x}, t)\,\mathrm{d}t + g(t)\,\mathrm{d}\boldsymbol{w}. \tag{2.47}$$

The simplest example of Equation 2.47 is $\mathrm{d}\boldsymbol{x} = \mathrm{d}\boldsymbol{w}$ which just adds increasing amounts of noise to $\boldsymbol{x}$ over time. As before, the forward process is fixed and known so the SDE has to be chosen beforehand. Previous works tried to find those that have desirable properties, for example, where the distribution of $\boldsymbol{x}_t$ approaches unit normal as $t \to \infty$ [252]. So we have only seen how to add noise to data, luckily, the following result shows us how to reverse this process. Given an SDE as in Equation 2.47, there is a reverse process [2]:

$$\mathrm{d}\boldsymbol{x} = [f(\boldsymbol{x}, t) - g(t)^2 \nabla_{\boldsymbol{x}} \log p_t(\boldsymbol{x})]\,\mathrm{d}t + g(t)\,\mathrm{d}\boldsymbol{w}, \tag{2.48}$$

where $\nabla_{\boldsymbol{x}} \log p_t(\boldsymbol{x})$ is the score of the conditional noising distribution p_t, indexed by the diffusion step t, similar to previous indexing with discrete steps n. This shows us that once we specify the forward SDE, we can reverse it if we know the score. Since we pick the forward SDE ourselves, the true score with respect to the noisy point $\boldsymbol{x}_t$ can be found in closed-form given the clean data point $\boldsymbol{x}_0$ that produced $\boldsymbol{x}_t$. We can then use the loss from Equation 2.43 directly to learn the true score with our model $s_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$.

To recap, using score-matching with multiple noise perturbations and SDEs allows us to learn complicated distributions by utilizing the following nice properties:

- The distribution of the final noisy value $\boldsymbol{x}_t$ as $t \to \infty$ is known and we can sample from it in closed-form.
- The true score is known and depends on the clean data $\boldsymbol{x}_0$, thus, can be used to learn it with model $s_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$ that depends only on $\boldsymbol{x}_t$.
- The reverse process exists and depends on the terms that we predefined (functions f and g), and on the learned score, meaning we can start from pure noise and get the sample from a data distribution by using the reverse SDE.

Denoising diffusion. To solve an SDE accurately one has to use numerical solvers which are simply discretizing the time domain and performing a fixed amount of updates. This could again be viewed as adding noise for N steps, similar to what we described before. The forward (noising) process is called diffusion and can be written down as a Markov chain; here we show one specific parameterization choice [102]:

$$q(\boldsymbol{x}_n|\boldsymbol{x}_{n-1}) = \mathcal{N}(\sqrt{1-\beta_n}\boldsymbol{x}_{n-1}, \beta_n \boldsymbol{I}) \tag{2.49}$$

where β_n is an increasing sequence of noise scales such that $\boldsymbol{x}_n$ is *more noisy* than $\boldsymbol{x}_{n-1}$, and $\boldsymbol{x}_0$ again corresponds to clean data. Since we use normal distribution it is possible to specify $q(\boldsymbol{x}_n|\boldsymbol{x}_0)$ and $q(\boldsymbol{x}_{n-1}|\boldsymbol{x}_n, \boldsymbol{x}_0)$ directly. The loss is the negative ELBO, that is, we optimize the lower bound on the likelihood. We will revisit both the continuous (SDE-based) and the discrete diffusion (Equation 2.49) in Chapter 6 in the context of modeling continuous functions.

3 Temporal Point Processes

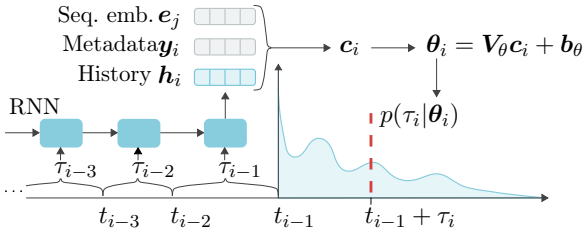


Figure 3.1: In our approach the parameters of a temporal point process are generated based on the history. The TPP itself is specified as a density model instead of using an intensity.

Visits to hospitals, purchases in e-commerce systems, financial transactions, posts in social media; various forms of human activity can be represented as discrete events happening at irregular intervals. The framework of temporal point processes is a natural choice for modeling such data. By combining temporal point process models with deep learning, we can design algorithms able to learn complex behavior from real-world data. Designing such models, however, usually involves trade-offs along the following dimensions:

- Flexibility: can the model approximate any distribution?
- Efficiency: can the likelihood function be evaluated in closed-form?
- Ease of use: is sampling and computing summary statistics easy?

Existing methods [61, 180, 199] that are defined in terms of the conditional intensity function typically fall short in at least one of these categories.

Instead of modeling the intensity function, we suggest treating the problem of learning in temporal point processes as an instance of conditional density estimation. By using tools from neural density estimation [21, 224], we can develop methods that have all of the above properties. To summarize, our contributions are the following:

- We connect the fields of temporal point processes and neural density estimation. We show how normalizing flows can be used to define flexible and theoretically sound models for learning in temporal point processes.
- We propose a simple mixture model that performs on par with the state-of-the-art methods. Thanks to its simplicity, the model permits closed-form sampling and moment computation.
- We show through a wide range of experiments how the proposed models can be used for prediction, conditional generation, sequence embedding and training with missing data.

	Exponential intensity	Neural Hawkes	Fully NN	Normalizing Flows	Mixture Distribution
Closed-form likelihood	✓	✗	✓	✓	✓
Flexible	✗	✓	✓	✓	✓
Closed-form $\mathbb{E}[\tau]$	✗	✗	✗	✗	✓
Closed-form sampling	✓	✗	✗	✗	✓

Table 3.1: Comparison of neural TPP models that encode history with an RNN.

3.1 Background

In this section we present the background on *neural* temporal point processes while reusing the notation from Section 2.2.1.

Definition. A temporal point process (TPP) is a random process whose realizations consist of a sequence of strictly increasing arrival times $\mathcal{T} = \{t_1, \dots, t_N\}$. A TPP can equivalently be represented as a sequence of strictly positive inter-event times $\tau_i = t_i - t_{i-1} \in \mathbb{R}_+$. Representations in terms of t_i and τ_i are isomorphic—we will use them interchangeably throughout this chapter. The traditional way of specifying the dependency of the next arrival time t on the history $\mathcal{H}_t = \{t_j \in \mathcal{T} : t_j < t\}$ is using the conditional intensity function $\lambda^*(t) := \lambda(t|\mathcal{H}_t)$ (Equation 2.26). Given the conditional intensity function, we can obtain the conditional probability density function (PDF) of the time τ_i until the next event by integration as:

$$p^*(\tau_i) := p(\tau_i|\mathcal{H}_{t_i}) = \lambda^*(t_{i-1} + \tau_i) \exp\left(-\int_0^{\tau_i} \lambda^*(t_{i-1} + s)\,\mathrm{d}s\right), \tag{3.1}$$

which is equivalent to Equation 2.27 but for a single event, given history.

Learning temporal point processes. Conditional intensity functions provide a convenient way to specify point processes with a simple predefined behavior, such as self-exciting [99] and self-correcting [112] processes as shown in Equations 2.28 and 2.29. Intensity parametrization is also commonly used when learning a model from the data—given a parametric intensity function $\lambda^*_{\boldsymbol{\theta}}(t)$ and a sequence of observations $\mathcal{T}$, the parameters $\boldsymbol{\theta}$ can be estimated by maximizing the log-likelihood:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \sum_i \log p^*_{\boldsymbol{\theta}}(\tau_i) = \arg\max_{\boldsymbol{\theta}} \left[\sum_i \log \lambda^*_{\boldsymbol{\theta}}(t_i) - \int_0^T \lambda^*_{\boldsymbol{\theta}}(s)\,\mathrm{d}s\right]. \tag{3.2}$$

The main challenge of such intensity-based approaches lies in choosing a good parametric form for $\lambda^*_{\boldsymbol{\theta}}(t)$. This usually involves the following trade-off: For a *simple* intensity function [61, 107], the integral $\Lambda^*(\tau_i) := \int_0^{\tau_i} \lambda^*(t_{i-1} + s)\,\mathrm{d}s$ has a closed-form, which makes the log-likelihood easy to compute. However, such models usually have limited expressiveness. A more sophisticated intensity function [180] can better capture the dynamics of the system, but computing log-likelihood will require approximating the integral using Monte Carlo.

Recurrent marked temporal point processes [61] fall in the category of models with closed-form likelihood. It uses an RNN to encode the event history into a vector $\boldsymbol{h}_i$. The history embedding $\boldsymbol{h}_i$ is then used to define the conditional intensity, for example, using the *constant intensity* model [107, 155]:

$$\lambda^*(t_i) = \exp(\boldsymbol{v}^T \boldsymbol{h}_i + b), \tag{3.3}$$

or the more flexible *exponential intensity* model [61, 271]:

$$\lambda^*(t_i) = \exp(w(t_i - t_{i-1}) + \boldsymbol{v}^T \boldsymbol{h}_i + b). \tag{3.4}$$

By considering the conditional distribution $p^*(\tau)$ of the two models, we can better understand their properties. Constant intensity corresponds to an exponential distribution, and exponential intensity corresponds to a Gompertz distribution (see Appendix A.2). Since these distributions are *only* unimodal, they might not be able to capture more complicated intensity functions.

FullyNN [199] is a flexible fully neural network intensity model, where they model the cumulative intensity function $\Lambda^*(\tau)$ with a neural network. It was proposed as a flexible, yet computationally tractable model for TPPs. The function Λ^* converts τ into an exponentially distributed random variable with unit rate [220], similarly to how normalizing flows model (Section 2.2.2) might convert τ into a random variable with a simple distribution (see Section 3.2). However, due to a suboptimal choice of the network architecture, the PDF of the FullyNN model does not integrate to 1, and the model assigns non-zero probability to *negative* inter-event times. We discuss this in more detail in Appendix A.3. Moreover, similar to other flow-based models, sampling from the FullyNN model requires iterative root finding.

This work. We show that the drawbacks of the existing approaches can be remedied by looking at the problem of learning TPPs from a different angle. Instead of modeling the conditional intensity $\lambda^*(t)$, we suggest to directly learn the conditional distribution $p^*(\tau)$. Modeling distributions with neural networks is a well-researched topic, that, surprisingly, was not usually discussed in the context of TPPs. By adopting this alternative point of view, we are able to develop new theoretically sound and effective methods (Section 3.2), as well as better understand the existing approaches (Section 3.2.6).

3.2 Models

We develop several approaches for modeling the distribution of inter-event times. First, we assume for simplicity that each inter-event time τ_i is conditionally independent of the history, given the model parameters (that is, $p^*(\tau_i) = p(\tau_i)$). In Section 3.2.1, we show how state-of-the-art neural density estimation methods based on normalizing flows can be used to model $p(\tau_i)$. Then in Section 3.2.2, we propose a simple mixture model that can match the performance of the more sophisticated flow-based models, while also addressing some of their shortcomings. Finally, we discuss how to make $p(\tau_i)$ depend on the history $\mathcal{H}_{t_i}$ in Section 3.2.3.

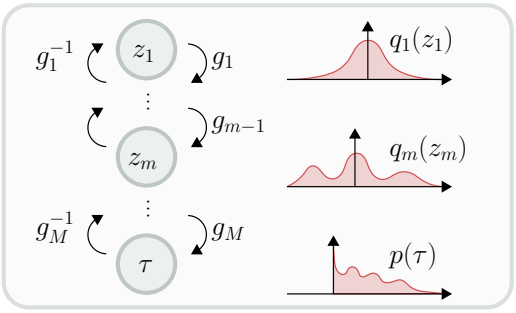


Figure 3.2: Normalizing flows define a flexible distribution via transformations.

3.2.1 Modeling $p(\tau)$ with normalizing flows

Recall from Section 2.2.2 that the core idea of normalizing flows is to define a flexible probability distribution by transforming a simple one. Assume that z has a PDF $q(z)$. Let $x = g(z)$ for some differentiable invertible transformation $g : \mathcal{Z} \to \mathcal{X}$ (where $\mathcal{Z}, \mathcal{X} \subseteq \mathbb{R}$). We consider only the one-dimensional case since our goal is to model the distribution of inter-event times $\tau \in \mathbb{R}_+$.

We can obtain the PDF $p(x)$ of x using the change of variables formula as defined in Equation 2.32. By stacking multiple transformations $g_1, \ldots, g_M$, we obtain an expressive probability distribution $p(x)$. To draw a sample $x \sim p(x)$, we need to draw $z \sim q(z)$ and compute the *forward* transformation $x = (g_M \circ \cdots \circ g_1)(z)$. To get the density of an arbitrary point x, it is necessary to evaluate the *inverse* transformation $z = (g_1^{-1} \circ \cdots \circ g_M^{-1})(x)$ and compute $q(z)$.

Modern normalizing flows architectures parameterize the transformations using extremely flexible functions $f_{\boldsymbol{\theta}}$, such as polynomials [116] or neural networks [106]. In this work, we don't consider invertible normalizing flows based on dimension splitting, such as RealNVP [59], since they are not applicable to one-dimensional data.

In the context of TPPs, our goal is to model the distribution $p(\tau)$ of inter-event times. In order to be able to learn the parameters of $p(\tau)$ using maximum likelihood, we need to be able to evaluate the density at any point τ. For this we need to define the inverse transformation $g^{-1} := (g_1^{-1} \circ \cdots \circ g_M^{-1})$. First, we set $z_M = g_M^{-1}(\tau) = \log \tau$ to convert a positive $\tau \in \mathbb{R}_+$ into $z_M \in \mathbb{R}$. Then, we stack multiple layers of parametric functions $f_{\boldsymbol{\theta}} : \mathbb{R} \to \mathbb{R}$ that can approximate any transformation. We consider two choices for $f_{\boldsymbol{\theta}}$:

- Deep sigmoidal flow (DSF) from Huang et al. [106]:

$$f^{DSF}(x) = \sigma^{-1}\left(\sum_{k=1}^{K} w_k \sigma\left(\frac{x - \mu_k}{s_k}\right)\right), \tag{3.5}$$

- Sum-of-squares (SOS) polynomial flow from Jaini et al. [116]:

$$f^{SOS}(x) = a_0 + \sum_{k=1}^{K} \sum_{p=0}^{R} \sum_{q=0}^{R} \frac{a_{p,k} a_{q,k}}{p+q+1} x^{p+q+1}, \tag{3.6}$$

where $\boldsymbol{a}, \boldsymbol{w}, \boldsymbol{s}, \boldsymbol{\mu}$ are the transformation parameters, K is the number of components, R is the polynomial degree. We denote the two variants of the model based on f^{DSF} and f^{SOS} building blocks as **DSFlow** and **SOSFlow**, respectively. Finally, after stacking multiple $g_m^{-1} = f_{\boldsymbol{\theta}_m}$, we apply a sigmoid transformation $g_1^{-1} = \sigma$ to convert z_2 into $z_1 \in (0, 1)$.

For both models, we can evaluate the inverse transformations $(g_1^{-1} \circ \cdots \circ g_M^{-1})$, which means the model can be efficiently trained via maximum likelihood. The density $p(\tau)$ defined by either DSFlow or SOSFlow model is extremely flexible and can approximate any distribution (Section 3.2.4). However, for some use cases, this is not sufficient. For example, we may be interested in the expected time until the next event, $\mathbb{E}_p[\tau]$. In this case, flow-based models are not optimal, since for them $\mathbb{E}_p[\tau]$ does not in general have a closed-form. Moreover, the forward transformation $(g_M \circ \cdots \circ g_1)$ cannot be computed in closed-form since the functions f^{DSF} and f^{SOS} cannot be inverted analytically. Therefore, sampling from $p(\tau)$ is also problematic and requires iterative root finding.

This raises the question: Can we design a model for $p(\tau)$ that is as expressive as the flow-based models, but in which sampling and computing moments is easy and can be done in closed-form?

3.2.2 Modeling $p(\tau)$ with mixture distributions

Model definition. While mixture models are commonly used for clustering, they can also be used for density estimation. Mixtures work especially well in low dimensions [176], which is the case in TPPs, where we model the distribution of one-dimensional inter-event times τ. Since the inter-event times τ are positive, we choose to use a mixture of log-normal distributions to model $p(\tau)$. The PDF of a log-normal mixture is defined as:

$$p(\tau | \boldsymbol{w}, \boldsymbol{\mu}, \boldsymbol{s}) = \sum_{k=1}^{K} w_k \frac{1}{\tau s_k \sqrt{2\pi}} \exp\left(-\frac{(\log \tau - \mu_k)^2}{2 s_k^2} \right), \tag{3.7}$$

where $\boldsymbol{w}$ are the mixture weights, $\boldsymbol{\mu}$ are the mixture means, and $\boldsymbol{s}$ are the standard deviations. Because of its simplicity, the log-normal mixture model has a number of attractive properties.

Moments. Since each component k has a finite mean, the mean of the entire distribution can be computed as a weighted average of component means:

$$\mathbb{E}_p[\tau] = \sum_k w_k \exp(\mu_k + s_k^2/2). \tag{3.8}$$

Higher moments can be computed based on the moments of each component [74].

Sampling. While flow-based models from Section 3.2.1 require iterative root-finding algorithms to generate samples, sampling from a mixture model can be done in closed-form:

1. $\boldsymbol{z} \sim \text{Categorical}(\boldsymbol{w})$, where $\boldsymbol{z}$ is a one-hot vector of size K,
2. $\varepsilon \sim \text{Normal}(0, 1)$,
3. $\tau = \exp(\boldsymbol{s}^T \boldsymbol{z} \cdot \varepsilon + \boldsymbol{\mu}^T \boldsymbol{z})$.

In some applications, such as reinforcement learning [271], we might be interested in computing gradients of the samples with respect to the model parameters. The samples τ drawn using the procedure above are differentiable with respect to the means $\boldsymbol{\mu}$ and scales $\boldsymbol{s}$. By using the Gumbel-softmax trick [117] when sampling $\boldsymbol{z}$, we can obtain gradients w.r.t. all the model parameters (Appendix A.4.6). Such reparameterization gradients have lower variance and are easier to implement than the score function estimators typically used in other works [182]. Other flexible models (such as multi-layer flow models from Section 3.2.1) do not permit sampling through reparameterization, and thus are not well-suited for the above-mentioned scenario. In Section 3.3.4, we show how reparameterization sampling can also be used to train with missing data by performing imputation on the fly.

3.2.3 Incorporating the conditional information

History. A crucial feature of temporal point processes is that the time $\tau_i = (t_i - t_{i-1})$ until the next event may be influenced by all the events that happened before. A standard way of capturing this dependency is to process the event history $\mathcal{H}_{t_i}$ with a recurrent neural network (RNN) and embed it into a fixed-dimensional vector $\boldsymbol{h}_i \in \mathbb{R}^H$ [61]. A detailed description of different RNN architectures can be found in Section 2.1.2.

Conditioning on additional features. The distribution of the time until the next event might depend on factors other than the history. For instance, distribution of arrival times of customers in a restaurant depends on the day of the week. As another example, if we are modeling user behavior in an online system, we can obtain a different distribution $p^*(\tau)$ for each user by conditioning on their metadata. We denote such side information as a vector $\boldsymbol{y}_i$. Such information is different from marks, since (i) the metadata may be shared for the entire sequence, and (ii) $\boldsymbol{y}_i$ only influences the distribution $p^*(\tau_i|\boldsymbol{y}_i)$, not the objective function. Such additional information is also known as covariates in the time series literature.

In some scenarios, we might be interested in learning from multiple event sequences. In such case, we can assign each sequence $\mathcal{T}_j$ a learnable **sequence embedding** vector $\boldsymbol{e}_j$. By optimizing $\boldsymbol{e}_j$, the model can learn to distinguish between sequences that come from different distributions. The learned embeddings can then be used for visualization, clustering or other downstream tasks.

Obtaining the parameters. We model the conditional dependence of the distribution $p^*(\tau_i)$ on all of the above factors in the following way. The history embedding $\boldsymbol{h}_i$, metadata

$\boldsymbol{y}_i$ and sequence embedding $\boldsymbol{e}_j$ are concatenated into a context vector $\boldsymbol{c}_i = [\boldsymbol{h}_i||\boldsymbol{y}_i||\boldsymbol{e}_j]$. Then, we obtain the parameters of the distribution $p^*(\tau_i)$ as an affine function of $\boldsymbol{c}_i$. For example, for the mixture model we have:

$$\boldsymbol{w}_i = \text{softmax}(\boldsymbol{V}_{\boldsymbol{w}}\boldsymbol{c}_i + \boldsymbol{b}_{\boldsymbol{w}}) \tag{3.9}$$

$$\boldsymbol{s}_i = \exp(\boldsymbol{V}_{\boldsymbol{s}}\boldsymbol{c}_i + \boldsymbol{b}_{\boldsymbol{s}}) \tag{3.10}$$

$$\boldsymbol{\mu}_i = \boldsymbol{V}_{\boldsymbol{\mu}}\boldsymbol{c}_i + \boldsymbol{b}_{\boldsymbol{\mu}} \tag{3.11}$$

where the softmax and exp transformations are applied to enforce the constraints on the distribution parameters, and $\boldsymbol{\theta} = \{\boldsymbol{V}_{\boldsymbol{w}}, \boldsymbol{V}_{\boldsymbol{s}}, \boldsymbol{V}_{\boldsymbol{\mu}}, \boldsymbol{b}_{\boldsymbol{w}}, \boldsymbol{b}_{\boldsymbol{s}}, \boldsymbol{b}_{\boldsymbol{\mu}}\}$ are learnable parameters. Such model resembles the mixture density network architecture [21]. The whole process is illustrated in Figure 3.1. We obtain the parameters of the flow-based models in a similar way (see Appendix A.4).

3.2.4 Universal approximation

The SOSFlow and DSFlow models can approximate any probability density on $\mathbb{R}$ arbitrarily well [116, Theorem 3], [106, Theorem 4]. It turns out, a mixture model has the same universal approximation property.

Theorem 1. *[50, Theorem 33.2] Let $p(x)$ be a continuous density on $\mathbb{R}$. If $q(x)$ is any density on $\mathbb{R}$ and is also continuous, then, given $\varepsilon > 0$ and a compact set $\mathcal{S} \subset \mathbb{R}$, there exist number of components $K \in \mathbb{N}$, mixture coefficients $\boldsymbol{w} \in \Delta^{K-1}$, locations $\boldsymbol{\mu} \in \mathbb{R}^K$, and scales $\boldsymbol{s} \in \mathbb{R}_+^K$ such that for the mixture distribution $\hat{p}(x) = \sum_{k=1}^K w_k \frac{1}{s_k} q(\frac{x-\mu_k}{s_k})$ it holds $\sup_{x\in\mathcal{S}} |p(x) - \hat{p}(x)| < \varepsilon$.*

This result shows that, in principle, the mixture distribution is as expressive as the flow-based models. Since we are modeling the conditional density, we additionally need to assume for all of the above models that the RNN can encode all the relevant information into the history embedding $\boldsymbol{h}_i$. This can be accomplished by invoking the universal approximation theorems for RNNs [244, 235].

Note that this result, like other universal approximation theorems of this kind [46, 49], does not provide any practical guarantees on the obtained approximation quality, and doesn't say how to learn the model parameters. Still, universal approximation intuitively seems like a desirable property of a distribution. This intuition is supported by experimental results. In Section 3.3.1, we show that models with the universal approximation property consistently outperform the less flexible ones.

Interestingly, Theorem 1 does not make any assumptions about the form of the base density $q(x)$. This means we could as well use a mixture of distribution other than log-normal. However, other popular distributions on $\mathbb{R}_+$ have drawbacks: log-logistic does not always have defined moments and gamma distribution doesn't permit straightforward sampling with reparameterization.

3.2.5 Intensity function

For both flow-based and mixture models, the conditional cumulative distribution function (CDF) $F^*(\tau)$ and the PDF $p^*(\tau)$ are readily available. This means we can easily compute

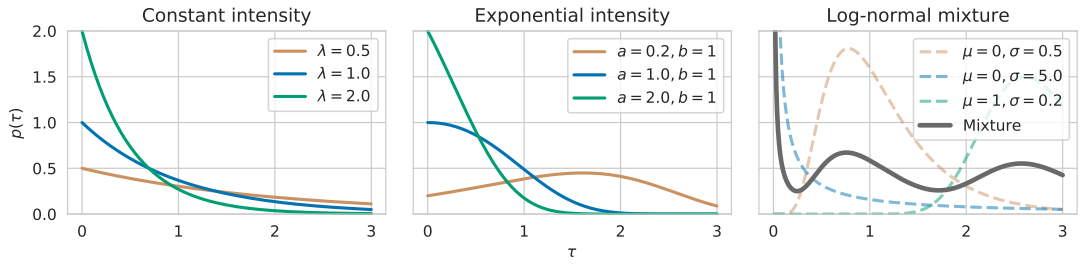


Figure 3.3: Different choices for modeling $p(\tau)$: exponential distribution (left), Gompertz distribution (center), log-normal mixture (right). Mixture distribution can approximate any density while being tractable and easy to sample from.

the respective intensity functions (see Appendix A.1). However, we should still ask whether we lose anything by modeling $p^*(\tau)$ instead of $\lambda^*(t)$. The main arguments in favor of modeling the intensity function in traditional models (e.g. self-exciting process) are that it's *intuitive*, *easy to specify* and *reusable* [272].

- "Intensity function is *intuitive*, while the conditional density is not."—While it is true that in simple models (e.g. in self-exciting or self-correcting processes) the dependence of $\lambda^*(t)$ on the history is intuitive and interpretable, modern RNN-based intensity functions (as in Du et al. [61], Mei and Eisner [180], and Omi et al. [199]) cannot be easily understood by humans. In this sense, our proposed models are as intuitive and interpretable as other existing intensity-based neural network models.

- "$\lambda^*(t)$ is *easy to specify*, since it only has to be positive. On the other hand, $p^*(\tau)$ must integrate to one."—As we saw, by using either normalizing flows or a mixture distribution, we automatically enforce that the PDF integrates to one, without sacrificing the flexibility of our model.

- "*Reusability:* If we merge two independent point processes with intensities $\lambda_1^*(t)$ and $\lambda_2^*(t)$, the merged process has intensity $\lambda^*(t) = \lambda_1^*(t) + \lambda_2^*(t)$."—An equivalent result exists for the CDFs $F_1^*(\tau)$ and $F_2^*(\tau)$ of the two independent processes. The CDF of the merged process is obtained as $F^*(\tau) = F_1^*(\tau) + F_2^*(\tau) - F_1^*(\tau)F_2^*(\tau)$ (derivation is provided in Appendix A.1).

As we just showed, modeling $p^*(\tau)$ instead of $\lambda^*(t)$ does not impose any limitations on our approach. Moreover, a mixture distribution is flexible, easy to sample from and has well-defined moments, which favorably compares it to other intensity-based deep learning models. Figure 3.3 shows densities that can be represented by some of our competitors: exponential and Gompertz distributions [61] (see Appendix A.2 for more details). Even though the history embedding $\boldsymbol{h}_i$ produced by an RNN may capture rich information, the resulting distribution $p^*(\tau_i)$ for both models has very limited flexibility, is unimodal and light-tailed. In contrast, a flow-based or a mixture model is significantly more flexible and can approximate any density.

3.2.6 Other related work

Neural temporal point processes. Fitting simple TPP models, such as self-exciting [99] or self-correcting [112] processes, to real-world data may lead to poor results because of model misspecification. Multiple recent works address this issue by proposing more flexible neural-network-based point process models. These neural models are usually defined in terms of the conditional intensity function. For example, Mei and Eisner [180] propose a novel RNN architecture that can model sophisticated intensity functions. This flexibility comes at the cost of inability to evaluate the likelihood in closed-form, and thus requiring Monte Carlo integration. We already discussed limitations of some other works in Section 3.1 and in Appendix.

Several works used mixtures of kernels to parameterize the conditional intensity function [261, 260, 196]. Such models can only capture self-exciting influence from past events. Moreover, these models do not permit computing expectation and drawing samples in closed-form. Türkmen et al. [268] propose neural models for learning marked TPPs. We pursue a similar goal in Chapter 4, with the emphasis on assigning uncertainty estimate to the mark prediction. Other recent works consider alternatives to the maximum likelihood objective for training TPPs. Examples include noise-contrastive estimation [91], Wasserstein distance [284, 285, 286], and reinforcement learning [155, 271]. This line of research is orthogonal to our contribution, and the models proposed in our work can be combined with the above-mentioned training procedures.

Neural density estimation. There exist two popular paradigms for learning flexible probability distributions using neural networks: in mixture density networks [21], a neural net directly produces the distribution parameters; in normalizing flows [259, 224], we obtain a complex distribution by transforming a simple one. Both mixture models [238, 66, 90] and normalizing flows [200, 303] have been applied for modeling sequential data. However, surprisingly, none of the existing works make the connection and consider these approaches in the context of TPPs. Other approaches, such as variational inference (Section 2.2.3), generative diffusion (Section 2.2.4) or GANs [85], do not offer exact likelihood computation.

3.3 Experiments

We evaluate the proposed models on the established task of event time prediction (with and without marks) in Sections 3.3.1 and 3.3.2. In the remaining experiments, we show how the log-normal mixture model can be used for incorporating extra conditional information, training with missing data and learning sequence embeddings. We use 6 real-world datasets containing event data from various domains: Wikipedia (article edits), MOOC (user interaction with an online course), Reddit (posts in social media) [143], Stack Overflow (badges received by users), LastFM (music playback) [61], and Yelp (check-ins to restaurants). We also generate 5 synthetic datasets (Poisson, Renewal, Self-correcting, Hawkes1, Hawkes2), as described in Omi et al. [199]. Detailed descriptions and summary statistics of all the datasets are provided in Appendix A.5.

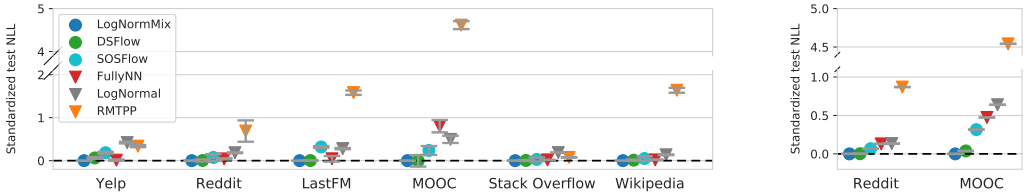


Figure 3.4: NLL loss for event time prediction without marks (Left) and with marks (Right). NLL of each model is standardized by subtracting the score of LogNormMix. **Lower values are better.** Despite its simplicity, LogNormMix consistently achieves excellent loss values.

3.3.1 Event time prediction using history

Setup. We consider two normalizing flow models, SOSFlow and DSFlow (Equations 3.6 and 3.5), as well a log-normal mixture model (Equation 3.7), denoted as LogNormMix. As baselines, we consider RMTPP (that is, Gompertz and exponential from Du et al. [61]) and FullyNN model by Omi et al. [199]. Additionally, we use a single log-normal distribution (denoted LogNormal) to highlight the benefits of the mixture model. For all models, an RNN encodes the history into a vector $\boldsymbol{h}_i$. The parameters of $p^*(\tau)$ are then obtained using $\boldsymbol{h}_i$ (Equation 3.11). We exclude the NeuralHawkes model from our comparison, since it is known to be inferior to RMTPP in time prediction [180], and, unlike other models, doesn't have a closed-form likelihood.

Each dataset consists of multiple sequences of event times. The task is to predict the time τ_i until the next event given the history $\mathcal{H}_{t_i}$. For each dataset, we use 60% of the sequences for training, 20% for validation and 20% for testing. We train all models by minimizing the negative log-likelihood (NLL) of the inter-event times in the training set. To ensure a fair comparison, we try multiple hyperparameter configurations for each model and select the best configuration using the validation set. Finally, we report the NLL loss of each model on the test set. All results are averaged over 10 train/validation/test splits. Details about the implementation, training process and hyperparameter ranges are provided in Appendix A.4. For each real-world dataset, we report the difference between the NLL loss of each method and the LogNormMix model (Figure 3.4). We report the *differences*, since scores of all models can be shifted arbitrarily by scaling the data. Absolute scores (not differences) in a tabular format, as well as results for synthetic datasets are provided in Appendix A.6.1.

Results. Simple unimodal distributions (Gompertz, RMTPP, LogNormal) are always dominated by the more flexible models with the universal approximation property (LogNormMix, DSFlow, SOSFlow, FullyNN). Among the simple models, LogNormal provides a much better fit to the data than RMTPP. The distribution of inter-event times in real-world data often has heavy tails, and the Gompertz distributions fails to capture this behavior. We observe that the two proposed models, LogNormMix and DSFlow consistently achieve the best loss values.

3.3.2 Learning with marks

Setup. We apply the models for learning in *marked* temporal point processes. Marks are known to improve performance of simpler models [61], we want to establish whether our proposed models work well in this setting. We use the same setup as in the previous section, except for two differences. The RNN takes a tuple (τ_i, m_i) as input at each time step, where m_i is the mark. Moreover, the loss function now includes a term for predicting the next mark (implementation details in Appendix A.6.2):

$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_i \left[\log p^*_{\boldsymbol{\theta}}(\tau_i) + \log p^*_{\boldsymbol{\theta}}(m_i)\right]. \tag{3.12}$$

Results. Figure 3.4 (Right) shows the time NLL loss (that is, $-\sum_i \log p^*(\tau_i)$) for Reddit and MOOC datasets. LogNormMix shows dominant performance in the marked case, just like in the previous experiment. Like before, we provide the results in tabular format, as well as report the marks NLL loss in Appendix A.6.

3.3.3 Learning with additional conditional information

Setup. We investigate whether the additional conditional information (Section 3.2.3) can improve performance of the model. In the Yelp dataset, the task is predict the time τ until the next check-in for a given restaurant. We postulate that the distribution $p^*(\tau)$ is different, depending on whether it's a weekday and whether it's an evening hour, and encode this information as a vector $\boldsymbol{y}_i$. We consider 4 variants of the LogNormMix model, that either use or don't use $\boldsymbol{y}_i$ and the history embedding $\boldsymbol{h}_i$.

Results. Figure 3.6 shows the test set loss for 4 variants of the model. We see that additional conditional information boosts performance of the LogNormMix model, regardless of whether the history embedding is used.

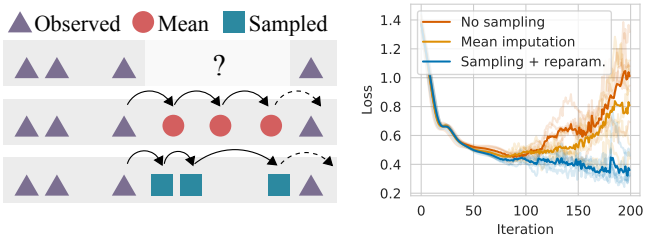
3.3.4 Missing data imputation

In practical scenarios, one often has to deal with missing data. For example, we may know that records were not kept for a period of time, or that the data is unusable for some reason. Since TPPs are a generative model, they provide a principled way to handle the missing data through imputation.

Setup. We are given several sequences generated by a Hawkes process, where some parts are known to be missing. We consider three different strategies for learning from such a partially observed sequence:

(a) ignore the gaps, maximize log-likelihood of observed inter-event times,

(b) fill the gaps with the average τ estimated from observed data, maximize log-likelihood of observed data,



Model	NLL
No imputation	1.01 ± 0.20
Mean imputation	0.81 ± 0.27
Sampling with reparametrization	0.36 ± 0.05

Figure 3.5: (Left) Illustration of the experiment with missing data imputation, (Middle) training curves for different models, (Right) resulting negative log-likelihood on the imputed data. By sampling the missing values from $p^*(\tau)$ during training, LogNormMix learns the true underlying data distribution. Other imputation strategies overfit the partially observed sequence.

(c) fill the gaps with samples generated by the model, maximize the *expected* log-likelihood of the observed points.

The setup is demonstrated in Figure 3.5. Note that in case (c) the expected value depends on the parameters of the distribution, hence we need to perform sampling with reparameterization to optimize such loss. A more detailed description of the setup is given in Appendix A.6.4.

Results. The 3 model variants are trained on the partially-observed sequence. Figure 3.5 shows the NLL of the fully observed sequence (not seen by any model at training time) produced by each strategy. We see that strategies (a) and (b) overfit the partially observed sequence. In contrast, strategy (c) generalizes and learns the true underlying distribution. The ability of the LogNormMix model to draw samples with reparameterization was crucial to enable such training procedure.

3.3.5 Sequence embedding

Different sequences in the dataset might be generated by different processes, and exhibit different distribution of inter-event times. We can "help" the model distinguish between them by assigning a trainable embedding vector $\boldsymbol{e}_j$ to each sequence j in the dataset. It seems intuitive that embedding vectors learned this way should capture some notion of similarity between sequences.

Learned sequence embeddings. We learn a sequence embedding for each of the sequences in the synthetic datasets (along with other model parameters). We visualize the learned embeddings using t-SNE [169] in Figure 3.8 colored by the true class. As we see, the model learns to differentiate between sequences from different distributions in a completely unsupervised way.

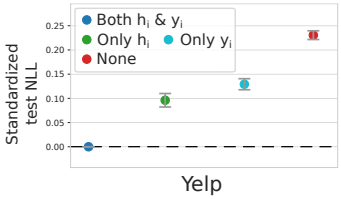


Figure 3.6: Conditional info improves the performance, as expected.

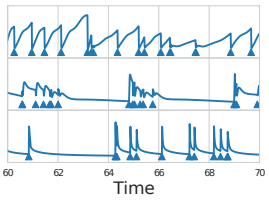


Figure 3.7: New sequences generated based on different embeddings.

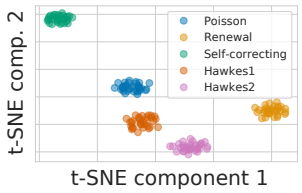


Figure 3.8: Sequence embeddings learned by the model.

Generation. We fit the LogNormMix model to two sequences generated from self-correcting and renewal processes, and, respectively, learn two embedding vectors $\boldsymbol{e}_{SC}$ and $\boldsymbol{e}_{RN}$ for them. After training, we generate three sequences from our model, using $\boldsymbol{e}_{SC}$, $\frac{1}{2}(\boldsymbol{e}_{SC} + \boldsymbol{e}_{RN})$ and $\boldsymbol{e}_{RN}$ as sequence embeddings. That is, the first sampled sequence should correspond to self-correcting process, the third to renewal process and the second to something in between. Additionally, we plot the *learned* conditional intensity function of our model for each generated sequence in Figure 3.7. We can see that the model learns to map the sequence embeddings to very different distributions.

3.4 Discussion

We used tools from neural density estimation to design new models for learning temporal point processes. We show that a simple mixture model is competitive with state-of-the-art normalizing flows methods, as well as convincingly outperforms other existing approaches. By looking at "learning in TPPs" from a different perspective, we were able to address the shortcomings of existing intensity-based approaches, such as insufficient flexibility, lack of closed-form likelihoods and inability to generate samples analytically.

The success of our approach can be found in imposing certain constraints on the neural network such that we get *nicer* behavior during both training and evaluation. By replacing the intensity function with the probability density we create a familiar setting for the maximum likelihood training. Although intensity parameterization is equivalent, the lack of numerical integration and the greater flexibility ensures we achieve better results at the lower training and evaluation costs.

3.4.1 Spatial point processes

So far we have seen how to model positions of points recorded on a real line. Let us briefly turn our attention to the problem of modeling point processes beyond the line, in particular, we are interested in spatial point processes whose realizations are sets of points lying on $\mathbb{R}^d$. The goal is to find a model that offers straightforward sampling and likelihood evaluation, similar to what we have for TPPs.

However, unlike the TPPs, data points now lack ordering so we cannot directly use an autoregressive model. We refer to such data as exchangeable, that is, any permutation of

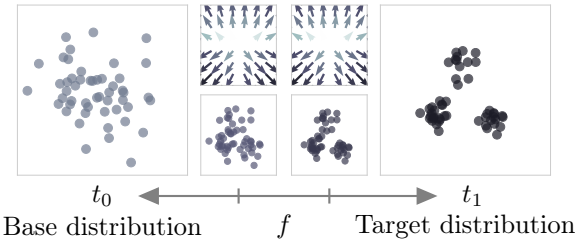


Figure 3.9: Illustration of the approach. Transforming random sets of points from the base distribution with an ordinary differential equation gives a realization in the target distribution. The opposite direction calculates the likelihood of the observed sample. The dynamics f has to be a permutation equivariant neural network.

random variables will have the same probability. Examples include point clouds, items in a shopping cart, tracking household electricity consumption in a city etc. Additionally, the number of points is also a random variable. The setup is explained in Section 2.2.3 and the probability of observing a set of points $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ can be written as:

$$p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) = n! p(n) \tilde{p}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n), \tag{2.30}$$

where $\tilde{p}$ is the symmetric density—one that is invariant to point permutations. We want to model $\tilde{p}$ without taking the i.i.d. assumption meaning the points should be able to interact with each other in order to capture clustering behavior, as an example. Previous works either impose the ordering on the points [11, 188, 255, 292] or model the conditionally independent per-point distribution with a variational autoencoder [288, 293].

We [15] propose an alternative solution by using normalizing flows to model the density of sets. The general approach is illustrated in Figure 3.9. The constraint on a network is that it has to take in arbitrary number of points and model permutation invariant densities. It can be shown that the resulting normalizing flow transformation has to be permutation equivariant for this to hold. In particular, we use a continuous normalizing flow since it can support arbitrary transformations, including the special case we actually need. On its own, this would result in a computationally heavy model since the computation of the trace of the Jacobian is expensive (see Equation 2.38). Therefore, we address the issue by decoupling the computation into different parts such that the trace is computed quickly in closed-form without losing the expressivity [15].

The final model has better performance, both in terms of training time and accuracy. The takeaway message is that marrying neural density estimation with point processes results in powerful models with very convenient properties: exact likelihood and straightforward sampling. Additional constraints imposed on a model (here: invariance) guarantee that it is operating withing prescribed bounds and often improve the final performance.

4 Uncertainty on Event Prediction

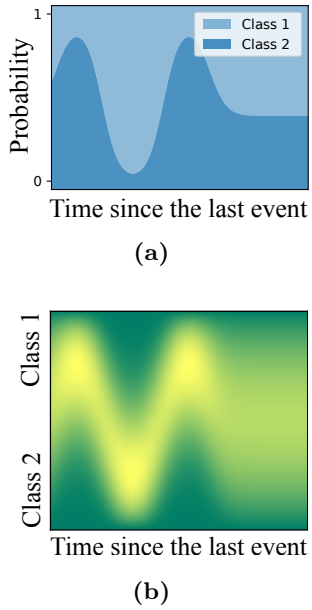


Figure 4.1: Two different *levels* of predicting with uncertainty. (a) An event can be expected multiple times in the future and the uncertainty is written down as a percentage. For example, we can believe there is 70% change that Class 2 will occur. (b) What we actually want is assign the amount of certainty in this percentage. Here, yellow shows high certainty areas where we are confident in the prediction and green denotes high uncertainty, both evolving with time. For example, we can be certain that "there is 70% probability Class 2 will occur" but uncertain if *anything* will happen in the future. That means we can reason beyond *either this or that* to include *neither* and *how sure*.

The problem we are solving in this chapter is: "given a (past) sequence of irregularly-sampled[1] events, what will happen next?" Answering this question enables us to predict real-world outcomes, for example, which action will an online user perform next or which car part will become anomalous so we can schedule the maintenance.

While many recurrent models for irregularly-sampled sequences have been proposed in the past [190, 61], they are ill-suited for this task since they output a *single prediction* only, for example, the most likely next event. In an irregular setting, however, such a single prediction is not enough since the most likely event can change with the passage of time—even if no other events happen. Consider a car approaching another vehicle in front of it. Assuming nothing happens in the meantime, we can expect different events at *different times in the future*. When forecasting on a short time window, one expects the driver to start overtaking; after a bit more time one would expect braking; in the long term, one would expect a collision! Thus, the expected behavior changes depending on the time we forecast, assuming no events occurred in the meantime.

Figure 4.2a illustrates this schematically: having observed events denoted with a square and a pentagon, it is likely to observe a square after a short amount of time and a circle

[1] In the original publication [14], we called these events asynchronous; it was even in the title. Here, however, we switch to irregular/irregularly-sampled to have uniform notation across all chapters.

after more time has passed. Clearly, if some event occurs (for example, 'braking' or 'square'), the event at the new observed time will be taken into account, updating the temporal prediction. An ad-hoc solution to this problem would be to discretize time. However, if the events are near each other, we need a high sampling frequency, giving us high computational cost. Besides, since there can be intervals without events, an artificial 'no event' class is required.

In this work, we solve these problems by directly predicting the entire evolution of the events over (continuous) time. Given a past irregularly-sampled sequence as input, we can predict and evaluate for *any* future time point what the event is likely to be (under the assumption that no other event happens in between which would lead to an update of our model). Crucially, the likelihood of the events might change and one event can be more likely than others over multiple time points in the future. This periodicity exists in many event sequences. For instance, given that a person is currently at home, a smart home would predict a high probability that the kitchen will be used at lunch and/or dinner time (see Figure 4.1a for an illustration of such periodicity). We require that our model captures this kind of multimodality.

While Figure 4.1a illustrates the evolution of the categorical distribution (corresponding to the probability of a specific event class to happen), the issue still arises outside of the observed data distribution. For example, in some time intervals we can be *certain* that two classes are *equiprobable*, having observed many similar examples. However, if the model has not seen any examples at specific time intervals during training, we do not want to give a confident prediction.

Therefore, we incorporate *uncertainty* in a prediction directly in our model. In places where we expect events, the confidence will be higher, and outside of these areas the uncertainty in a prediction will grow as illustrated in Figure 4.1b. Technically, instead of modeling the evolution of a categorical distribution, we model the *evolution of a distribution on the probability simplex*. Overall, our model enables us to operate with the *irregular discrete* event data from the past as input to perform *continuous-time* predictions to the future incorporating the predictions' uncertainty. This is in contrast to the temporal point process works as described in Chapter 3.

4.1 Method

We consider a sequence of events $[e_1, \dots, e_n]$, where each event $e_i = (c_i, t_i)$ is a tuple of an event class $c_i \in \{1, \dots, C\}$ and $t_i \in \mathbb{R}$ is its time of occurrence. This is similar to notation of marked TPPs from Chapter 3. Therefore, we assume the events arrive over time with $t_i > t_{i-1}$, and we introduce $\tau_i^* = t_i - t_{i-1}$ as the observed time gap between the ith and the $(i-1)$th event. The history preceding the ith event is denoted again by $\mathcal{H}_i$.

Let $S = \{\boldsymbol{p} \in [0,1]^C, \sum_c p_c = 1\}$ denote the set of probability vectors that form the $(C-1)$-dimensional simplex, and $P(\theta)$ be a family of probability distributions on this simplex parameterized by parameters θ. Every sample $\boldsymbol{p} \sim P(\theta)$ corresponds to a categorical class distribution. Such a simplex is illustrated on the right-hand side of Figure 4.2b. Each point inside the two-dimensional simplex (triangle) corresponds to the

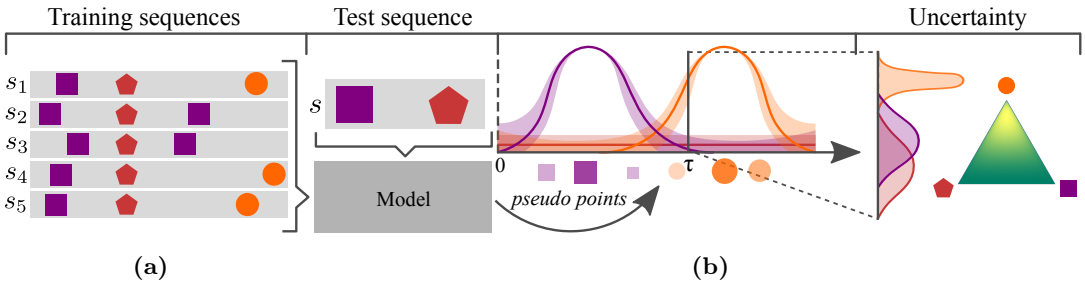


Figure 4.2: The model framework. (a) During training we use sequences s_i. (b) Given a new sequence of events s the model generates pseudo points that describe $\boldsymbol{\theta}(\tau)$, i.e. the temporal evolution of the distribution on the simplex. These pseudo points are based on the data that was observed in the training examples and weighted accordingly. We also have a measure of certainty in our prediction.

probability of choosing each of the three classes. The main idea in this work is to put the probability over the simplex.

Given e_{i-1} and $\mathcal{H}_{i-1}$, our goal is to model the evolution of the class probabilities of the next event i over time, together with their uncertainty. Technically, we model parameters $\theta(\tau)$, leading to a distribution P over the class probabilities $\boldsymbol{p}$ for all $\tau \geq 0$. Thus, we can estimate the most likely class after a time gap τ by calculating:

$$c^\star = \operatorname*{argmax}_c \mathbb{E}_{\boldsymbol{p}(\tau)\sim P(\theta(\tau))}[\boldsymbol{p}(\tau)]_c,$$

that is, we compute the expected probability vector at τ and then simply take the class with the largest value in the expected vector. Alternatively, we can view this as finding the class that carries the most probability mass over the simplex.

Further, since we do not consider only a point estimate, we can obtain the amount of certainty in a prediction. For this, we estimate the probability of class c being more likely than the other classes, given by:

$$q_c(\tau) := \mathbb{E}_{\boldsymbol{p}(\tau)\sim P(\theta(\tau))}[\mathbf{1}_{\boldsymbol{p}(\tau)_c \geq \max_{c' \neq c} \boldsymbol{p}(\tau)_{c'}}].$$

This tells us how certain we are that one class is the most probable, that is, how often is c the argmax when sampling from P.

We have to tackle two core challenges:

1. **Expressiveness.** Since the time dependence of $\theta(\tau)$ may be of different forms, we need to capture complex behavior.

2. **Locality.** For regions out of the observed data we want to have a higher uncertainty in our predictions. Specifically for $\tau \to \infty$, that is, far into the future, the distribution should have a high uncertainty.

Two expressive and well-established choices for the family P are the Dirichlet distribution and the logistic-normal distribution.

Dirichlet distribution with concentration parameters $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_K)$, where $\alpha_i > 0$, has the probability density function:

$$f(\boldsymbol{x}; \boldsymbol{\alpha}) = \frac{\prod_{i=1}^{K} \Gamma(\alpha_i)}{\Gamma\left(\sum_{i=1}^{K} \alpha_i\right)} \prod_{i=1}^{K} x_i^{\alpha_i - 1}, \tag{4.1}$$

where Γ is a Gamma function: $\Gamma(\alpha) = \int_0^\infty \alpha^{z-1} e^{-\alpha} \, \mathrm{d}z$.

Logistic-normal distribution (LN) is a generalization of the logit-normal distribution for the multivariate case. If $\boldsymbol{y} \in \mathbb{R}^C$ follows $\boldsymbol{y} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ then

$$\boldsymbol{x} = \left[\frac{e^{y_1}}{\sum_{i=1}^{C} e^{y_i}}, \ldots, \frac{e^{y_C}}{\sum_{i=1}^{C} e^{y_i}} \right]$$

follows a logistic-normal distribution.

Based on a common modeling idea, in the following we present two models that exploit the specificities of these distributions: the WGP-LN (Section 4.1.1) and the FD-Dir (Section 4.1.2). We also introduce a novel loss to train these models in Section 4.1.3.

4.1.1 Logistic-normal via weighted Gaussian process (WGP-LN)

We start by describing our model for the case when P is the family of logistic-normal (LN) distributions. How to model a compact yet expressive evolution of the LN distribution? Our core idea is to exploit the fact that the LN distribution corresponds to a multivariate random variable whose *logits* follow a *normal distribution*. From here, a natural way to model the evolution of a normal distribution is a *Gaussian Process*. Given this insight, the core idea of our model is illustrated in Figure 4.2:

1. We generate M pseudo points from the hidden state of the RNN,
2. We fit a Gaussian Process to the pseudo points, capturing the temporal evolution,
3. We use the fitted GP for obtaining the parameters $\boldsymbol{\mu}(\tau)$ and $\boldsymbol{\Sigma}(\tau)$ of the final LN distribution at any specific time τ.

Therefore, by generating a small number of points we characterize the full distribution.

Classic GP. To keep the complexity low, we fit one GP per class c. That is, our model generates M points $(\tau_j^{(c)}, y_j^{(c)})$ per class c, where $y_j^{(c)}$ represents logits. Note that the first coordinate of each pseudo point corresponds to time, leading to the temporal evolution when fitting the GP. Essentially, we perform a non-parametric regression from the time domain to the logit space. Indeed, using a classic GP along with the pseudo points, the parameters θ of the logistic-normal distribution, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$, can be easily computed for any time τ in closed-form [219]:

$$\mu_c(\tau) = \boldsymbol{k}_c^T \boldsymbol{K}_c^{-1} \boldsymbol{y}_c, \;\; \sigma_c^2(\tau) = s_c - \boldsymbol{k}_c^T \boldsymbol{K}_c^{-1} \boldsymbol{k}_c \tag{4.2}$$

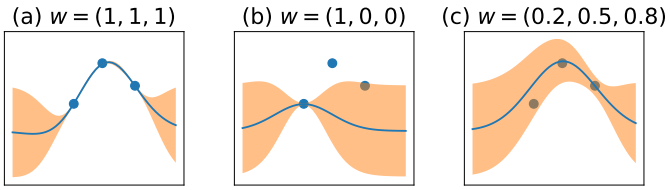


Figure 4.3: Weighted GP on toy data with different weight values. (a) When all the weights are one we recover a classic GP. (b) Wherever we have zero weights, we discard points. (c) Mixed weight assignment offers the tradeoff between the two extremes.

where $\boldsymbol{K}_c$ is the Gram matrix with respect to M pseudo points of class c based on a kernel k. For example, we can use a radial basis function:

$$k(\tau_1, \tau_2) = \exp(-\gamma^2(\tau_1 - \tau_2)^2). \tag{4.3}$$

Vector $\boldsymbol{k}_c$ contains at position j the value $k(\tau_j^{(c)}, \tau)$, and $\boldsymbol{y}_c$ the value $y_j^{(c)}$, and $s_c = k(\tau, \tau)$. At every time point τ the logits then follow a multivariate normal distribution with mean $\boldsymbol{\mu}(\tau)$ and covariance $\boldsymbol{\Sigma} = \text{diag}(\boldsymbol{\sigma}^2(\tau))$.

Using a GP enables us to describe complex functions. Furthermore, since GP models uncertainty in the prediction depending on the pseudo points, uncertainty is higher in areas far away from the pseudo points. Specifically, it holds for distant future; thus, matching the idea of locality. However, uncertainty is always low around the M pseudo points. Therefore, M should be carefully picked since there is a trade-off between having high certainty at (too) many time points and the ability to capture complex behavior. In the following we present an alternative version that solves this problem.

Weighted GP. We would like to pick M large enough to express rich multimodal functions but also allow the model to discard unnecessary points. To do this we generate an additional weight vector $\boldsymbol{w}^{(c)} \in [0, 1]^M$ that assigns the weight $w_j^{(c)}$ to a point at $\tau_j^{(c)}$. Setting the weight to zero at any point should discard it, and setting it to one will return the same result as with a classic GP. To achieve this goal, we propose a new kernel:

$$k'(\tau_1, \tau_2) = f(w_1, w_2)k(\tau_1, \tau_2) \tag{4.4}$$

where k is the same as above. The function f gives weights to the kernel k according to the weights for τ_1 and τ_2. We require f to have the following properties:

1. Function f should be a valid kernel over the weights, since then the function k' is a valid kernel as well,

2. The importance of pseudo points should not increase, giving $f(w_1, w_2) \leq \min(w_1, w_2)$; this fact implies that a point with zero weight will be discarded since $f(w_1, 0) = 0$ as desired.

The function $f(w_1, w_2) = \min(w_1, w_2)$ is a simple choice that fulfills these properties. In Figure 4.3 we show the effect of different weights when fitting a GP.

The behavior of the min kernel can intuitively explained by considering the Gram matrix $\boldsymbol{K}$ and vector $\boldsymbol{k}$, which are required to estimate μ and σ^2 for a new time point τ. Without loss of generality, consider M pseudo points $\tau_1, \ldots, \tau_M$ such that $w_1 < \cdots < w_M$. Since the new query point is observed we assign it weight 1. It follows:

$$\boldsymbol{k} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_M \end{bmatrix} \odot \begin{bmatrix} k(\tau_1, \tau) \\ k(\tau_2, \tau) \\ \vdots \\ k(\tau_M, \tau) \end{bmatrix}, \; \boldsymbol{K} = \begin{bmatrix} w_1 & w_1 & \ldots & w_1 \\ w_1 & w_2 & \ldots & w_2 \\ \vdots & \vdots & \ddots & \vdots \\ w_1 & w_2 & \ldots & w_M \end{bmatrix} \odot \begin{bmatrix} k(\tau_1, \tau_1) & \ldots & k(\tau_1, \tau_M) \\ k(\tau_2, \tau_1) & \ldots & k(\tau_2, \tau_M) \\ \vdots & \ddots & \vdots \\ k(\tau_M, \tau_1) & \ldots & k(\tau_M, \tau_M) \end{bmatrix} \tag{4.5}$$

Setting $w_1 = 0$ returns $\boldsymbol{k}$ without the first row and $\boldsymbol{K}$ without the first row and column. Plugging them back into Equation 4.2 we can see that the point τ_1 is discarded, as desired. In practice, the weights have values from interval $[0, 1]$ which in turn gives us the ability to *softly discard* points. This is shown in Figure 4.3 as we can see that the mean line does not have to cross through the points with weights < 1 and the variance can remain higher around them.

Full model. To predict μ and σ^2 for a new time τ, we can now simply apply Equation 4.2 based on the new kernel k', where the weight for the *query* point τ is 1.

To summarize: From a hidden state $h_i = \text{RNN}(e_{i-1}, \mathcal{H}_{i-1})$ we use a neural network to generate M weighted pseudo points $(w_j^{(c)}, \tau_j^{(c)}, x_j^{(c)})$ per class c. Fitting a Weighted GP to these points enables us to model the temporal evolution of $\mathcal{N}(\mu_c(\tau), \sigma_c^2(\tau))$ and, therefore, of the logistic-Normal distribution. Figure 4.4 shows an illustration of the model.

Note that the cubic complexity of a GP, due to the matrix inversion, is not an issue since the number M is usually small (< 10), while still allowing to represent rich multimodal functions. Crucially, given the loss defined in Section 4.1.3, our model is fully differentiable, enabling us efficient training.

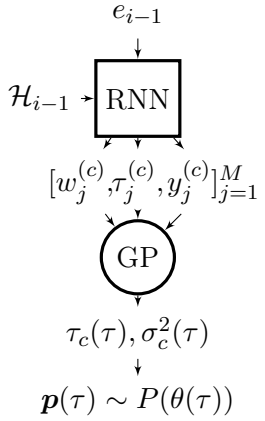


Figure 4.4: Model diagram

4.1.2 Dirichlet via function decomposition (FD-Dir)

Next, we consider the Dirichlet distribution to model the uncertainty in the predictions. The goal is to model the evolution of the concentrations parameters $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_C)^T$ of the Dirichlet over time. Since, unlike the logistic-normal case, we cannot draw the connection to the GP, we propose to decompose the parameters of the Dirichlet distribution with expressive (local) functions in order to allow complex dependence on time.

The concentration parameters $\alpha_c(\tau)$ need to be positive so we propose the following decomposition of $\alpha_c(\tau)$ in the log-space:

$$\log \alpha_c(\tau) = \sum_{j=1}^{M} w_j^{(c)} \cdot \mathcal{N}(\tau | \tau_j^{(c)}, \sigma_j^{(c)}) + \nu, \tag{4.6}$$

(a) $\tau = 0$ **(b)** $\tau = 0.5$ **(c)** $\tau = 1$ **(d)** $\tau = 2$

Figure 4.5: FD-Dir learns to evolve the Dirichlet distribution in continuous time.

where the real-valued scalar ν is a constant prior on $\log \alpha_c(\tau)$ which takes over in regions where the Gaussians are close to 0. The decomposition into a sum of Gaussians is beneficial for various reasons:

1. Note that the concentration parameter α_c can be viewed as the effective number of observations of class c. Accordingly, the larger the value $\log \alpha$, the more certain becomes the prediction. Functions $\mathcal{N}(\tau|\tau_j^{(c)}, \sigma_j^{(c)})$ can then describe time regions where we observed data and, thus, correspond to more certainty, in regions around the time $\tau_j^{(c)}$ where the 'width' is controlled by $\sigma_j^{(c)}$.

2. Since most of the functions' mass is centered around the mean, the locality property is fulfilled. Put differently: In regions where we did not observe data, that is, where the functions $\mathcal{N}(\tau|\tau_j^{(c)}, \sigma_j^{(c)})$ are close to 0, the value $\log \alpha_c(\tau)$ is close to the prior ν. In the experiments, we use $\nu = 0$ so $\alpha_c(\tau) = 1$ holds in the out-of-observed data regions. This is a common uninformative prior for the Dirichlet parameters. Specifically for $\tau \to \infty$, the resulting predictions have high uncertainty.

3. Lastly, a linear combination of translated Gaussians is able to approximate a large family of functions [27]. And similarly to the weighted GP, the coefficients $w_j^{(c)}$ allow discarding unnecessary basis functions.

The basis functions parameters $(w_j^{(c)}, \tau_j^{(c)}, \sigma_j^{(c)})$ are the output of the neural network, and can also be interpreted as weighted pseudo points that determine the regression of Dirichlet parameters $\theta(\tau)$, that is, $\alpha_c(\tau)$ over time (Figure 4.2 & Figure 4.5). The concentration parameters $\alpha_c(\tau)$ themselves also have a natural interpretation: they can be viewed as the rate of events after time gap τ.

Example. Our goal was to model the evolution of a distribution on a probability simplex. Figure 4.1b shows this for two classes. In general, we can do the same for multiple classes. Figure 4.5 shows an example of the Dirichlet distribution for three classes, and how it changes over time. This evolution is the output of the FD-Dir model trained on a synthetic dataset that is created to mimic the car example from the introduction (see also Figure B.5a in Appendix B.3). The three classes: *overtaking*, *breaking* and *collision* occur independently of each other at three different times. The corners of the triangle correspond to the classes.

We can distinguish three cases: (a) at first we are certain that the most likely class is *overtaking*—high density at the bottom-left corner; (b) as time passes, the most likely class becomes *breaking*, (c) and finally *collision*. After that, we are in the area where we have not seen any data and do not have a confident prediction (d)—density is uniform over the whole triangle.

4.1.3 Model training with the distributional uncertainty loss

The core feature of our models is forecasting into the future with uncertainty. The classical cross-entropy loss, however, is not well suited to learn uncertainty on the categorical distribution since it is only based on a single (point estimate) of the class distribution. That is, the standard cross-entropy loss for the ith event between the true categorical distribution $\boldsymbol{p}_i^*$ and the predicted mean categorical distribution $\overline{\boldsymbol{p}}_i$ is:

$$\mathcal{L}_i^{\text{CE}} = \text{H}[\boldsymbol{p}_i^*, \overline{\boldsymbol{p}}_i(\tau_i^*)] = -\sum_c p_{ic}^* \log \overline{p}_{ic}(\tau_i^*)$$

Because of the point estimate $\overline{\boldsymbol{p}}_i(\tau) = \mathbb{E}_{\boldsymbol{p}_i \sim P_i(\theta(\tau))}[\boldsymbol{p}_i]$, the uncertainty on $\boldsymbol{p}_i$ is ignored. Instead, we propose the loss which takes into account uncertainty, which we call *uncertainty cross-entropy*:

$$\mathcal{L}_i^{\text{UCE}} = \mathbb{E}_{\boldsymbol{p}_i \sim P_i(\theta(\tau_i^*))}[\text{H}[\boldsymbol{p}_i^*, \boldsymbol{p}_i]] = -\int P_i(\theta(\tau_i^*)) \sum_c p_{ic}^* \log p_{ic} \tag{4.7}$$

Remark that the uncertainty cross-entropy does not use the compound distribution $\overline{\boldsymbol{p}}_i(\tau)$ but considers the expected cross-entropy. Based on Jensen's inequality, it holds: $0 \leq \mathcal{L}_i^{\text{CE}} \leq \mathcal{L}_i^{\text{UCE}}$. Consequently, a low value of the uncertainty cross-entropy guarantees a low value for the classical cross-entropy loss, while additionally taking the variation in the class probabilities into account. A comparison between the classical cross-entropy and the uncertainty cross-entropy on a simple classification task and anomaly detection in irregular event setting is presented in Appendix B.2.

In practice the true distribution $\boldsymbol{p}_i^*$ is often a one hot-encoded representation of the observed class c_i which simplifies the computations. During training, the models compute $P_i(\theta(\tau))$ and evaluate it at the true time of the next event τ_i^* given the past event e_{i-1} and the history $\mathcal{H}_{i-1}$. The final loss for a sequence of events is simply obtained by summing up the loss for each event $\mathcal{L} = \sum_i \mathbb{E}_{\boldsymbol{p}_i \sim P_i(\theta(\tau_i^*))}[\text{H}[\boldsymbol{p}_i^*, \boldsymbol{p}_i]]$.

Fast computation. In order to have an efficient computation of the uncertainty cross-entropy, we propose closed-form expressions.

(1) *Closed-form loss for Dirichlet.* Given that the observed class c_i is one hot-encoded by $\boldsymbol{p}_i^*$, the uncertainty loss can be computed in closed-form for the Dirichlet distribution:

$$\mathcal{L}_i^{\text{UCE}} = \mathbb{E}_{\boldsymbol{p}_i(\tau_i^*) \sim \mathbf{Dir}(\boldsymbol{\alpha}(\tau_i^*))}[\log p_{c_i}(\tau_i^*)] = \Psi(\alpha_{c_i}(\tau_i^*)) - \Psi(\alpha_0(\tau_i^*)), \tag{4.8}$$

where Ψ denotes the digamma function and $\alpha_0(\tau_i^*) = \sum_c^C \alpha_c(\tau_i^*)$.

(2) *Loss approximation for GP.* For WGP-LN, we approximate $\mathcal{L}_i^{\text{UCE}}$ based on second order series expansion (proof is provided in Appendix B.1):

$$\begin{aligned}\mathcal{L}_i^{\text{UCE}} \approx \mu_{c_i}(\tau_i^*) - \log\left(\sum_c^C \exp\left(\mu_c(\tau_i^*) + \frac{\sigma_c^2(\tau_i^*)}{2}\right)\right) \\ + \frac{\sum_c^C (\exp(\sigma_c^2(\tau_i^*)) - 1)\exp(2\mu_c(\tau_i^*) + \sigma_c^2(\tau_i^*))}{2\left(\sum_c^C \exp\left(\mu_c(\tau_i^*) + \frac{\sigma_c^2(\tau_i^*)}{2}\right)\right)^2}.\end{aligned} \tag{4.9}$$

Note that we can take the gradient of the above losses and, therefore, fully backpropagate through our models, enabling us to implement them in any modern deep learning framework and train efficiently with gradient descent.

Regularization. While the above loss incorporates uncertainty much better compared to cross-entropy, it is still possible to generate pseudo points with high weight values outside of the observed data regime giving us predictions with high confidence. To eliminate this behavior we introduce a regularization term r_c:

$$r_c = \alpha \underbrace{\int_0^T (\mu_c(\tau))^2 \, d\tau}_{\text{Pushes mean to } 0} + \beta \underbrace{\int_0^T (\nu - \sigma_c^2(\tau))^2 \, d\tau}_{\text{Pushes variance to } \nu}. \tag{4.10}$$

For the WGP-LN, $\mu_c(\tau)$ and $\sigma_c(\tau)$ correspond to the mean and the variance of the class logits which are pushed to prior values of 0 and ν. For the FD-Dir, $\mu_c(\tau)$ and $\sigma_c(\tau)$ correspond to the mean and the variance of the class probabilities where the regularizer on the mean can actually be neglected because of the prior ν introduced in the function decomposition (Equation 4.6). In experiments, ν is set to 1 for WGP-LN and $\frac{C-1}{C^2(C+1)}$ for FD-Dir which is the variance of the classic Dirichlet prior with concentration parameters equal to 1. For both models, this regularizer forces high uncertainty on the interval $[0, T]$. In practice, the integrals can be estimated with Monte-Carlo sampling whereas α and β are hyperparameters which are tuned on a validation set.

In Malinin and Gales [171], training models capable of uncertain prediction requires out-of-distribution samples, by using another dataset or a generative model. In contrast, our regularizer suggests a simple way to consider out-of-distribution data which does not require another model or dataset.

4.1.4 Modeling the temporal point process

Our models FD-Dir and WGP-LN predict $P(\theta(\tau))$, enabling us to evaluate, for example, $\overline{\boldsymbol{p}}$ after a specific time gap τ. This corresponds to a conditional distribution $q(c|\tau) := \overline{p}_c(\tau)$ over the classes. In this section, we introduce a *point process* framework to generalize FD-Dir to also predict the time distribution $q(\tau)$. This enables us to predict, for example, the most likely time the next event is expected or to evaluate the joint distribution $q(c, \tau)$. We call the model FD-Dir-PP.

We modify the model so that each class c is modeled using an inhomogeneous Poisson point process with positive locally integrable intensity function $\lambda_c(\tau)$. Instead of generating parameters $\theta(\tau) = (\alpha_1(\tau), \dots, \alpha_C(\tau))$ by function decomposition, FD-Dir-PP generates intensity parameters over time: $\log \lambda_c(\tau) = \sum_{j=1}^{M} w_j^{(c)} \mathcal{N}(\tau | \tau_j^{(c)}, \sigma_j^{(c)}) + \nu$. The main advantage of such general decomposition is its potential to describe complex multimodal intensity functions contrary to other models like RMTPP [61] (see Chapter 3). Since the concentration parameter $\alpha_c(\tau)$ and the intensity parameter $\lambda_c(\tau)$ both relate to the number of events of class c around time τ, it is natural to convert one to the other.

Given this C-multivariate point process, the probability of the next class given time and the probability of the next event time are $q(c|\tau) = \frac{\lambda_c(\tau)}{\lambda_0(\tau)}$ and $q(\tau) = \lambda_0(\tau) e^{-\int_0^\tau \lambda_0(s) ds}$ where $\lambda_0(\tau) = \sum_{c=1}^{C} \lambda_c(\tau)$. Since the classes are now modeled via TPP, the log-likelihood of the event $e_i = (c_i, \tau_i^*)$ is:

$$\log q(c_i, \tau_i^*) = \log q(c_i|\tau_i^*) + \log q(\tau_i^*) = \underbrace{\log \frac{\lambda_{c_i}(\tau_i^*)}{\lambda_0(\tau_i^*)}}_{(i)} + \underbrace{\log \lambda_0(\tau_i^*)}_{(ii)} - \underbrace{\int_0^{\tau_i^*} \lambda_0(t) dt}_{(iii)} . \tag{4.11}$$

The terms (ii) and (iii) act like a regularizer on the intensities by penalizing large cumulative intensity $\lambda_0(\tau)$ on the time interval $[t_{i-1}, t_i]$ where no events occurred. This exactly corresponds to Equation 2.27. The term (i) is the standard cross-entropy loss at time τ_i. Or equivalently, by modeling the distribution $\mathbf{Dir}(\lambda_1(\tau), \dots, \lambda_C(\tau))$, we see that term (i) is equal to $\mathcal{L}_i^{\text{CE}}$ (see Section 4.1.3). Using this insight, we obtain our final FD-Dir-PP model: We achieve uncertainty on the class prediction by modeling $\lambda_c(\tau)$ as concentration parameters of a Dirichlet distribution and train the model with the loss from Equation 4.11 replacing the term (i) with $\mathcal{L}_i^{\text{UCE}}$. As it becomes apparent, FD-Dir-PP differs from FD-Dir only in the regularization of the loss function, enabling it to be interpreted as a point process.

4.2 Related work

Predictions based on discrete sequences of events ignoring the continuous time have been modeled by Markov models [9] and recurrent neural networks [103, 38]. To exploit the time information some models [157, 190] additionally take time as an input but still output a single prediction for the entire future. In contrast, temporal point process framework defines the intensity function that describes the rate of events occurring over time and can naturally capture the irregularly-sampled sequences (Chapter 3).

As discussed in Chapter 3, RMTPP [61] is not able to capture complex evolution over time, such as multimodal distributions. Similarly, classical TPPs [99, 112] are restricted to a certain type of behavior. On the other hand, neural Hawkes process [180] uses continuous-time LSTM which allows specifying more complex intensity functions but the likelihood evaluation is not in closed-form anymore. Finally, these approaches, unlike our models, do not provide the same kind of uncertainty in the predictions. We proposed the solution in Section 4.1.4 to extend WGP-LN and FD-Dir with a point process framework while having the expressive power to represent complex time evolutions.

Uncertainty in machine learning is, in general, an important topic with many proposed different approaches [73, 69, 148]. For example, uncertainty can be imposed by introducing distributions over the weights to define the so-called Bayesian neural networks [23, 175, 226]. A simpler approach is to introduce uncertainty on the class prediction directly, by using Dirichlet distribution [171, 230], similar to how we defined it but without time dependence. That is, FD-Dir models complex temporal evolution of Dirichlet distribution via function decomposition, which can also be adapted to have a point process interpretation. Other methods introduce uncertainty in time series forecasting by learning state space model with Gaussian processes [67, 269]. Alternatively, RNN architecture has been used to model the probability density function over time [290]. Compared to these models, WGP-LN uses both Gaussian processes and RNNs to model uncertainty together with the time.

Both models are defined using pseudo points which has a connection to inducing points in sparse Gaussian processes. There, the goal is to have a reduced computational complexity [247], while our goal is to give the points different importance. If necessary, we can speed up the computation simply by using less pseudo points. Wen et al. [279] introduce a weighted Gaussian process that rescales the data points; in contrast, our model uses a custom kernel to discard (pseudo) points.

4.3 Experiments

We evaluate our models on large-scale synthetic and real-world data. We compare to neural point process models: **RMTPP** [61] and **Neural hawkes process** [180]. Additionally, we use various RNN models with the knowledge of the time of the next event. We measure the accuracy of class prediction, accuracy of time prediction, and evaluate on an anomaly detection task to show prediction uncertainty.

We split the data into train, validation and test set (60%–20%–20%) and tune all models on a validation set using grid search over learning rate, hidden state dimension and L_2 regularization. After running models on all datasets 5 times we report the mean and the standard deviation of the test set accuracy. Details on model selection can be found in Appendix B.4.1. The code and further supplementary material is available online.[2] We use the following data (more details in Appendix B.3):

- **Graph.** We generate data from a directed Erdős-Rényi graph where nodes represent the states and edges the weighted transitions between them. The time it takes to cross one edge is modeled with one normal distribution per edge. By randomly walking along this graph we created 10k irregularly-sampled events with 10 unique classes.

- **Stack Exchange.** Sequences contain rewards as events that users get for participation on a question answering website. After preprocessing according to [61] we have 40 classes and over 480k events spread over 2 years of activity of around 6700 users. The goal is to predict the next reward a particular user will receive.

[2] `https://www.cs.cit.tum.de/daml/uncertainty-event-prediction/`

- **Smart Home** [33]**.** We use a recorded sequence from a smart house with 14 classes and over 1000 events. Events correspond to the usage of different appliances. The next event will depend on the time of the day, history of usage and other appliances.
- **Car Indicators.** We obtained a sequence of events from car's indicators that has around 4000 events with 12 unique classes. The sequence is highly irregular in time, with τ ranging from milliseconds to minutes.

4.3.1 Visualization

To analyze the behavior of the models, we propose visualizations of the evolutions of the parameters predicted by FD-Dir and WGP-LN.

Setup. We use two toy datasets where the probability of an event depends only on time. The first one (**3-G**) has three classes occurring at three distinct times. It represents the events in the Figure B.5a. The second one (**Multi-G**) consists of two classes where one of them has two modes and corresponds to the Figure 4.1a. We use these datasets to showcase the importance of time when predicting the next event. In Figure 4.6, the four top plots show the evolution of the categorical distribution for the FD-Dir and the logits for the WGP-LN with 10 points each. The four bottom plots describe the certainty of the models on the probability prediction by plotting the probability $q_c(\tau)$ that the probability of class c is higher than others, as introduced in Section 4.1.

Results. Both models learn meaningful evolutions of the distribution on the simplex. For the 3-G data, we can distinguish four areas: the first three correspond to the three classes; after that the prediction is uncertain. The Multi-G data shows that both models are able to approximate multimodal evolutions.

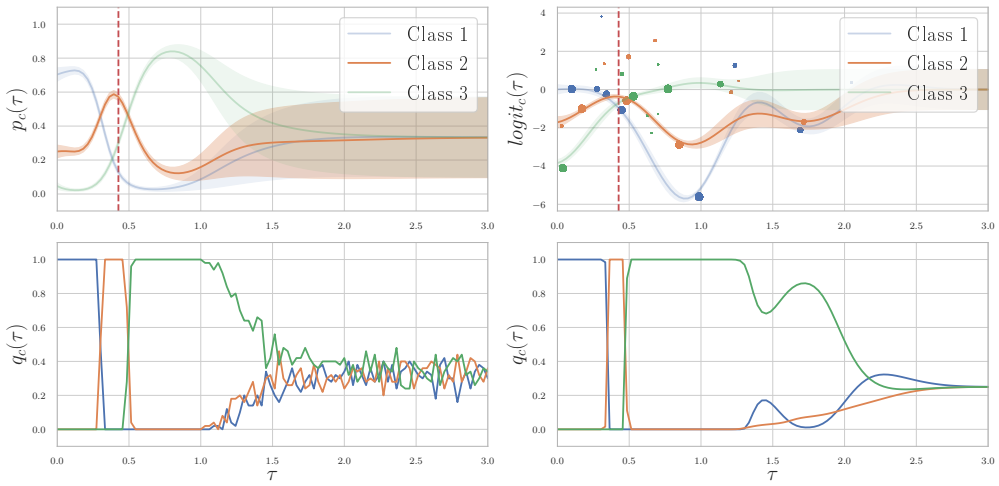
4.3.2 Class prediction accuracy

The goal of this experiment is to assess whether our models can correctly predict the class of the next event, given the time at which it occurs. For this purpose, we compare our models against Hawkes and RMTPP and evaluate prediction accuracy on the test set.

Results. We can see from Figure 4.7 that our models consistently outperform their competitors on all datasets. Results including other baselines can be found in Appendix B.4, where we reach the same conclusion—our models perform the best.

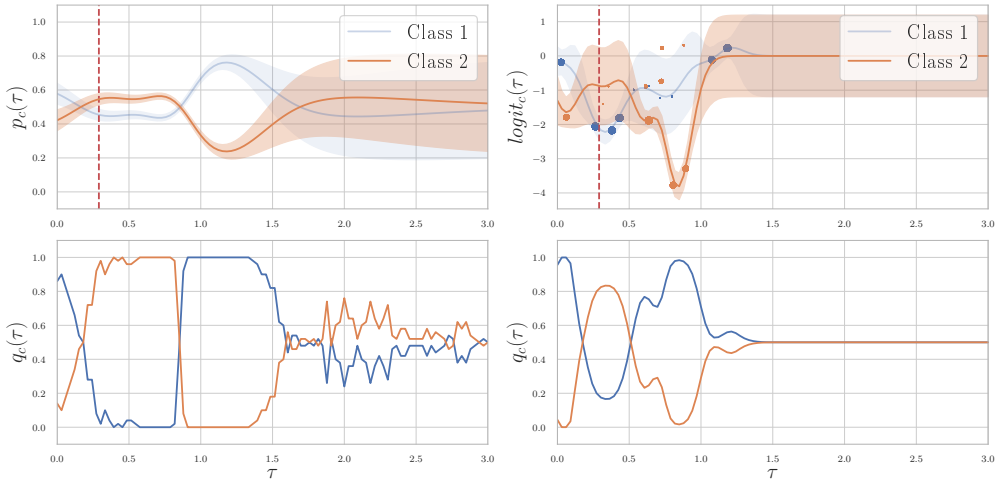
4.3.3 Time-Error evaluation

Next, we aim to assess the quality of the time intervals at which we have confidence in one class. Even though WGP-LN and the FD-Dir do not model explicit distribution over time, like TPPs, they still have intervals at which we are certain in a class prediction, making the conditional probability a good indicator of the time occurrence of the event.



FD-Dir on 3-G

WGP-LN on 3-G



FD-Dir on Multi-G

WGP-LN on Multi-G

Figure 4.6: Visualization of the prediction evolution. The dotted vertical red line indicates the true time of the next event for the example sequence. Here, both models correctly predict the second class (in orange), and capture the variation of the class distributions over time. Generated points from WGP-LN are plotted with the size corresponding to the weight. We can see that the model does not need all the pseudo points to capture the true evolution of the prediction so it discards them by assigning small weights to them. For predictions in the far future, both models given high uncertainty.

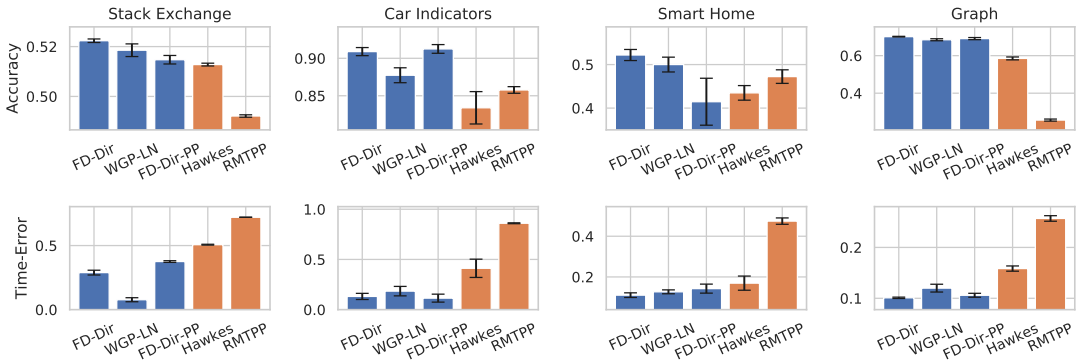


Figure 4.7: Class accuracy (top; higher is better) and Time-Error (bottom; lower is better).

Setup. While models predicting a *single* time $\hat{\tau}_i$ for the next event often use the MSE score $\frac{1}{n}\sum_{i=1}^{n}(\hat{\tau}_i - \tau_i^*)^2$, in our case the MSE is not suitable since one event can occur at multiple time points. In the conventional least-squares approach, the mean of the true distribution is an optimal prediction; however, here it is almost always wrong. Therefore, we use another metric which is better suited for multimodal distributions. Assume that a model returns a score function $g_i^{(c)}(\tau)$ for each class regarding the next event i, where a large value means the class c is likely to occur at time τ. We define:

$$\text{Time-Error} = \frac{1}{n}\sum_{i=1}^{n}\int \mathbf{1}_{g_i^{(c)}(\tau) \geq g_i^{(c)}(\tau_i^*)}\,\mathrm{d}\tau.$$

The Time-Error computes the size of the time intervals where the predicted score is larger than the score of the observed time τ_i^*. Hence, a performant model would achieve a low Time-Error if its score function $g_i^{(c)}(\tau)$ is high at time τ^*. As the score function in our models, we use the corresponding class probability $\bar{p}_{ic}(\tau)$.

Results. We can see that our models clearly obtain the best results on all datasets. The point process version of FD-Dir does not improve the performance. Thus, taking also into account the class prediction performance, we recommend to use our other two models. In Appendix B.4.2 we compare FD-Dir-PP with other neural point process models on time prediction using the MSE score and achieve similar results.

4.3.4 Anomaly detection and uncertainty

The goal of this experiment is twofold: (1) it assesses the ability of the models to detect anomalies in irregular sequences, (2) it evaluates the quality of the predicted uncertainty on the categorical distribution. For this, we use a similar set-up as described in Malinin and Gales [171].

Setup. The experiments consist in introducing anomalies in datasets by changing the occurrence time of 10% of the events (at random after the time transformation described in

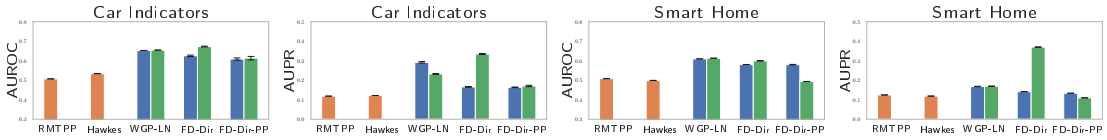


Figure 4.8: AUROC and APR comparison on different datasets for the anomaly detection task. The orange and blue bars use categorical uncertainty score whereas the green bars use distributional uncertainty.

Appendix B.3). Hence, the anomalies form out-of-distribution data, whereas unchanged events represent in-distribution data. The performance of the anomaly detection is measured using an area under the receiver operating characteristic (AUROC) and an area under the precision-recall curve (AUPR). We use two approaches: (i) We consider the *categorical uncertainty* on $\bar{\boldsymbol{p}}(\tau)$, that is, we detect anomalies by using the predicted probability of the true event as the anomaly score. (ii) We use the *distribution uncertainty* at the observed occurrence time provided by our models. For WGP-LN, we can evaluate $q_c(\tau)$ directly as a difference between the two normal distributions. For FD-Dir, this probability does not have a closed-form solution so, instead, we use the concentration parameters which are also indicators of out-of-distribution events. For all scores, that is, for $\bar{\boldsymbol{p}}(\tau)_c$, $q_c(\tau)$ and $\alpha_c(\tau)$, a low value indicates a potential anomaly around time τ.

Results. As seen in Figure 4.8, the FD-Dir and the WGP-LN have particularly good performance. We observe that the FD-Dir gives better results especially with distributional uncertainty. This might be due to the power of the concentration parameters that can be viewed as number of similar events around a given time.

4.4 Conclusion

We proposed two new methods to predict the evolution of the probability of the next event in irregularly-sampled sequences, including the distributions' uncertainty. Both methods follow a common framework consisting in generating pseudo points able to describe rich multimodal time-dependent parameters for the distribution over the probability simplex. The complex evolution is captured via Gaussian Process or function decomposition, respectively; while still enabling straightforward training. We also provided an extension and interpretation within a point process framework. In the experiments, WGP-LN and FD-Dir have clearly outperformed state-of-the-art models based on point processes; for event and time prediction as well as for anomaly detection.

5 Neural Flows

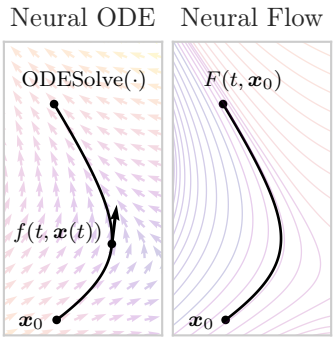


Figure 5.1: Comparison between learning a vector field (Left) and learning a flow (Right). Neural ordinary differential equations require a numerical solver and evaluate the neural network f at many points along the solution curve. Our approach learns the solutions from initial conditions directly, avoiding the solvers.

Ordinary differential equations (ODEs) are among the most important tools for modeling complex systems, both in natural and social sciences. They describe the *instantaneous change* in the system, which is often an easier way to model physical phenomena than specifying the whole system itself. For example, the change of the pendulum angle or the change in population can be naturally expressed in the differential form. Similarly, Chen et al. [35] introduce neural ODEs that describe how some quantity of interest, represented as a vector $\boldsymbol{x}$, changes with time: $\dot{\boldsymbol{x}} = f(t, \boldsymbol{x}(t))$, where f is now a neural network. We repeat the solution to a given ODE from Section 2.1.1: starting at some initial value $\boldsymbol{x}(t_0)$ measured at time t_0 we can find the result of this dynamic at any t_1:

$$\boldsymbol{x}(t_1) = \boldsymbol{x}(t_0) + \int_{t_0}^{t_1} f(t, \boldsymbol{x}(t))\,\mathrm{d}t = \mathrm{ODESolve}(\boldsymbol{x}(t_0), f, t_0, t_1). \tag{2.4}$$

As already discussed in Section 2.1.1, it is sufficient for f to be continuous in t and Lipschitz continuous in $\boldsymbol{x}$ to have a unique solution. This mild condition is already satisfied by a large family of neural networks. In most practically relevant scenarios, the integral in Equation 2.4 has to be solved numerically, requiring a trade-off between computation cost and numerical precision. Much of the followup work to [35] focused on retaining expressive dynamics while requiring fewer solver evaluations [71, 125].

In the machine learning context we are given a set of initial conditions (often at $t_0 = 0$) and a loss function for the solution evaluated at time t_1. One example is modeling time series where the latent state is evolved in continuous time and is used to predict the observed measurements [51]. Here, unlike in physics for example, the function f is completely unknown and needs to be learned from data. Thus, Chen et al. [35] use neural networks for their ability to capture complex dynamics. Note, however, that unlike in physics the resulting ODE is not as interpretable.

Since solving an ODE is expensive, we want to find a way to keep the desired properties of neural ODEs at a much smaller computation cost. If we take a step back, we see that neural ODEs take initial values as inputs and return non-intersecting solution curves (Figure 5.1). We propose to model the solution curves directly, with a neural network, instead of specifying the derivative. That is, given an initial condition we return the solution with a single forward pass through our network. Straight away, this leads to improvements in computation performance because we avoid using ODE solvers altogether. We will show how our method can be used as a faster alternative to ODEs in existing models [34, 51, 118, 229], while improving the modeling performance at the same time. In the following, we derive the conditions that our method needs to satisfy and propose different architectures that implement them.

5.1 Method

In this section, we present our method, *neural flows*, that directly models the solution curve of an ODE with a neural network. For simplicity, let us briefly assume that the initial condition $\boldsymbol{x}_0 = \boldsymbol{x}(t_0)$ is specified at $t_0 = 0$. We handle the general case shortly. Then, Equation 2.4 can be written as $\boldsymbol{x}(t) = F(t, \boldsymbol{x}_0)$, where F is the solution to the initial value problem, $\dot{\boldsymbol{x}} = f(t, \boldsymbol{x}(t))$, $\boldsymbol{x}_0 = \boldsymbol{x}(0)$. We will model F with a neural network. For this, we first list the conditions that F must satisfy so that it is a solution to some ODE. Let $F : [0, T] \times \mathbb{R}^d \to \mathbb{R}^d$ be a smooth function satisfying:

i) $F(0, \boldsymbol{x}_0) = \boldsymbol{x}_0$,

ii) $F(t, \cdot)$ is invertible, $\forall t$.

Property i) ensures we satisfy the initial condition. Property ii) ensures the uniqueness of the solution given the initial value $\boldsymbol{x}_0$, that is, the curves specified by F corresponding to different initial values do not intersect for any t.

There is an exact correspondence between a function F with the above properties and an ODE defined with f such that the derivative $\frac{\mathrm{d}}{\mathrm{d}t}F(t, \boldsymbol{x}_0)$ matches $f(t, \boldsymbol{x}(t))$ everywhere, given $\boldsymbol{x}_0 = \boldsymbol{x}(0)$ [152, Theorem 9.12]. In general, we can say that f defines a vector field and F defines a family of integral curves, also known as the *flow* in mathematics.[1] As F will be parameterized with a neural network, property i) requires that its parameters must depend on t such that we have an identity map at $t = 0$.

Note that by providing $\boldsymbol{x}_0$ we define a smooth trajectory $F(\cdot, \boldsymbol{x}_0)$—the solution to some ODE with the initial condition at $t_0 = 0$. If we relax the restriction $t_0 = 0$ and allow $\boldsymbol{x}_0$ to be specified at an arbitrary $t_0 \in \mathbb{R}$, the solution can be obtained with a simple procedure. We first go back to the case $t = 0$ where we obtain the corresponding "initial" value $\hat{\boldsymbol{x}}_0 := \boldsymbol{x}(0) = F^{-1}(t_0, \boldsymbol{x}_0)$. This then gives us the required solution $F(\cdot, \hat{\boldsymbol{x}}_0)$ to the original initial value problem. Thus, we often prefer functions with an analytical inverse.

Finally, we tackle implementing F. The second property instructs us that the function $F(t, \cdot)$ is a diffeomorphism on $\mathbb{R}^d$. We can satisfy this by drawing inspiration from existing

[1] Not to be confused with normalizing flow from Section 2.2.2, although, we will take inspiration in normalizing flows to satisfy the invertibility constraint.

works on normalizing flows and invertible neural networks (Section 2.2.2). In our case, the parameters must be conditioned on time, with identity at $t = 0$. As a starting example, consider a linear ODE $f(t, \boldsymbol{x}(t)) = \boldsymbol{A}\boldsymbol{x}(t)$, with $\boldsymbol{x}(0) = \boldsymbol{x}_0$. Its solution can be expressed as $F(t, \boldsymbol{x}_0) = \exp(\boldsymbol{A}t)\boldsymbol{x}_0$, where exp is the matrix exponential. Here, the learnable parameters $\boldsymbol{A}$ are simply multiplied by t to ensure property i); and given fixed t, the network behaves as an invertible linear transformation. In the following we propose other, more expressive functions suitable for applications such as time series modeling.

5.1.1 Proposed architectures

ResNet flow. A single residual layer $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + g(\boldsymbol{x}_t)$ [100] is similar to Equation 2.4 since we can view it as a discretized version of a continuous transformation, as was already discussed in Section 2.1.1. Although plain ResNets are not invertible, one could use spectral normalization [87] to enforce a small Lipschitz constant of the network, which guarantees invertibility [10, Theorem 1]. Thus, ResNets become a natural choice for modeling the solution curve F resulting in the following extension:

$$F(t, \boldsymbol{x}) = \boldsymbol{x} + \varphi(t)g(t, \boldsymbol{x}), \tag{5.1}$$

where $\varphi : \mathbb{R} \to \mathbb{R}^d$. This satisfies properties i) and ii) from above when $\varphi(0) = \mathbf{0}$ and $|\varphi(t)_i| < 1$; and $g : \mathbb{R}^{d+1} \to \mathbb{R}^d$ is an arbitrary contractive neural network ($\mathrm{Lip}(g) < 1$). One simple choice for φ is a tanh function. The inverse of F can be found via fixed point iteration similar to [10].

GRU flow. Time series data is traditionally modeled with recurrent neural networks, for example, with a gated recurrent unit (GRU) [38], such that the hidden state $\boldsymbol{h}_{t-1}$ is updated at fixed intervals with the new observation $\boldsymbol{x}_t$:

$$\boldsymbol{h}_t = \mathrm{GRUCell}(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t) = \boldsymbol{z}_t \odot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{z}_t) \odot \boldsymbol{c}_t, \tag{5.2}$$

where $\boldsymbol{z}_t$ and $\boldsymbol{c}_t$ are the result of the transformation acting on the combination of the previous state $\boldsymbol{h}_{t-1}$ and the new input $\boldsymbol{x}_t$ (see Section 2.1.2).

De Brouwer et al. [51] derived the continuous equivalent of this architecture called GRU-ODE. Given the initial condition $\boldsymbol{h}_0 = \boldsymbol{h}(t_0)$, they evolve the hidden state $\boldsymbol{h}(t)$ between the observations with an ODE:

$$\frac{\mathrm{d}\boldsymbol{h}(t)}{\mathrm{d}t} = (1 - \boldsymbol{z}(t)) \odot (\boldsymbol{c}(t) - \boldsymbol{h}(t)). \tag{5.3}$$

With the new observation $\boldsymbol{x}$, the hidden state is updated using a discrete GRU update (Equation 5.2). In summary, given an initial state $\boldsymbol{h}_0$ and the observation $\boldsymbol{x}_{t_1}$ at t_1 we get the updated hidden state in a two-step process:

$$\begin{aligned} &1. \quad \bar{\boldsymbol{h}}_{t_1} = \mathrm{ODESolve}(\boldsymbol{h}_0, \text{Equation 5.3}, t_0, t_1), \\ &2. \quad \boldsymbol{h}_{t_1} = \mathrm{GRUCell}(\bar{\boldsymbol{h}}_{t_1}, \boldsymbol{x}_{t_1}). \end{aligned} \tag{5.4}$$

Here, we will derive the flow version of GRU-ODE. If we rewrite Equation 5.2 by regrouping the terms: $\boldsymbol{h}_t = \boldsymbol{h}_{t-1} + (1 - \boldsymbol{z}_t) \odot (\boldsymbol{c}_t - \boldsymbol{h}_{t-1})$, we see that GRU update acts as a single ResNet layer. This leads to the following result.

Definition 1. *Let* $f_z, f_r, f_c : \mathbb{R}^{d+1} \to \mathbb{R}^d$ *be any arbitrary neural networks and let* $z(t, \boldsymbol{h}) = \alpha \cdot \sigma(f_z(t, \boldsymbol{h}))$, $r(t, \boldsymbol{h}) = \beta \cdot \sigma(f_r(t, \boldsymbol{h}))$, $c(t, \boldsymbol{h}) = \tanh(f_c(t, r(t, \boldsymbol{h}) \odot \boldsymbol{h}))$, *where* $\alpha, \beta \in \mathbb{R}$ *and* σ *is a sigmoid function. Further, let* $\varphi : \mathbb{R} \to \mathbb{R}^d$ *be a continuous function with* $\varphi(0) = \mathbf{0}$ *and* $|\varphi(t)_i| < 1$. *Then the evolution of GRU state in continuous time is defined as:*

$$F(t, \boldsymbol{h}) = \boldsymbol{h} + \varphi(t)(1 - z(t, \boldsymbol{h})) \odot (c(t, \boldsymbol{h}) - \boldsymbol{h}). \tag{5.5}$$

Theorem 2. *A neural network defined by Equation 5.5 specifies a flow when the functions* f_z, f_r *and* f_c *are contractive maps; it is sufficient* $\mathrm{Lip}(f_\cdot) < 1$, *and* $\alpha = \frac{2}{5}$, $\beta = \frac{4}{5}$.

We prove Theorem 2 in Appendix C.1.2 by showing that the second summand on the right hand side in Equation 5.5 satisfies Lipschitz constraint making the whole network invertible. We also show that the GRU flow has the same desired properties as GRU-ODE, namely, bounding the hidden state in $(-1, 1)$ and having the Lipschitz constant of 2. Note that GRU flow (Equation 5.5) acts as a replacement to ODESolve in Equation 5.4. Alternatively, we can append $\boldsymbol{x}_t$ to the input of f_z, f_r and f_c, which would give us a continuous-in-time version of GRU.

Coupling flow. The disadvantage of both the ResNet flow and GRU flow is the missing analytical inverse. Although it is not always required, having an easy to compute inverse will be useful in a density estimation task. To this end, we propose a continuous-in-time version of an invertible transformation based on splitting the input dimensions into two disjoint sets A and B, $A \cup B = \{1, 2, \ldots, d\}$. Recall from Equation 2.33 (Section 2.2.2) that we copy the values indexed by B and transform the rest with, for example, an affine function. To make this a flow we also have to implement the initial condition:

$$F(t, \boldsymbol{x})_A = \boldsymbol{x}_A \exp(u(t, \boldsymbol{x}_B)\varphi_u(t)) + v(t, \boldsymbol{x}_B)\varphi_v(t), \tag{5.6}$$

where u, v are arbitrary neural networks and $\varphi_u(0) = \varphi_v(0) = \mathbf{0}$. We can easily see that this satisfies property i): at $t = 0$, $\boldsymbol{x}_B$ is copied regardless and Equation 5.6 evaluates to $\boldsymbol{x}_A$. Since it is also invertible by design [59], regardless of t, it defines a proper flow. Same as in Section 2.2.2, we apply multiple consecutive transformations, choosing different partitions A and B, as some values stay constant after a single transformation.

More generally, for all flow models we can stack multiple layers of transformations $F = F_1 \circ \cdots \circ F_n$ and still define a proper flow since the composition of invertible functions is invertible, and consecutive identities give an identity.

We can think of φ (including φ_u, φ_v) as a time embedding function that has to be zero at $t = 0$. Since it is a function of a single variable, we would like to keep the complexity low and avoid using general neural networks in favor of interpretable and expressive basis functions. A simple example is linear dependence on time $\varphi(t) = \boldsymbol{\alpha} t$, or $\tanh(\boldsymbol{\alpha} t)$ for ResNet flow. An alternative, more powerful embedding consists of Fourier features $\varphi(t)_i = \sum_k \boldsymbol{\alpha}_{ik} \sin(\boldsymbol{\beta}_{ik} t)$.

5.1.2 On approximation capabilities

Previous works established that neural ODEs are *sup*-universal for diffeomorphic functions [265] and are L^p-universal for continuous maps when composed with terminal family [153]. A similar result also holds for affine coupling flows [264], whereas general residual networks can approximate any function [163]. The ResNet flow, as defined in Equation 5.1, can be viewed as an Euler discretization, meaning it is enough to stack appropriately many layers to uniformly approximate any ODE solution [153]. GRU flow can be viewed as a ResNet flow and coupling flow shares a similar structure, meaning that if we can set them to act as an Euler discretization we can match any ODE. However, this is of limited use in practice, as is often the case with theoretical approximation results, since we can only use finitely many layers. The main focus of this work is to provide the empirical evidence that we can outperform neural ODEs on relevant real-world tasks.

Other results [62, 295] consider limitations of neural ODEs in modeling general homeomorphisms (for example, $x \mapsto -x$) and propose the solution that adds dimensions to the input $\boldsymbol{x}$. Such augmented networks can model higher order dynamics. This can be explicitly defined through certain constraints for further improvements in performance and better interpretability [193]. We can apply the same trick to our models. However, instead of augmenting $\boldsymbol{x}$, a simpler solution is to relax the conditions on F given the task. For example, if we do not need invertibility, we can remove the Lipschitz constraint in Equation 5.1. Since neural flows offer such flexibility, they might be of more practical relevance in these use cases.

5.2 Applications

In this section we review two main applications of neural ODEs: modeling irregularly-sampled time series and density estimation. We describe the existing modeling approaches and propose extensions using neural flows. In Section 5.3 we will use models presented here to qualitatively and quantitatively compare neural flows with neural ODEs.

5.2.1 Continuous-time latent variable models

Autoregressive [200, 232] and state space models [110, 218] have achieved considerable success modeling regularly-sampled time series. However, many real-world applications do not have a constant sampling rate and may contain missing values, e.g., in healthcare we have very sparse measurements at irregular time intervals. Here we describe how our neural flow models can be used in such scenario.

Encoder. In this setting, we are given a sequence of observations $\boldsymbol{X} = (\boldsymbol{x}_1, \dots, \boldsymbol{x}_n)$, $\boldsymbol{x}_i \in \mathbb{R}^d$ at times $\boldsymbol{t} = (t_1, \dots, t_n)$. To represent this type of data, previous RNN-based works relied on exponentially decaying hidden state [32], time gating [190], or simply adding time as an additional input [61]. More recently, various ODE-based models built on top of RNNs to evolve the hidden state between observations in continuous time, giving rise to, e.g., ODE-RNN [229], while outperforming previous approaches. Another

model is GRU-ODE [51], which we already described in Equation 5.4. We proposed the GRU flow (Equation 5.5) that can be used as a straightforward replacement.

Lechner and Hasani [149] showed that simply evolving the hidden state with a neural ODE can cause vanishing or exploding gradients, a known issue in RNNs (Section 2.1.2). Thus, they propose using an LSTM-based [103] model instead. The difference to ODE-RNN [229] is using an LSTMCell and introducing another hidden state that is not updated continuously in time, which in turn allows gradient propagation via internal LSTM gating. To adapt this to our framework, we simply replace the ODESolve with the ResNet or coupling flow to obtain a neural flow model.

Decoder. Once we have a hidden state representation $\boldsymbol{h}_i$ of the irregularly-sampled sequence up to $\boldsymbol{x}_i$, we are interested in making future predictions. The ODE based models continue evolving the hidden state using a numerical solver to get the representation at time t_{i+1}, with $\boldsymbol{h}_{i+1} = \text{ODESolve}(\boldsymbol{h}_i, f, t_i, t_{i+1})$. With neural flows we can simply pass the next time point t_{i+1} into F and get the next hidden state directly. In the following we show how the presented encoder-decoder model is used in both the smoothing and filtering approaches for irregular time series modeling.

Smoothing approach. The given sequence of observations $(\boldsymbol{X}, \boldsymbol{t})$ is modeled with latent variables or states $(\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) \sim \mathbb{R}^h$, such that $\boldsymbol{x}_i \sim p(\boldsymbol{x}_i|\boldsymbol{z}_i)$, conditionally independent of other $\boldsymbol{x}_j$ [35, 229]. That is, we assume there is a continuous underlying sequence of latent states which produces our observations.[2] There is a predesignated prior state $\boldsymbol{z}_0$ at $t = 0$ from which the latent state is assumed to evolve continuously. More precisely, if $\boldsymbol{z}_0$ is a sample from the initial latent state distribution, then a latent state sample at any future time step t is given by $\boldsymbol{z}_t = F(t, \boldsymbol{z}_0)$.

Since the exact inference on the initial state $\boldsymbol{z}_0$, $p(\boldsymbol{z}_0|\boldsymbol{X}, \boldsymbol{t})$, is intractable, we proceed by doing approximate inference following the variational auto-encoder approach [35, 229]. Recall from Section 2.2.3 that we have to define an approximate posterior $q(\boldsymbol{z}_0|\boldsymbol{X}, \boldsymbol{t})$ conditioned on the input data. Using an LSTM-based neural flow encoder we process the input $(\boldsymbol{X}, \boldsymbol{t})$ to obtain the vector representation of time series (final hidden state), and output the posterior parameters $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ to define $q(\boldsymbol{z}_0|\boldsymbol{X}, \boldsymbol{t}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}\boldsymbol{I})$.

The decoder returns all $\boldsymbol{z}_i$ deterministically at times $\boldsymbol{t}$ with $F(t, \boldsymbol{z}_0)$, with initial condition $\boldsymbol{z}_0 \sim q(\boldsymbol{z}_0|\boldsymbol{X}, \boldsymbol{t})$. For the latent state at an arbitrary t_i, the target is generated according to the model $\boldsymbol{x}_i \sim p(\boldsymbol{x}_i|\boldsymbol{z}_i)$. Given the prior $p(\boldsymbol{z}_0) = \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$, the overall model is trained by maximizing the evidence lower bound, same as in Equation 2.42:

$$\text{ELBO} = \mathbb{E}_{\boldsymbol{z}_0 \sim q(\boldsymbol{z}_0|\boldsymbol{X}, \boldsymbol{t}))}[\log p(\boldsymbol{X})] - \text{KL}[q(\boldsymbol{z}_0|\boldsymbol{X}, \boldsymbol{t})||p(\boldsymbol{z}_0)]. \tag{5.7}$$

Using continuous time models brings up multiple advantages, from handling irregular time points automatically to making predictions at any, and as many time points as required. This allows us to do reconstruction of the time series, impute missing values

[2] This would correspond to treating the data as continuous instead of event-based (see Chapter 1). Note that we will still model event data using the TPP framework because the intensity function is continuous.

and forecast into the future. Since both neural flows and ODEs share the same desired properties, the defining difference becomes the computational complexity, where we offer a more efficient model which will matter especially as we scale to bigger data.

Filtering approach. In contrast to the previous approach, we can alternatively do the inference in an online fashion at each of the observed time points, that is, by estimating the posterior $p(\boldsymbol{z}_i|\boldsymbol{x}_{1:i}, \boldsymbol{t}_{1:i})$ after seeing observations until the current time step i. This is also known as filtering. Here, the prediction for future time steps is done by evolving the posterior corresponding to the final observed time point $p(\boldsymbol{z}_n|\boldsymbol{X}, \boldsymbol{t})$ instead of the initial time point $p(\boldsymbol{z}_0|\boldsymbol{X}, \boldsymbol{t})$, as was done in the smoothing approach.

In this work, we follow the general approach suggested by De Brouwer et al. [51] for capturing non-linear dynamics. We use GRU flow (instead of GRU-ODE) for evolving the hidden state $\boldsymbol{h}_i \in \mathbb{R}^h$ and we output the mean and the variance of the approximate posterior $q(\boldsymbol{z}_i|\boldsymbol{x}_{1:i}, \boldsymbol{t}_{1:i})$. The log-likelihood cannot be computed exactly under this model so De Brouwer et al. [51] suggest using a custom objective that is the analogue to Bayesian filtering (see Appendix C.1.1 for details). Unlike [51], which needs to solve one ODE per observation, our method only needs a single pass through the network for every observation, therefore, it is again more computationally efficient.

5.2.2 Temporal point processes

Sometimes temporal data is measured irregularly *and* the times at which we observe the events come from some underlying process modeled with temporal point processes (see Section 2.2.1 and Chapter 3). We reuse the notation and observe a realization of a TPP on an interval $[0, T]$ as an increasing sequence of arrival times $\boldsymbol{t} = (t_1, \dots, t_n)$, $t_i \in [0, T]$, where n is a random variable. The model is defined with an intensity function $\lambda(t)$ which has to be positive. We define the history $\mathcal{H}_{t_i}$ as the events that precede t_i, and further define the conditional intensity function $\lambda^*(t)$ which depends on history. Alternatively, working with inter-event times $\tau_i = t_i - t_{i-1}$ is equivalent. We train the model by maximizing the likelihood defined in Equation 2.27.

Following the approach from Chapter 3 we use an autoregressive model to represent the history with a fixed-size vector $\boldsymbol{h}_i$. We differentiate between two approaches:

1) Exact likelihood—The intensity function can correspond to a simple distribution [61] or a mixture of distributions [240].

2) Estimated likelihood—Modeling $\lambda(t)$ with an arbitrary neural network which requires Monte Carlo integration [14, 180].

In the context of neural ODE models, Jia and Benson [118] propose a jump ODE model that evolves the hidden state $\boldsymbol{h}(t)$ with an ODE and updates the state with new observations, similar to LSTM-ODE. In this case, obtaining the hidden state and solving the integral in Equation 2.27 can be done in a single solver call. That means the numerical ODE solver computes both the integral corresponding to the solution of the intensity function at t and the area under the intensity curve up to t, at the same time. Note that

this is still more expensive than having a closed-form likelihood since such models can simply evaluate the intensity at a point without a solver. We will define both versions of models, 1) those that replace the RNN from previous works with a neural flow encoder but keep the closed-form loss, and 2) those that evolve the intensity with a neural flow and estimate the loss using Monte Carlo.

Marked point processes. Since we are often interested in what type of event happened at time point t_i, we can model the observed type $\boldsymbol{x}_i$, also called mark, along with the arrival times: $p(\boldsymbol{t}, \boldsymbol{X}) = p(\boldsymbol{t})p(\boldsymbol{X}|\boldsymbol{t})$. Written like this, we can keep the TPP model for arrival times as discussed above. We add a module that inputs the history $\boldsymbol{h}_i$ and the next time point t_{i+1} that outputs the probabilities for each mark type. The special case is having continuous marks $\boldsymbol{x}_i \in \mathbb{R}^d$ which is covered in the next section.

5.2.3 Time-dependent density estimation

We again denote the arrival times with $\boldsymbol{t}$ and marks with $\boldsymbol{X}$, $\boldsymbol{x}_i \in \mathbb{R}^d$, which are now continuous variables. This is known as the spatio-temporal point process (Section 2.2.1) since $\boldsymbol{x}_i$ often correspond to locations of events, for example, earthquakes [195] or disease outbreaks [181]. The TPP models $p(\boldsymbol{t})$ the same way as in Section 5.2.2, so we are only left with the conditional density $p(\boldsymbol{X}|\boldsymbol{t})$. There are plenty of generative model choices to pick from (Section 2.2), but using the same reasoning as in Chapter 3 we opt for normalizing flows. In particular, they offer closed-form likelihood and allow for straightforward sampling. A natural candidate is a continuous normalizing flow [35] which defines the instantaneous change of variables (Equation 2.37) and the corresponding log-likelihood formulation (Equation 2.38). In the usual density estimation setting, the ODE integration boundaries are fixed, however, since we now have continuous data we could utilize the continuity to naturally evolve the distribution with time.

Chen et al. [34] propose several spatio-temporal models that use this idea, the first one being the time-varying CNF where $p(\boldsymbol{x}_i|t_i)$ is estimated by integrating Equation 2.38 from $t_0 = 0$ until the observed t_i:

$$\log p(\boldsymbol{x}_i|t_i) = \log q(\boldsymbol{z}(0)) - \int_0^{t_i} \operatorname{Tr}\left(\frac{\partial f}{\partial \boldsymbol{z}(t)}\right) \mathrm{d}t. \tag{5.8}$$

where $q(\boldsymbol{z}(0))$ is the base density learned with another normalizing flow to avoid trivial solutions at $t = 0$. An alternative model, attentive CNF [34], is more expressive compared to the time-varying CNF and more efficient than jump ODE models [118]. The density of $\boldsymbol{x}_i$ depends on all the previous values $\boldsymbol{x}_{j<i}$ through the attention mechanism [273].

We can keep the same kind of time dependence by using our notation for neural flows in the familiar discrete change of variables formula:[3]

$$p(\boldsymbol{x}_i|t_i) = q(F^{-1}(t_i, \boldsymbol{x}_i))|\det J_{F^{-1}}(\boldsymbol{x}_i)|. \tag{5.9}$$

[3]The term "discrete" here refers to the fact we do not use continuous normalizing flows. However, our proposed model is aware of continuous nature of the data and evolves the density with time.

This works because F is a diffeomorphism by definition and will define a valid distribution at any t_i. The only question is whether we can compute the inverse and the determinant with ease. This is simply satisfied if we use our coupling flow as defined in Equation 5.6. The determinant is the product of the diagonal values of the Jacobian with respect to $\boldsymbol{x}_i$, which are simply exponential terms from Equation 5.6. The difference between a RealNVP coupling model [59] and our implementation is that we include the time dependency meaning the density changes with time and is equal to the base density at $t = 0$. Unlike a CNF model, we do not use the solver or trace estimation to compute the likelihood. To generate new realizations at t_i, we first sample from q to get $\boldsymbol{x}_0 \sim q(\boldsymbol{x}_0)$, then simply evaluate $F(t_i, \boldsymbol{x}_0)$.

For the attentive version of our model, we represent all the previous points $\boldsymbol{x}_{j<i}$ with an attention encoder and define a conditional coupling NF $p(\boldsymbol{x}_i|t_i, \boldsymbol{x}_{j<i})$. We describe the full model in Appendix C.1.4. It is also possible to use ResNet flow, but the benefits over ODEs vanish since the determinant and the inverse require iterative procedure.

5.3 Experiments

In this section we show that flow-based models can match or outperform ODEs at a smaller computation cost, both in latent variable time series modeling, as well as TPPs and time-dependent density estimation. To make fair comparison, we used recently introduced reparameterization trick for ODEs that allows faster mini-batching [34], and the semi-norm trick for faster backpropagation [127], making the models more competitive compared to the original works. In all experiments we split the data into train, validation and test set; train with early stopping and report results on test set. We use Adam optimizer [134]. For training we use two different machines, one with 3.4GHz processor and 32GB RAM and another with 61GB RAM and NVIDIA Tesla V100 GPU 16GB [161]. All datasets are publicly available, we include the download links and release the code that reproduces the results.[4]

5.3.1 Synthetic data

We compare the performance of neural ODEs and neural flows on periodic signals and data generated from autonomous ODEs, in particular: sine, sawtooth, square and triangle signals, and sink and ellipse ODEs (Figure C.1). A detailed setup and results are presented in Appendix C.2. We observe that training with adaptive solvers [60] is slower compared to fixed-step solvers (Figure C.2), as expected. With the fixed step, however, we are not guaranteed invertibility [201], which can be an issue in, for example, density estimation (see Appendix C.2.1). Using the same setup, our models are an order of magnitude faster. Finally, as can be seen from Figure 5.2, neural ODEs struggle with non-smooth signals while neural flows perform much better, although they also only output smooth dynamics. Neural flows are also better at extrapolating, although none of the models excel here.

[4] `https://www.cs.cit.tum.de/daml/neural-flows/`

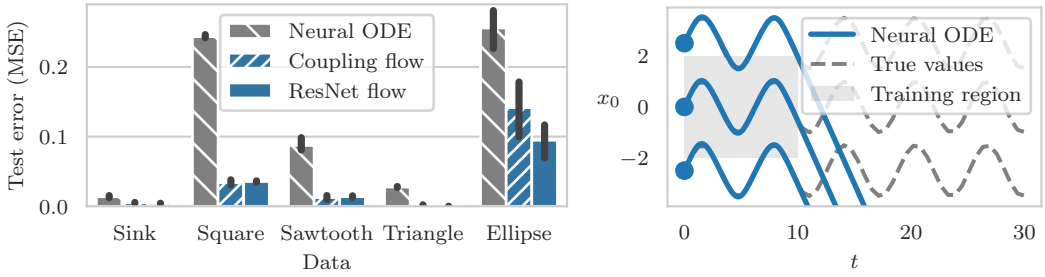


Figure 5.2: (Left) Test error for synthetic data. (Right) All models fail extrapolating in time.

5.3.2 Stiff ODEs

The numerical approach to solving ODEs is not only slow but it can be unstable. This can happen when the ODE becomes stiff, that is, when the solver needs to take very small steps even though the solution curve is smooth. For neural ODEs, it can happen that the target dynamic is known to be stiff or the latent dynamic becomes stiff during training.

To see the effects of this, we use the experiment from Ghosh et al. [80]. The ODE is given by: $\dot{x} = -1000x + 3000 - 2000e^{-t}$. We train a neural ODE model and a coupling flow to match the data, minimizing MSE. The data contains initial conditions and solutions, on small intervals with $t_2 - t_1 = 0.125$, $t \in [0, 15]$. The flow first finds the solution at $t_0 = 0$ and then solves for t_2 (Section 5.1). We evaluate on an extended time interval given $x_0 = 0$. Figure 5.3 shows that the neural ODE with an adaptive solver does not match the correct solution, due to its stiffness. In contrast, flow captures the solution correctly, as expected, since it does not use a numerical solver. The test MSE is lower compared to an ODE with temporal regularization [80].

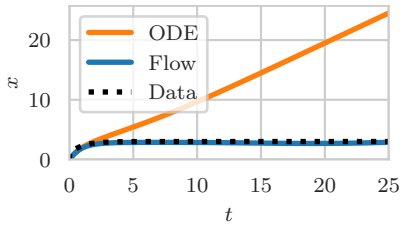


Figure 5.3: Flows handle stiffness better.

5.3.3 Smoothing approach

Following Rubanova et al. [229], we use three datasets: Activity, Physionet, and MuJoCo. Activity contains 6554 time series of 3d positions of 4 sensors attached to an individual. The goal is to classify one of the 7 possible activities (e.g., walking, lying, etc.). Physionet [245] contains 8000 time series and 37 features of patients' measurements from the first 48 hours after being admitted to ICU. The goal is to predict the mortality. MuJoCo is created from a simple physics simulation "Hopper" [263] by randomly sampling initial positions and velocities and letting dynamics evolve deterministically in time. There are 10000 sequences, with 100 time steps and 14 features.

We use the encoder-decoder model (Section 5.2.1) and maximize the ELBO (Equation 2.42). We use the same number of hidden layers and the same size of latent states

	MuJoCo	Activity		Physionet	
	MSE	MSE	Accuracy	MSE	AUC
Neural ODE	8.403±0.142	6.390±0.136	0.756±0.013	**4.833±0.078**	0.777±0.012
Coupling flow	**4.217±0.147**	6.579±0.049	0.752±0.012	4.860±0.070	**0.788±0.004**
ResNet flow	5.147±0.171	**6.279±0.098**	**0.760±0.004**	4.903±0.125	0.784±0.010

Table 5.1: Test mean squared error (lower is better) and accuracy/area under curve (higher is better). Best result is bolded, result within one standard deviation is underlined. Averaged over 5 runs.

	MIMIC-III		MIMIC-IV	
	MSE	NLL	MSE	NLL
GRU-ODE	0.507±0.005	**0.770±0.023**	0.379±0.005	0.748±0.045
ResNet flow	0.508±0.007	0.779±0.023	0.379±0.005	0.774±0.059
GRU flow	**0.499±0.004**	0.781±0.041	**0.364±0.008**	**0.734±0.054**

Table 5.2: Forecasting on healthcare data averaged over 5 runs (lower is better).

for both the neural ODE, coupling flow and ResNet flow, giving approximately the same number of trainable parameters. ODE models use either Euler or adaptive solvers and we report whichever has the better results in the end. The results in Table 5.1 show the reconstruction error and the accuracy of prediction. For better readability, we scale MSE scores the same way as in Rubanova et al. [229], by 10^2 for Activity and 10^3 for the rest.

Neural flows outperform ODE models everywhere, with Physionet reconstruction task being within the confidence interval. A further improvement of the results might be possible with bigger flow models but we focused on having similar sized models to show that we can get better results at a much smaller cost.

5.3.4 Speed improvements

In the smoothing experiment, our method offers more than two times speed-up during training compared to an ODE using an Euler method (Figure 5.4, different boxes corresponding to different datasets, grouped by experiment types). The gap is even larger for adaptive solvers. Note that Figure 5.4 shows an average time to run one training epoch which includes other operations, such as data fetching, state update etc. This shows that ODESolve contributes significantly to long training times. When comparing ODEs and flows alone, our method is much faster. In the following we will discuss the results from Figure 5.4 for other experiments as well as other results.

5.3.5 Filtering approach

Following De Brouwer et al. [51], we use clinical database MIMIC-III [120], pre-processed to contain 21250 patients' time series, with 96 features. We also process newly released MIMIC-IV [83, 119] to obtain 17874 patients. The details are in Appendix C.4.2. The

		MOOC		Reddit		Wiki	
	Discrete GRU	-0.4448	2.7563	-0.9299	1.8468	-0.5832	8.0527
Cont.	Jump ODE	0.8710	4.6118	0.1308	3.6654	-0.3115	10.6040
Cont.	Coupling flow	0.7694	5.5494	-0.1263	3.6312	-0.2807	9.7214
Cont.	ResNet flow	**-1.2379**	2.9466	**-1.2962**	2.3932	-1.2907	10.4368
Mix.	Jump ODE	-0.2626	3.0723	-1.0907	1.9057	**-1.3635**	**7.5537**
Mix.	Coupling flow	-0.4026	**2.5877**	-1.0933	**1.6817**	-1.2702	8.8018
Mix.	ResNet flow	-0.5664	3.0005	-1.0605	1.9491	-1.1937	8.5489

Table 5.3: Test NLL for TPP (left columns, per dataset) and marked TPP (right columns); full results in Appendix C.3. 'Cont.' denotes models with continuous intensity, and 'Mix.' with mixture distribution.

goal is to predict the next three measurements in the 12 hour interval after the observation window of 36 hours.

Table 5.2 shows that our GRU flow model (Equation 5.5) mostly outperforms GRU-ODE [51]. Additionally, we show that the ordinary ResNet flow with 4 stacked transformations (Equation 5.1) performs worse. The reason might be because it is missing GRU flow properties, such as boundedness. Similarly, an ODE with a regular neural network does not outperform GRU-ODE [51]. Finally, we report that the model with GRU flow requires 60% less time to run one training epoch.

5.3.6 Temporal point processes

As we saw in Section 5.2.2, most of the TPP models consist of two parts: the encoder that processes the history, and the network that outputs the intensity. In the context of neural ODEs, we would like to answer: 1) whether having continuous state $\boldsymbol{h}(t)$ outperforms RNNs, and 2) if intertwining the hidden state evolution with the intensity outperforms other approaches. For this purpose we propose the following models based on continuous intensity and mixture distributions.

Jump ODE evolves $\boldsymbol{h}(t)$ continuously together with the intensity function $\lambda(t) = g(\boldsymbol{h}(t))$ [118, 34], where g is a neural network. The neural flow version replaces an ODE with our proposed flow models to evolve $\boldsymbol{h}(t)$ and uses Monte Carlo integration to evaluate Equation 2.27. Note that this operation can be parallelized unlike solving an ODE.

The mixture-based models keep the same continuous time encoders (ODEs and flows) but output the stationary log-normal mixture for the next arrival time. That is, instead of outputting the continuous intensity, they only use the hidden state at the last observation to define the probability density function [240]. As a baseline, we use a discrete GRU with the same mixture decoder.

We use both synthetic and real-world data, following [199, 240]. We generate 4 synthetic datasets corresponding to homogeneous, renewal and self-correcting processes. For real-world data, we collect timestamps of forum posts (Reddit), interactions of students with an online course system (MOOC), and Wiki page edits [143]. The details of the data are in Appendix C.4.3.

	Bikes	Covid	EQ
Time-var. CNF	2.315	1.984	1.709
Attentive CNF	2.371	1.973	1.668
Time-var. coupling	**2.280**	**1.916**	1.633
Attentive coupling	2.330	1.926	**1.457**

Table 5.4: Test NLL for spatial datasets.

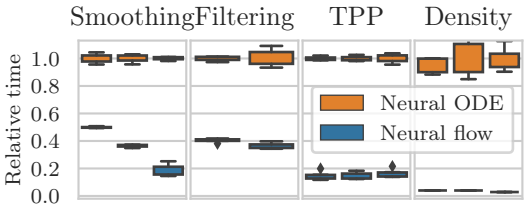


Figure 5.4: Comparing per-epoch wall-clock times. Each column represents a dataset (from left to right: order by appearance in text).

We report the test negative log-likelihood on real-world data in Table 5.3, for models trained both with and without marks. Full results, including synthetic data can be found in the Appendix C.3. We note that all the models capture the synthetic data, although continuous intensity models struggle compared to those with the mixture distribution. We can see this is the case for real-world data too, where the mixture distribution usually outperforms the corresponding continuous intensity model. In general, neural flows are better than ODE-based models, with the exception of one ODE model on Wiki dataset. We can conclude that having a continuous encoder is preferred to a discrete RNN because it can capture the irregular time sequence better. However, there is no benefit in parameterizing the intensity function in a continuous fashion, especially since this is a much slower approach.

Table C.4 in Appendix C.3 shows the comparison of wall clock times. Comparing only continuous intensity models we can see that Monte Carlo integration is faster than solving an ODE. As expected, using the mixture distribution gives the best performance. Thus, our flow models offer more than an order of magnitude faster processing compared to ODEs with continuous intensity. Figure 5.4 shows the difference for continuous models on the respective real-world datasets, the gap is even bigger if we include mixture-based models, where the speed-up is over an order of magnitude.

5.3.7 Spatial data

We compare the continuous normalizing flows with our continuous-time version of the coupling NF on time-dependent density estimation. We use two versions of each model: time-varying and attentive, as described in Section 5.2.3. Following Chen et al. [34], we use locations of bike rentals (Bikes), Covid cases for the state of New Jersey [266], and earthquake events in Japan (EQ) [270].

Results in Table 5.4 show the test NLL for spatial data, that is, we do not report the TPP loss since this is shared between models. Our continuous coupling NF models perform better on all datasets. Since affine coupling is a simple transformation, we require bigger models with more parameters. At the same time, our models are still more than an order of magnitude faster. Adapting some other, more expressive normalizing flows to satisfy flow constraints might reduce the number of parameters.

5.4 Discussion

In this work we presented neural flows as an efficient alternative to neural ODEs. We retain all the desirable properties of neural ODEs, without using numerical solvers. Our method outperforms the ODE based models in time series modeling and density estimation, at a much smaller computation cost. This brings the possibility to scale to larger datasets and models in the future.

A potential limitation of our approach is the fact that many ODEs do not have closed-form solutions, so we cannot always find the *exact* flow corresponding to a particular ODE. However, this is usually not an issue since in most applications, such as those presented in Section 5.2, it is sufficient for both neural ODEs and neural flows to approximate an unknown dynamic.

Additionally, since neural ODEs reuse the same function f in the solver, essentially defining *implicit layers*, they can be more parameter efficient. Sometimes we might need more parameters to represent the same dynamic when using neural flows, as we have observed in the density estimation task. But even here, the results show neural flows are more computationally efficient. In the restrictive setting with limited memory, we can resort to existing solutions such as Chen et al. [36].

5.4.1 Other related work

Early works on approximating the ODE solutions without numerical solvers used splines or radial basis functions [177, 159], or functions similar to modern ResNets [146]. More recently, [209] approximate the solution by minimizing the error of the solution points and of the boundary condition. Unlike these approaches, we do not approximate the solution to some given ODE but learn the solutions which corresponds to learning the unknown ODE. Also, our method guarantees that we always define a proper flow, as is required in certain applications.

ResNets were initially recognized as a discretization of dynamical systems [160, 277] and were used to tackle infinite depth [7, 165], stability [41, 96] and invertibility [30, 114]. We take a different approach and propose modified ResNets, among other, avoiding any iterative procedure. Viewing ResNets as a discretized dynamical system ultimately led to neural ODEs which introduce memory efficient backpropagation as one of the main features [70, 35]. Further, to combat solver inefficiency, many improvements have been proposed, such as adding regularization [71, 80, 125], improving training [79, 127, 302] and having faster inference [211].

5.4.2 Autonomous ODEs

Autonomous differential equations are defined with a vector field that is fixed in time $\dot{\boldsymbol{x}} = f(\boldsymbol{x}(t))$. Note that function f does not depend on time t like before. Therefore, the conditions i) and ii) from Section 5.1 are not enough to define the corresponding flow. To be precise, the flow F defines an autonomous ODE if it satisfies the additional condition:

3. $F(t_1 + t_2, \boldsymbol{x}_0) = F(t_2, F(t_1, \boldsymbol{x}_0))$,

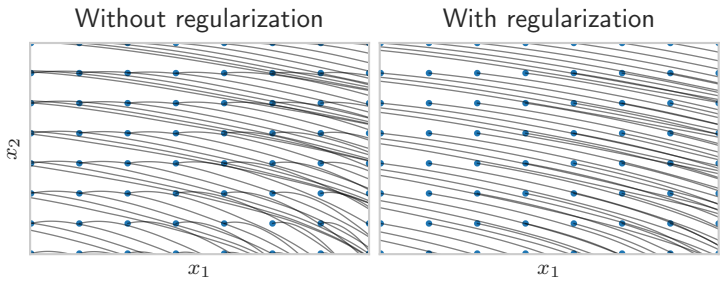


Figure 5.5: Comparison of the phase space for the same model trained with and without the autonomous regularization (Equation 5.10). Dots denote initial conditions. Note that the overlapping dynamic does not mean the solutions are not unique, only that the vector field is dependent on time.

meaning that solving for t_1 first, then t_2, is the same as solving for $t_1 + t_2$ once, given initial condition $\boldsymbol{x}_0$.

More formally, we defined flow F on set $\mathbb{R}^d$ as a group action of the additive group $G = (\mathbb{R}, +)$ (elements being time points). Equivalently, group action of G on $\mathbb{R}^d$ is a group homeomorphism from G to $\mathrm{Sym}(\mathbb{R}^d)$ (symmetric group, bijective functions and composition $(\phi, \circ)$), i.e., some function $\varphi : G \to \mathrm{Sym}(\mathbb{R}^d)$ maps time t to parameters of an invertible neural network ϕ, with $\varphi(t_1 + t_2) = \varphi(t_1) \circ \varphi(t_2)$. Identity element of G, 0 is mapped to an identity function, inverse $-t$ is mapped to an inverse function.

It's clear that our proposed architectures from Section 5.1 do not satisfy condition iii), unless we redefine it to allow time-dependence. Therefore, one way to satisfy iii) is to have $\frac{\mathrm{d}}{\mathrm{d}t}F$ independent of time. Note, however, that if we define the ResNet flow as $\boldsymbol{x}_t := F(t, \boldsymbol{x}_0) = \boldsymbol{x}_0 + t \cdot h(\boldsymbol{x}_0)$, then even though time disappears from the derivative $\frac{\mathrm{d}}{\mathrm{d}t}F$, the derivative is expressed in terms of $\boldsymbol{x}_0$, not $\boldsymbol{x}_t$. This means time is still implicitly included since starting at different $\boldsymbol{x}_0$ gives different values.

Matrix exponential $\exp(\boldsymbol{A}t)\boldsymbol{x}$, as a solution to a linear ODE: $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x}$, is one example of a closed-form solution to an autonomous ODE. Another potential *autonomous flow* is of the form $\boldsymbol{x} + \varphi(t)$, but not $g(\boldsymbol{x}) + \varphi(t)$, since this does not satisfy initial condition or g must depend on time. To the best of our knowledge, there is no general neural flow parametrization that can capture all autonomous ODEs. Therefore, we can try to approximate the desired behavior instead of guaranteeing it.

We can add a penalty to our loss that directly corresponds to condition iii). Given the loss function $\mathcal{L}$ and the current batch of n elements $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, $\boldsymbol{t} \in \mathbb{R}^n$, where we can represent each $t_i \in \boldsymbol{t}$ as $t_i = t_i^{(1)} + t_i^{(2)}$, with $t_i^{(1)}, t_i^{(2)}$ uniformly sampled on $[0, t_i]$, the total loss is:

$$\mathcal{L}_{\text{total}} = \mathcal{L} + \gamma \frac{1}{n} \sum_i (F(t_i, \boldsymbol{x}_i) - F(t_i^{(2)}, F(t_i^{(1)}, \boldsymbol{x}_i)))^2, \tag{5.10}$$

where γ is some positive value. The second term penalizes flows that do not satisfy iii), meaning we should get the flow that is closer to the underlying autonomous ODE. This can be calculated in parallel to other computations.

Figure 5.5 shows the comparison between learning the data generated from an autonomous ODE, using the regularization as defined in Equation 5.10 and without such regularization. We can see that the base model already learns good behavior but when we include the regularization, the trajectories overlap less frequently.

5.4.3 Modeling stochastic differential equations

Neural ODEs define how vector values change with a small time step. If we add some noise to that change, we arrive at stochastic differential equations, which are widely used in physics and mathematical finance [185, 197]. The SDE can be defined as follows:

$$\mathrm{d}\boldsymbol{x}_t = \mu(\boldsymbol{x}_t, t)\,\mathrm{d}t + \sigma(\boldsymbol{x}_t, t)\,\mathrm{d}W_t, \tag{5.11}$$

where $\mu : [0, T] \times \mathbb{R}^d \to \mathbb{R}^d$ is a drift function similar to ODE, and $\sigma : [0, T] \times \mathbb{R}^h \to \mathbb{R}^{h \times d}$ is a new term representing diffusion, with W_t being Brownian motion. The solution F to Equation 5.11 shares the same properties as the solution to an ODE, namely, it is a diffeomorphism, identity at $t = 0$, and has the group action property. The only difference is that here, F depends on the full Brownian path $w(\cdot)$,[5] which is why it is also known as the *stochastic* flow of diffeomorphisms [144, 145]. There are two main challenges:

1. **Embedding the Brownian path.** The continuous path w is *infinite dimensional* and we cannot directly use it as an input. A simple way to get around this issue is to discretize it and use the sampled points $W_1, \ldots, W_n \in w(\cdot)$ to get a unique fixed-size representation. One idea is to use some non-parametric embedding such as a signature transform [37, 129] which is, conveniently, invariant to the number of the samples and their positions.

2. **Learning.** Given a general stochastic flow model $F_{t,w}$, that is, a flow that depends on time and Brownian path, we are usually not able to estimate the likelihood efficiently. We can adopt viewing the SDE learning as equivalent to training a GAN [128]. If we want likelihood-based method we can, for example, adopt the normalizing flows to transform the initial samples (Brownian path) with a diffeomorphism to the target distribution, similar to Deng et al. [56].

An alternative is to only model the distribution of the SDE solutions, which is also known as the Fokker-Planck PDE [72, 210]. The limitation of this approach is that we cannot distinguish different stationary processes, that is, those with the constant marginal distribution. However, since we are learning with exact likelihood, the training is significantly simplified. This setting is the most similar to Section 5.2.3.

We [18] evaluate different proposed approaches and show that it is feasible to model SDEs with a stochastic flow. However, since the training is the same as in the neural SDE literature, it can suffer from various issues, like underestimating the diffusion part of the SDE. In Chapter 6 we propose an alternative for modeling SDEs, and continuous functions in general, based on generative diffusion that mitigates these issues.

[5] More precisely, when evaluating at a time point t we use the path on $[0, t]$.

5.4.4 Modeling partial differential equations

In many cases, the dynamical system is governed by a partial differential equation:

$$\partial_t u = f(t, \boldsymbol{x}, u, \partial_{\boldsymbol{x}} u, \partial_{\boldsymbol{xx}} u, \dots), \tag{5.12}$$

where u is the solution, and we are given the initial condition $u_0(\boldsymbol{x})$ at $t = 0$. The ∂ notation is used to represent the partial derivatives. PDEs are usually defined on some bounded domain so we get an additional boundary condition, for example, a so-called Dirichlet condition $u_{\mathcal{B}}(t)$ specifies the values on the boundary $\mathcal{B}$.

We want to model the solution u with an initial value model F, a neural network that satisfies both initial and boundary conditions. One way to model the solutions of PDEs is with physics informed neural networks (PINNs) by using the known structure of the Equation 5.12 to achieve a more efficient and accurate learning [215]. The boundary and initial condition are not guaranteed to be satisfied but the model learns them through additional terms in the loss function.

We can extend this by implementing the initial condition the same way as before and additionally have to ensure the model satisfies the boundary condition. Since these conditions can differ, they would have to be implemented on a case-by-case basis. In [18] we show how one can specify the condition for a periodic boundary. Note that we do not need the invertibility anymore since PDE solutions are not always unique.

Another neural PDE method is simply learning the solution based on the previous observations, that is, the model is an encoder-decoder which inputs one or more previous time steps and outputs the prediction for one or more future time steps. When such a model is also resolution invariant, it is referred to as a neural operator, since it learns mappings between function spaces. The approach can be written as [158, 25]:

$$F(t + \Delta t) = \mathcal{A}(F(t), \Delta t), \tag{5.13}$$

where existing methods usually fix Δt. We do not want to discretize the time domain and, additionally, we want to satisfy the initial condition at $\Delta t = 0$. Further, our model will now have the ability to predict at multiple time points at once with unlimited time horizon. In [18] we extend Fourier neural operators [158] to include initial condition. The constrained architecture outperforms the baseline on two synthetic datasets [18].

Again, we can conclude that introducing additional, informative constraints on the network removes the burden of learning the constraints from the network and allows it to focus on optimizing the task at hand.

6 Denoising Diffusion for Functions

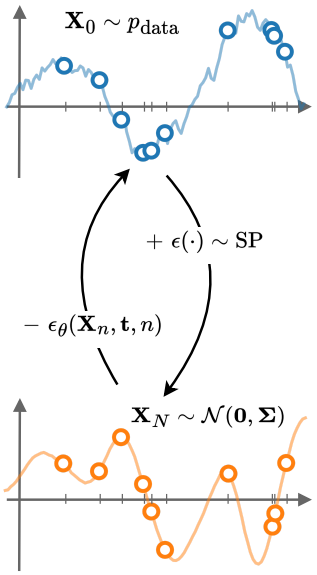


Figure 6.1: Data, represented with blue points in the top figure, assumed to follow some unknown continuous function (blue curve). The diffusion is defined as the forward process in which we add noise from a stochastic process (SP) to the *whole* time series to get pure noise (bottom figure). The model $\boldsymbol{\epsilon}_\theta$ learns to reverse this process which ultimately allows generating new data.

Time series data is collected from measurements of some real-world system that evolves via some complex unknown dynamics. In Chapters 3 and 4, we have seen how to model event data—things that occur sparingly. In Chapter 5 we encountered continuously measured data, such as that recorded from a dynamical system. In this chapter we will only consider the latter case and propose a generative model for continuous data that is based on the diffusion framework (Section 2.2.4). In particular, we assume time series follows some underlying continuous function and, as a consequence, our generative model will generate functions. Therefore, we make a natural connection to neural processes—a way to model stochastic processes with neural networks.

As we have seen in the previous chapter, different approaches for modeling continuous data have been proposed, from neural (ordinary or stochastic) differential equations [35, 156], neural flows (Chapter 5) to normalizing flows [56] and neural processes [76]. As it turns out, capturing the true generative process proves difficult, especially with the inherent stochasticity of the data. Take neural SDEs as an example. If one trains them with a VAE formulation [156], the term corresponding to the noise is often underestimated, while the same does not happen if they are trained with an adversarial loss [128]; however, the latter inherits all training issues that GANs observe. Our goal is to have a model that is easy to train and produces correct samples. We will see that the generative diffusion framework is exactly what we are looking for.

Recently, denoising diffusion models have shown great promise in modeling very complicated data distributions such as those in the image domain [102, 252]. The approach consists of first adding the noise to data gradually until it becomes pure noise, corresponding to some base distributions. At the same time, the model is trained to reverse this process. To generate a new data point, we start with an initial noisy value sampled from the base distribution; then the model gradually denoises it to reach a sample from the learned data distribution. The general idea behind diffusion is covered in Section 2.2.4 and the implementations we base our approach on are defined more rigorously in Section 6.1.

In this work, we expand on this general framework to define the diffusion for data measured in continuous time by treating it as a discretization of some continuous function. That is, instead of adding noise to each data point independently, we add the noise to the whole function while preserving its continuity. In Section 6.2, we show that this can be done by using stochastic processes as noise generators. We additionally show that the final noisy function will also correspond to a sample from a known stochastic process. Next, we specify the transition probabilities in the forward noising process, the evidence bound on the likelihood used in the training, and the new sampling procedure, for both the fixed-step and SDE-based diffusion approaches.

Figure 6.1 shows an illustration of our approach. Data is observed as a set of (irregularly-sampled) points that correspond to some underlying function. By adding noise to this function we reach the prior stochastic process. At the same time, the model can reverse this process, allowing us to generate new function samples.

In Section 6.3 we describe different use cases that we can tackle with our model while highlighting the benefits over previous approaches. For example, we can use conditioning to output the distribution over future values, that is, we use our method for multivariate probabilistic forecasting. Since we define the distribution over functions we can also view our model as a neural process [76], allowing us to estimate missing points from the observed data. In Section 6.4 we empirically show that our model outperforms the baselines on these tasks and others.

6.1 Background

Given training data $\{\boldsymbol{x}_i\}$, with $\boldsymbol{x}_i \in \mathbb{R}^d$, the goal of generative modeling is to learn the probability density function $p(\boldsymbol{x})$ and be able to generate new samples from this learned distribution. Diffusion models achieve both of these goals by learning to reverse some fixed process that adds noise to the data. In the following, we present a brief overview of the two ways to define diffusion; in Section 6.1.1 the noise is added across N increasing scales [102], which is then taken to the limit in Section 6.1.2 using a stochastic differential equation (SDE) [252].

6.1.1 Fixed-step diffusion

Sohl-Dickstein et al. [248] and Ho et al. [102] propose the denoising diffusion probabilistic model (DDPM) which gradually adds *fixed* Gaussian noise to the observed data point $\boldsymbol{x}_0$ via known scales β_n to define a sequence of progressively noisier values $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$.

The final noisy output $\boldsymbol{x}_N \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ carries no information about the original data point. The sequence of positive noise (variance) scales $\beta_1, \ldots, \beta_N$ has to be increasing such that the first noisy output $\boldsymbol{x}_1$ is close to the original data $\boldsymbol{x}_0$, and the final value $\boldsymbol{x}_N$ is pure noise. The goal is then to learn to reverse this process.

As diffusion forms a Markov chain, the transition between any two consecutive points is defined with a conditional probability $q(\boldsymbol{x}_n|\boldsymbol{x}_{n-1}) = \mathcal{N}(\sqrt{1-\beta_n}\boldsymbol{x}_{n-1}, \beta_n \boldsymbol{I})$. Since the transition kernel is Gaussian, the value at any step n can be sampled directly from $\boldsymbol{x}_0$. Let $\alpha_n = 1 - \beta_n$ and $\bar{\alpha}_n = \prod_{k=1}^{n} \alpha_k$, then we can write:

$$q(\boldsymbol{x}_n|\boldsymbol{x}_0) = \mathcal{N}(\sqrt{\bar{\alpha}_n}\boldsymbol{x}_0, (1-\bar{\alpha}_n)\boldsymbol{I}). \tag{6.1}$$

Further, we can derive a posterior probability of any intermediate value $\boldsymbol{x}_{n-1}$ given its successor $\boldsymbol{x}_n$ and initial $\boldsymbol{x}_0$ as:

$$q(\boldsymbol{x}_{n-1}|\boldsymbol{x}_n, \boldsymbol{x}_0) = \mathcal{N}(\tilde{\boldsymbol{\mu}}_n, \tilde{\beta}_n \boldsymbol{I}), \tag{6.2}$$

where:

$$\tilde{\boldsymbol{\mu}}_n = \frac{\sqrt{\bar{\alpha}_{n-1}}\beta_n}{1-\bar{\alpha}_n}\boldsymbol{x}_0 + \frac{\sqrt{\alpha_n}(1-\bar{\alpha}_{n-1})}{1-\bar{\alpha}_n}\boldsymbol{x}_n,$$
$$\tilde{\beta}_n = \frac{1-\bar{\alpha}_{n-1}}{1-\bar{\alpha}_n}\beta_n.$$

The generative model learns the reverse process. To this end, Sohl-Dickstein et al. [248] set $p(\boldsymbol{x}_{n-1}|\boldsymbol{x}_n) = \mathcal{N}(\boldsymbol{\mu}_\theta(\boldsymbol{x}_n, n), \beta_n \boldsymbol{I})$, and parameterized $\boldsymbol{\mu}_\theta$ with a neural network. The training objective is to maximize the evidence lower bound:

$$\begin{aligned}\log p(\boldsymbol{x}_0) \geq \mathbb{E}_q\left[\log p(\boldsymbol{x}_0|\boldsymbol{x}_1)\right] - D_{\mathrm{KL}}(q(\boldsymbol{x}_N|\boldsymbol{x}_0)\|p(\boldsymbol{x}_N)) \\ - \sum_{n>1} D_{\mathrm{KL}}(q(\boldsymbol{x}_{n-1}|\boldsymbol{x}_n, \boldsymbol{x}_0)\|p(\boldsymbol{x}_{n-1}|\boldsymbol{x}_n)).\end{aligned} \tag{6.3}$$

The objective boils down to minimizing the KL-divergence between the known posterior distribution $q(\boldsymbol{x}_{n-1}|\boldsymbol{x}_n, \boldsymbol{x}_0)$ and the learned reverse transition distribution $p(\boldsymbol{x}_{n-1}|\boldsymbol{x}_n)$. In practice, however, the approach by Ho et al. [102] is to reparameterize $\boldsymbol{\mu}_\theta$ and predict the noise $\boldsymbol{\epsilon}$ that was added to $\boldsymbol{x}_0$, using a neural network $\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_n, n)$, and minimize a simplified, yet equivalent loss function:

$$\mathcal{L} = \mathbb{E}_{\boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\boldsymbol{I})}\mathbb{E}_{n\sim\mathcal{U}(\{0,\ldots,N\})}\left[\|\boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_n}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_n}\boldsymbol{\epsilon}, n) - \boldsymbol{\epsilon}\|_2^2\right], \tag{6.4}$$

To generate new data, the first step is to sample a point from the final distribution $\boldsymbol{x}_N \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ and then iteratively denoise it using the above model ($\boldsymbol{x}_N \mapsto \boldsymbol{x}_{N-1} \mapsto \cdots \mapsto \boldsymbol{x}_0$) to get a sample from the data distribution, using Langevin sampling from Section 2.2.4. To summarize, the forward process adds the noise $\boldsymbol{\epsilon}$ to the input $\boldsymbol{x}_0$, at different scales, to produce $\boldsymbol{x}_n$. The model inverts this, that is, predicts $\boldsymbol{\epsilon}$ from $\boldsymbol{x}_n$.

6.1.2 Score-based SDE diffusion

Instead of taking a finite number of diffusion steps as in Section 6.1.1, Song et al. [252] introduce a continuous diffusion of vector valued data, $\boldsymbol{x}_0 \mapsto \boldsymbol{x}_s$ where $s \in [0, S]$ is now a continuous variable. We repeat the forward and reverse SDE equations from Section 2.2.4 using s for the diffusion step instead of t to avoid confusing it with the time of observation:

$$\mathrm{d}\boldsymbol{x}_s = f(\boldsymbol{x}_s, s)\,\mathrm{d}s + g(s)\,\mathrm{d}W_s, \tag{2.47}$$

$$\mathrm{d}\boldsymbol{x}_s = [f(\boldsymbol{x}_s, s) - g(s)^2 \nabla_{\boldsymbol{x}_s} \log p(\boldsymbol{x}_s)]\,\mathrm{d}s + g(s)\,\mathrm{d}W_s. \tag{2.48}$$

Solving the above SDE from S to 0, given initial condition $\boldsymbol{x}_S \sim p(\boldsymbol{x}_S)$, returns a sample from the data distribution. The generative model's goal is to learn the score function via a neural network $\psi_{\boldsymbol{\theta}}(\boldsymbol{x}_s, s)$, by minimizing:

$$\mathcal{L} = \mathbb{E}_{\boldsymbol{x}_s \sim \mathrm{SDESolve}(\mathrm{d}\boldsymbol{x}_t, \boldsymbol{x}_0, 0, s)} \mathbb{E}_{s \sim \mathcal{U}(0,S)} \left[\| \psi_{\boldsymbol{\theta}}(\boldsymbol{x}_s, s) - \nabla_{\boldsymbol{x}_s} \log p(\boldsymbol{x}_s) \|_2^2 \right], \tag{6.5}$$

where the first expectation is over SDE solutions of Equation 2.47 at time s given initial $\boldsymbol{x}_0$. Song et al. [252] define a continuous analogue to DDPM forward process (Section 6.1.1) as the following SDE:

$$\mathrm{d}\boldsymbol{x}_s = -\frac{1}{2}\beta(s)\boldsymbol{x}_s\,\mathrm{d}s + \sqrt{\beta(s)}\,\mathrm{d}W_s, \tag{6.6}$$

where $\beta(s)$ and S are chosen in such a way that ensures the final noise distribution is unit normal, $\boldsymbol{x}_S \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. Since the variance is constant in the end, this kind of diffusion is named variance preserving. There are other potential parameterizations, such as using a variance exploding process or modifying Equation 6.6 to perform better on likelihoods [252]. Given a specific parameterization, one can easily derive the transition probability $q(\boldsymbol{x}_s|\boldsymbol{x}_0)$ and calculate the exact score in closed-form (see Section 6.2.3 and Appendix D.1.3). This allows trivial loss computation as in Equation 6.5 compared to a more complicated Equation 2.44 from Section 2.2.4.

6.1.3 Extensions

Generative modeling with diffusion recently gained a lot of traction as it provides good sampling quality synthesizing images [57, 216, 228], replacing GANs [85] as the state-of-the-art method. The modeling power translates to other tasks as well, with successful applications in modeling waveforms [139] and time series forecasting [222], but also generating discrete data such as text [3] and molecules [1, 151].

Many of the advances over the original diffusion focused on improving the sampling speed [39, 121, 167], while others implement the noise scheduling for better modeling capacity [192, 133]. Although this particular area of research is orthogonal to the ideas we present in this chapter, we can easily implement these techniques to make our method perform faster or have better sampling quality. That is, since we later show that modeling continuous functions with diffusion resembles the original formulation, any improvement in diffusion modeling can be applied to our method as well.

6.2 Method

In contrast to the previous section which explicitly deals with data points that are represented by vectors or sequences of vectors, we are interested in generative modeling of the underlying continuous function. We still represent the data as a time-indexed sequence of points observed across M timestamps: $\boldsymbol{X} = (\boldsymbol{x}(t_0), \ldots, \boldsymbol{x}(t_{M-1}))$, $t_i \in \boldsymbol{t} \subset [0, T]$. The observations can be equally spaced but this formulation encompasses irregularly-sampled data as well. We assume that each observed time series comes from its corresponding underlying continuous function $\boldsymbol{x}(\cdot)$.

Our approach can be viewed as modeling the distribution "$p(\boldsymbol{x}(\cdot))$" over functions instead of vectors, which amounts to learning the stochastic process. We review stochastic processes in more detail in Section 6.3.2. To preserve continuity, we cannot apply the ideas from Section 6.1 directly, unless we assume measurements are independent of each other. A direct consequence of adding an independent noise in the forward and reverse process is that it produces discontinuous samples, which is at odds with our assumption.

6.2.1 Stochastic processes as noise sources for diffusion

Instead of defining the diffusion by adding some scaled noise vector $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ to a data vector $\boldsymbol{x}$, we add a noise *function* (stochastic process) $\boldsymbol{\epsilon}(\cdot)$ to the underlying data function $\boldsymbol{x}(\cdot)$. The only restriction on $\boldsymbol{\epsilon}(\cdot)$ is that it has to be continuous so that the output remains continuous as well, which clearly rules out stochastic processes where time is indexed by a *finite* set, e.g., $\boldsymbol{\epsilon}(\boldsymbol{t}) \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$. However, using a normal distribution proved to be very convenient in Section 6.1 as it allowed for closed-form formulations of various terms, especially the loss. This is mostly due to the nice properties of the Gaussian distribution.

Therefore, our goal is to define $\boldsymbol{\epsilon}(\cdot)$ which will satisfy the continuity property while giving us tractable training and sampling. Note that t refers to the time of the observation and $\boldsymbol{\epsilon}(t) \in \mathbb{R}^d$ is the noise at t and $\boldsymbol{\epsilon}(\boldsymbol{t}) \in \mathbb{R}^{M \times d}$ is noise added to a time series with M observations. In contrast, in previous section we had *time-like* variables n and s that refer to the noise scale, not arrival time.

We could consider obtaining the noise from a standard Wiener process $\boldsymbol{\epsilon}(t) = W_t$. A clear disadvantage of this approach is that variance grows with time. Additionally, the distribution of W_0 is degenerate as we never add any noise. This can be solved in an ad hoc manner by shifting the whole time series similar to Deng et al. [56].

Instead, in the following, we present two *stationary* stochastic processes that add the same amount of noise regardless of the time of the observation. Note that the noise is *correlated* in the time dimension, hence the use of the stochastic process. An additional nice property of these processes is that they reduce to the diffusion from Section 6.1 in the trivial case of time series with only one element.

Let us shortly restrict the discussion to univariate time series $\boldsymbol{X} \in \mathbb{R}^M$ and producing noise $\boldsymbol{\epsilon}(\boldsymbol{t}) \in \mathbb{R}^M$. We present the general approach at the end of this section.

A) Gaussian process prior. Given a set of M time points $\boldsymbol{t}$, we propose sampling $\boldsymbol{\epsilon}(\boldsymbol{t})$ from a Gaussian process $\mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$, where each element of the covariance matrix is specified with a kernel $\boldsymbol{\Sigma}_{ij} = k(t_i, t_j)$, where $t_i, t_j \in \boldsymbol{t}$. This produces *smooth* noise functions $\boldsymbol{\epsilon}(\cdot)$ that can be evaluated at any point t, even outside given $\boldsymbol{t}$.

To define a stationary process, we have to use a stationary kernel; we will use a radial basis function $k(t_i, t_j) = \exp(-\gamma(t_i - t_j)^2)$. Adjusting the parameter γ lets us vary the flatness of the noise curves, higher values give noisier curves. Given a set of time points $\boldsymbol{t}$, we can easily sample from this process by first computing the covariance $\boldsymbol{\Sigma}(\boldsymbol{t})$ and then sample from the multivariate normal distribution $\mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$.

B) Ornstein-Uhlenbeck diffusion. An alternative noise distribution is a stationary OU process that is specified as a solution to the following SDE:

$$\mathrm{d}\epsilon_t = -\gamma\epsilon_t \,\mathrm{d}t + \mathrm{d}W_t, \tag{6.7}$$

where W_t is the standard Wiener process and we use the initial condition $\epsilon_0 \sim \mathcal{N}(0, 1)$. We can obtain samples from OU process easily by sampling from a time-changed and scaled Wiener process: $\exp(-\gamma t)W_{\exp(2\gamma t)}$. The covariance can be calculated as $\boldsymbol{\Sigma}_{ij} = \exp(-\gamma|t_i - t_j|)$. The OU process is a special case of a Gaussian process with a Matérn kernel ($\nu = 0.5$) [219, p. 86]. We discuss different sampling techniques and their trade-offs in Appendix D.1.4.

In the end, both the GP and OU processes are defined with a multivariate normal distribution over a finite collection of points, where the covariance is calculated using the times of the observations. As opposed to the methods from Section 6.1, we use correlated noise in the forward process. Our approach allows us to produce continuous functions as samples and will prove to be a natural way to do forecasting and imputation.

Multivariate time series. In our work, we consider multivariate time series which means we record measurements coming from d univariate time series. Alternatively, we can view this as observing a d-dimensional vector and its evolution over time. In the forward diffusion process, we treat the data as d individual univariate time series and add the noise to them independently. In case of noise coming from a Gaussian process, this is equivalent to using block-diagonal covariance matrix of size $(Md) \times (Md)$ with $\boldsymbol{\Sigma}$ repeated on the diagonal. This is in line with the previous works where, for example, independent noise is added to individual pixels in an image.

Note that this does not mean we do not model correlations between dimensions. As we will see in the following section, the reverse process takes a complete multivariate time series and captures these correlations. This is again similar to image synthesis—although forward process is independent over pixels, the reverse process captures the whole image. The difference in our approach is that we also enforce the continuity across the time dimension, which means our model is *guaranteed* to produce continuous samples.

Algorithm 1 Training (DSPD-GP diffusion)

while not converged **do**
  $\boldsymbol{X}_0, \boldsymbol{t} \sim p_{\text{data}}(\boldsymbol{X}, \boldsymbol{t})$
  $\boldsymbol{\Sigma} = k(\boldsymbol{t}, \boldsymbol{t}^T)$
  $\boldsymbol{L} = \text{Cholesky}(\boldsymbol{\Sigma})$
  $\tilde{\boldsymbol{\epsilon}} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$
  $n \sim \mathcal{U}(\{1, \dots, N\})$
  $\boldsymbol{X}_n = \sqrt{\alpha_n}\boldsymbol{X}_0 + \sqrt{1-\alpha_n}\boldsymbol{L}\tilde{\boldsymbol{\epsilon}}$
  Take gradient step on
    $\nabla_\theta ||\tilde{\boldsymbol{\epsilon}} - \boldsymbol{\epsilon}_\theta(\boldsymbol{X}_n, \boldsymbol{t}, n)||_2^2$
end while

6.2.2 Discrete stochastic process diffusion (DSPD)

We apply the discrete diffusion framework to the time series setting. Note, *discrete* refers to the number of diffusion steps (Section 6.1.1), that is, we still model continuous functions. Reusing the notation from before, $\boldsymbol{X}_0$ denotes the input data and $\boldsymbol{X}_n = (\boldsymbol{x}_n(t_0), \dots, \boldsymbol{x}_n(t_{M-1}))$ is the noisy output after n diffusion steps. In contrast to the classical DDPM [102] where one adds independent Gaussian noise to data, we now add the noise from a stochastic process. In particular, given the times of the observations, we can compute the covariance $\boldsymbol{\Sigma}$ and sample noise $\boldsymbol{\epsilon}(\cdot)$ from a GP or OU process as defined in Section 6.2.1. We write the transition kernel and the posterior as:

$$q(\boldsymbol{X}_n|\boldsymbol{X}_0) = \mathcal{N}(\sqrt{\bar{\alpha}_n}\boldsymbol{X}_0, (1-\bar{\alpha}_n)\boldsymbol{\Sigma}), \tag{6.8}$$

$$q(\boldsymbol{X}_{n-1}|\boldsymbol{X}_n, \boldsymbol{X}_0) = \mathcal{N}(\tilde{\boldsymbol{\mu}}_n, \tilde{\beta}_n\boldsymbol{\Sigma}), \tag{6.9}$$

where $\tilde{\boldsymbol{\mu}}_n, \tilde{\beta}_n$ are the same as in Equation 6.2 (full derivation is in Appendix D.1.1). Even though we are now able to sample functions instead of a single point, the distributions are still similar to the previous case, with the main change occurring in the covariance. This nice result will be useful later to define the loss which is analogous to Equation 6.4.

We define a generative model $p(\boldsymbol{X}_{n-1}|\boldsymbol{X}_n) = \mathcal{N}(\boldsymbol{\mu}_\theta(\boldsymbol{X}_n, \boldsymbol{t}, n), \beta_n\boldsymbol{\Sigma})$ as a reverse process, similar to Ho et al. [102] while keeping our time-dependent covariance $\boldsymbol{\Sigma}$. The key difference is that the model now takes the full time series consisting of noisy observations $\boldsymbol{X}_n$ with their timestamps $\boldsymbol{t}$ in order to predict the noise $\boldsymbol{\epsilon}$ which has the same size as $\boldsymbol{X}_n$. The architecture, therefore, has to be a type of a time series encoder-decoder.

Since all the distributions that appear in the ELBO (Equation 6.3) are now multivariate normal, the loss can be calculated in closed-form. In Appendix D.1.2 we show the full derivation. Further, we show how we can reparameterize the model such that the covariance $\boldsymbol{\Sigma}$ disappears from the final loss. In particular, if our model predicts the noise that was added to the original data we can simplify the loss to only compute the squared difference between the predicted and true noise, similar to Equation 6.4:

$$\mathcal{L} = \mathbb{E}_{\boldsymbol{\epsilon},n}\left[\|\boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_n}\boldsymbol{X}_0 + \sqrt{1-\bar{\alpha}_n}\boldsymbol{\epsilon}, \boldsymbol{t}, n) - \boldsymbol{\epsilon}\|_2^2\right]. \tag{6.10}$$

Algorithm 2 Sampling (DSPD-GP diffusion)

input: $\boldsymbol{t} = (t_0, \ldots, t_{M-1})$
$\boldsymbol{\Sigma} = k(\boldsymbol{t}, \boldsymbol{t}^T)$
$\boldsymbol{L} = \text{Cholesky}(\boldsymbol{\Sigma})$
$\boldsymbol{X}_N \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{\Sigma})$
for $n = N, \ldots, 1$ **do**
    $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{\Sigma})$
    $\boldsymbol{X}_{n-1} = \frac{1}{\sqrt{\alpha_n}} \left(\boldsymbol{X}_n - \frac{1-\alpha_n}{\sqrt{1-\bar{\alpha}_n}} \boldsymbol{L}\boldsymbol{\epsilon}_\theta(\boldsymbol{X}_n, \boldsymbol{t}, n) \right) + \beta_n \boldsymbol{z}$
end for
return $\boldsymbol{X}_0$

Finally, in order to sample, the initial noise has to come from a stochastic process instead of an independent normal distribution. The same is the case for the noise that is used in the intermediate steps of the Langevin dynamics.

In Algorithms 1 and 2 we provide the pseudocode for training the model and sampling new data, when using DSPD-GP model. Analogously for OU, we would replace the noise source using the third sampling option from Appendix D.1.4. For the score-based model we compute the mean squared error between the predicted and true conditional score function where the sampling procedure uses either ODE or SDE solver, just as in [252].

6.2.3 Continuous stochastic process diffusion (CSPD)

Similarly to the previous section, we can extend the continuous diffusion framework to use the noise coming from a Gaussian or OU process. Now, the noise scaling $\beta(s)$ is continuous in the diffusion time s, see Section 6.1.2. Given a factorized covariance matrix $\boldsymbol{\Sigma} = \boldsymbol{L}\boldsymbol{L}^T$, we modify the variance preserving diffusion SDE [252]:

$$\mathrm{d}\boldsymbol{X}_s = -\frac{1}{2}\beta(s)\boldsymbol{X}_s \,\mathrm{d}s + \sqrt{\beta(s)}\boldsymbol{L}\,\mathrm{d}W_s, \tag{6.11}$$

which gives us the following transition probability:

$$\begin{aligned} q(\boldsymbol{X}_s|\boldsymbol{X}_0) &= \mathcal{N}(\tilde{\boldsymbol{\mu}}, \tilde{\boldsymbol{\Sigma}}) \\ &= \mathcal{N}\left(\boldsymbol{X}_0 e^{-\frac{1}{2}\int_0^s \beta(s)\,\mathrm{d}s}, \boldsymbol{\Sigma}\left(1 - e^{-\int_0^s \beta(s)\,\mathrm{d}s}\right)\right). \end{aligned} \tag{6.12}$$

This result is derived using Equation 5.51 from Särkkä and Solin [234], with an analogous result for non time series data in Song et al. [252]. We discuss this in more detail in Appendix D.1.3. Since this probability is normal, the value of the score function can be computed in closed-form:

$$\nabla_{\boldsymbol{X}_s} \log q(\boldsymbol{X}_s|\boldsymbol{X}_0) = -\tilde{\boldsymbol{\Sigma}}^{-1}(\boldsymbol{X}_s - \tilde{\boldsymbol{\mu}}), \tag{6.13}$$

which we can use to optimize the same objective as in Equation 6.5. Our neural network $\boldsymbol{\epsilon}_\theta(\boldsymbol{X}_s, \boldsymbol{t}, s)$ will take in the full time series, together with the observation times $\boldsymbol{t}$ and the

diffusion time s, and predict the values of the score function. As it turns out, we can again use the reparameterization in which we predict the noise, whilst the score is only calculated when sampling new realizations. That is, since the model predicts $\tilde{\boldsymbol{\epsilon}}_{\boldsymbol{\theta}}$ which is assumed to come from an isotropic normal, we have to multiply it by $\boldsymbol{L}$ to get the noise prediction from a stochastic process. Equation 6.13 then reduces to $-\tilde{\boldsymbol{\Sigma}}^{-1}\sigma\boldsymbol{L}\tilde{\boldsymbol{\epsilon}}_{\boldsymbol{\theta}}$, where $\sigma^2 = 1 - \exp(-\int_0^s \beta(s)\,\mathrm{d}s)$ (Equation 6.12). However, since $\boldsymbol{\Sigma} = \boldsymbol{L}\boldsymbol{L}^T$ and $\tilde{\boldsymbol{\Sigma}} = \sigma^2\boldsymbol{\Sigma}$, we get that the predicted score is simply $(\boldsymbol{L}^T)^{-1}\tilde{\boldsymbol{\epsilon}}_{\boldsymbol{\theta}}/\sigma$ which is easy to evaluate and does not require explicitly inverting $\boldsymbol{L}^T$ as it is a triangular matrix.

6.3 Applications

To train a generative model, it must learn to reverse the forward diffusion process by predicting the noise that was added to the clean data. The input to the model is the time series $(\boldsymbol{X}_0, \boldsymbol{t})$ along with the diffusion step n or diffusion time s, and the output is of the same size as $\boldsymbol{X}_0$. If additional inputs are available, we can also model the conditional distribution; for example, we often have covariates for each time point of $\boldsymbol{t}$. We can also condition the generation on the past observations which essentially defines a probabilistic forecaster (Section 6.3.1) or condition only on the observed values which defines a neural process (Section 6.3.2) or an imputation model (Section 6.3.3).

6.3.1 Forecasting multivariate time series

Forecasting is answering what is going to happen, given what we have seen, and as such is the most prominent task in time series analysis. Probabilistic forecasting adds the layer of (aleatoric) uncertainty on top of that and returns the confidence intervals which is often a requirement for deploying models in real-world settings. Neural forecasters are usually encoder-decoders, where the history of observations $(\boldsymbol{X}^H, \boldsymbol{t}^H)$ is represented with a single vector $\boldsymbol{z}$ and the decoder outputs the distribution of the future values $\boldsymbol{X}^F$ given $\boldsymbol{z}$ at time points $\boldsymbol{t}^F$. Previous works relied on specifying the parameters of the output distribution, for example, via a diagonal covariance [232] or some low-rank approximation [231], relying on normalizing flows [13, 223], or GANs [140].

Recently, Rasul et al. [222] introduced a diffusion based forecasting model to learn the conditional probability $p(\boldsymbol{X}^F|\boldsymbol{X}^H)$. In particular, let $\boldsymbol{X}^H = (\boldsymbol{x}(t_0), \ldots, \boldsymbol{x}(t_{M-1}))$ be a history window of size M sampled randomly from the full training data. They specify the distribution $p(\boldsymbol{x}(t_M)|\boldsymbol{X}^H)$ using a conditional DDPM model. The forward process adds independent Gaussian noise to $\boldsymbol{x}(t_M)$ the same way as in DDPM. However, the reverse denoising model is conditioned on the history $\boldsymbol{X}^H$ which is represented with a fixed sized vector $\boldsymbol{z}$. After training is completed, the predictions are made in the following way:

1. Encode the history $\boldsymbol{X}^H$ with an RNN to get $\boldsymbol{z}$,
2. Sample initial prediction $\boldsymbol{x}_N(t_M) \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ from unit normal distribution,
3. Denoising using the sampling algorithm from Ho et al. [102] but conditioned on $\boldsymbol{z}$ to obtain $\boldsymbol{x}_0(t_M)$.

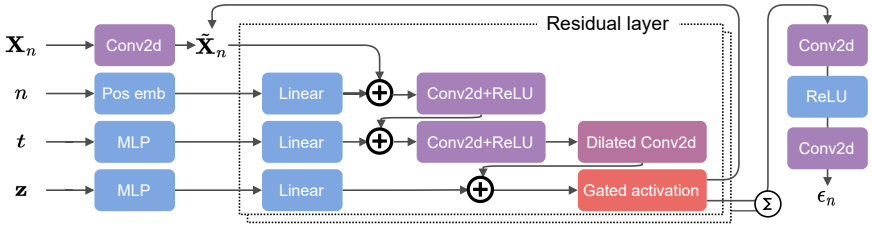


Figure 6.2: Overview of the forecasting model. The inputs are the noisy time series $\boldsymbol{X}_n$, diffusion steps n, observation times $\boldsymbol{t}$, and the history vector $\boldsymbol{z}$. The output is the predicted noise value $\boldsymbol{\epsilon}_n$.

The final denoised value is the forecast whereas sampling multiple values allows computing empirical confidence intervals of interest.

In Rasul et al. [222], the timestamps are always discrete and the prediction is autoregressive, that is, the values are produced *one by one*. Our diffusion framework offers the following key improvements: (i) the predictions can be made at *any* future time point, that is, in continuous time, not discrete steps; and (ii) we can predict *multiple* values in parallel which scales better on modern hardware.

In our case, the prediction $\boldsymbol{X}^F$ will not be a single vector but a sequence of vectors $(\boldsymbol{x}(t_M), \dots, \boldsymbol{x}(t_{M+K}))$ of size K, where K can vary in size. This type of data is naturally handled by our stochastic process diffusion as defined in Section 6.2. Note that the predicted values are also not conditionally independent but we model the interactions between them in the denoising model $\boldsymbol{\epsilon}_\theta$.

We design $\boldsymbol{\epsilon}_\theta$ in the following way. Previous observations are again represented with an RNN to obtain $\boldsymbol{z}$ and condition the reverse process. We propose an architecture similar to the TimeGrad model [222, 139] but which is not autoregressive as it outputs all the values simultaneously. Figure 6.2 shows the architecture overview. The inputs are the noisy future predictions $\boldsymbol{X}_n^F$, the diffusion step n, future timestamps $\boldsymbol{t}$ and the encoded history $\boldsymbol{z}$. In contrast to previous works, we use 2D convolution where the extra dimension corresponds to the time dimension.

For example, after training a Gaussian process DSPD model (Section 6.2.2), we can forecast in the following way:

1. Encode the history $\boldsymbol{X}^H$ with an RNN to get $\boldsymbol{z}$,
2. Sample initial prediction $\boldsymbol{X}_N^F$ from a GP prior,
3. Denoise using $\boldsymbol{\epsilon}_\theta(\boldsymbol{X}_n^F, \boldsymbol{t}, n, \boldsymbol{z})$ with Algorithm 2.

Instead of an RNN we can also use transformers [273] but we wanted to keep the architecture similar to Rasul et al. [222] and showcase the novel stochastic process-based diffusion.

6.3.2 Diffusion process as a neural process

So far we have used stochastic processes as noise sources to generate continuous functions. We can view such a model as a type of a stochastic process as well. Stochastic process is defined as a collection of random variables $\{X(t)\}_t$ indexed over some set $\mathcal{T}$, in our case $\mathcal{T} \subseteq \mathbb{R}$. We usually care about the finite sequences of points since this is what we encounter in real-world. In that case, a model that specifies a probability measure p is a stochastic process if it satisfies consistency conditions, as defined in the Kolmogorov extension theorem [198]. Essentially, the model has to be permutation equivariant, that is, the order of the incoming points should not matter.

Based on this, neural processes [76] are a class of latent variable models that define a stochastic process with neural networks. Given a set of data points (a dataset), the model outputs the probability distribution over the functions that generated this dataset. That is, for different datasets, the model will define different stochastic processes. Due to this behavior, neural processes bear a resemblance to the Gaussian processes but can also be viewed as a meta learning approach [105].

Let $\boldsymbol{X}^A$ denote the observed data, in our case, a time series, and let $\boldsymbol{X}^B$ be the unobserved data at the time points $\boldsymbol{t}^B$. Garnelo et al. [76] construct the encoder-decoder model that uses amortized variational inference for training [136]. The encoder takes in a set of observed points $(\boldsymbol{X}^A, \boldsymbol{t}^A)$ and outputs the distribution over the latent variable $q(\boldsymbol{z})$. The decoder takes in the sampled latent vector $\boldsymbol{z}$ and the query time points $\boldsymbol{t}^B$ and predicts the values of the unobserved points $\boldsymbol{X}^B$. In order to produce permutation equivariant measure, it is crucial that the encoder is permutation invariant, that is, the input order does not alter the result. Then the probability of $\boldsymbol{X}^B$ is conditionally independent given $\boldsymbol{z}$ [52]. This is easy to achieve using, for example, deep sets [294].

Since our approach samples functions, we can condition the generation on an input dataset $(\boldsymbol{X}^A, \boldsymbol{t}^A)$ in order to create our version of a neural process, based purely on the diffusion framework. The encoder will be a deterministic neural network that outputs the latent vector $\boldsymbol{z}$, contrary to Garnelo et al. [76] which outputs the distribution. Similar to Section 6.3.1, the diffusion is conditioned on $\boldsymbol{z}$ and we can output samples for any query $\boldsymbol{t}^B$. For example, if we again take DSPD-GP model, we sample as follows:

1. Permutation invariant encode $(\boldsymbol{X}^A, \boldsymbol{t}^A)$ to get $\boldsymbol{z}$,
2. Sample initial points $\boldsymbol{X}_N^B$ at $\boldsymbol{t}^B$ from a GP prior,
3. Denoise using $\boldsymbol{\epsilon}_\theta(\boldsymbol{X}_n^B, \boldsymbol{t}^B, n, \boldsymbol{z})$ with Algorithm 2.

Therefore, we capture the distribution $p(\boldsymbol{X}^B|\boldsymbol{X}^A)$ directly.

We achieve equivariance using a transformer-like model $\boldsymbol{\epsilon}_\theta$ [273] that utilizes a learnable RBF kernel as a similarity function. In particular, the model takes in $\boldsymbol{X}^A$ (observed points) as a conditioning variable and $\boldsymbol{X}_n^B$ (target points) as the noisy input. We first run a learnable RBF kernel $k(\boldsymbol{t}^A, \boldsymbol{t}^B)$ to obtain a similarity matrix $\boldsymbol{K}$ between the observed and unobserved time points. We project $\boldsymbol{X}^A$ with a neural network by transforming each point independently to obtain $\boldsymbol{Z}$, and then obtain the latent variable of the same time dimension size as $\boldsymbol{X}^B$ by multiplying $\boldsymbol{K}$ and $\boldsymbol{Z}$. We then use $\boldsymbol{Z}$ as a conditioning

vector and add it to projected $\boldsymbol{X}^B$, transform with a multilayer network, and obtain the output. During training, we adopt the approach from Garnelo et al. [76] of feeding in data such that we actually learn $p(\boldsymbol{X}^A \cup \boldsymbol{X}^B | \boldsymbol{X}^A)$ which helps the model learn to output high certainty around $\boldsymbol{t}^A$.

In the end, our model sees many observed-unobserved pairs corresponding to different true underlying processes. The model learns to represent the observed points $\boldsymbol{X}^A$ such that the denoising process corresponds to the correct distribution, given $\boldsymbol{X}^A$. After training is completed, we take a time series $\boldsymbol{X}^A$ and output the samples at any set of query time points $\boldsymbol{t}^B$. We can view such an approach as an interpolation or imputation model that fills-in the missing values across time. The main appeal is the ability to capture different stochastic processes within a single model.

A concurrent work [64] proposes using diffusion as an alternative to Gaussian processes, however, it uses an independent noise, therefore, it does not guarantee producing continuous functions.

6.3.3 Probabilistic time series imputation

The previous section considered interpolating in time. Now, we look into filling-in the missing values across the observation dimensions, that is, the imputation of the vectors. Each element $\boldsymbol{x}(t_i)$ of the time series $\boldsymbol{X}$ is assigned a mask $\boldsymbol{m}$ of the same dimension that indicates whether the j-th value $x^{(j)}$ of the vector $\boldsymbol{x}(t_i)$ has been observed ($m^{(j)} = 1$) or if it is missing ($m^{(j)} = 0$).

Given observed $\boldsymbol{X}^A$ and missing points $\boldsymbol{X}^B$, Tashiro et al. [262] propose a model that learns a conditional distribution $p(\boldsymbol{X}^B | \boldsymbol{X}^A)$. The model is built upon a diffusion framework and the reverse process is conditioned on $\boldsymbol{X}^A$, similar to that in Section 6.3.2. We extend this by introducing noise from a stochastic process, as presented above. The learnable model remains the same but we introduce the correlated noise in the loss and sampling. We posit that continuous noise process, as an inductive bias for the irregular time series, is a more natural choice.

6.4 Experiments

In this section we will present experimental evidence that suggests using stochastic process diffusion, as defined in Section 6.3 is favorable to regular diffusion on time series, and additionally, outperforms other time series models.

6.4.1 Probabilistic modeling

We start by investigating the pure generative capabilities of our model, that is, unconditional generation of time series.

Baselines. Previously, neural ODEs [35] were introduced as a way to capture the irregularly sampled time series since they can naturally handle the continuous time. As such, they can be seen as a building block that can also be used alongside our method to

CSPD–	Gauss	GP	OU
CIR	-0.4769±0.0249	-0.4766±0.0224	-0.4688±0.0178
OU	0.5089±0.0092	0.5222±0.0281	0.529±0.0138
Predator-prey	-3.4643±0.1039	-9.4934±0.2561	-7.0098±1.4929
Sine	-1.3438±0.0938	-3.2526±1.4108	-3.851±0.1231
Sink	-5.6637±0.1839	-11.4179±0.3627	-9.8487±1.2962
Lorenz	1.5162±0.3861	-3.4893±8.2181	-6.6377±0.2015

Table 6.1: Negative log-likelihood on synthetic data (lower is better) shows OU and Gaussian process noise are a better option than independent noise on datasets that exhibit smoothness, while being on par on stochastic datasets.

	CTFP	Latent ODE	DSPD-GP (Our)
CIR	0.999±0.0012	1.0±0.0	**0.511±0.0282**
Lorenz	0.995±0.0057	0.998±0.0019	**0.513±0.0288**
OU	0.783±0.0756	0.512±0.0331	**0.505±0.0458**
Predator-prey	0.789±0.0227	0.958±0.0213	**0.585±0.0219**
Sine	0.981±0.0104	1.0±0.0	**0.525±0.009**
Sink	0.727±0.1378	0.907±0.0394	**0.513±0.0103**

Table 6.2: Accuracy of the discriminator trained to distinguish real data and model samples. The closer the result is to 0.5 the better.

devise different denoising networks. Rubanova et al. [229] construct an encoder-decoder architecture based on neural ODEs which resembles the variational autoencoder [136]. The time series is, thus, modeled in a latent space by sampling a random vector which is propagated with an ODE. Neural SDEs [156] extend this by adding noise in every solver step but they either do not produce noisy-enough samples [156] or are GAN-based and difficult to train [128]. Finally, continuous-time flow process (CTFP) [56] uses normalizing flows (Section 2.2.2) to generate the time series by sampling the initial noise from the stochastic process and transforms it with an invertible function to obtain a sample from the target distribution. Although this allows exact likelihood training, the method cannot capture some processes [55] and is often augmented to be trained as a VAE.

Ablation. We test our DSPD and CSPD with independent Gaussian noise and noise from a stochastic process (GP and OU) on six synthetic datasets, corresponding to deterministic and stochastic dynamical systems. See Appendix D.2.1 for more details. We first check whether using a model that captures interactions across time (for example, RNN or transformer) outperforms the model that treats each data point in the time series independently. Table D.2 shows we need to model interactions across time, as expected.

Now, we check if using our stochastic process outperforms independent Gaussian noise; we compare our method to Ho et al. [102] and Song et al. [252]. Table 6.1 shows that using a stochastic process achieves lower negative log-likelihood for CSPD (see Appendix D.2.1 for DSPD). This is especially evident on datasets where we need to generate smoother samples. Finally, Figure 6.3 demonstrates the quality of the samples.

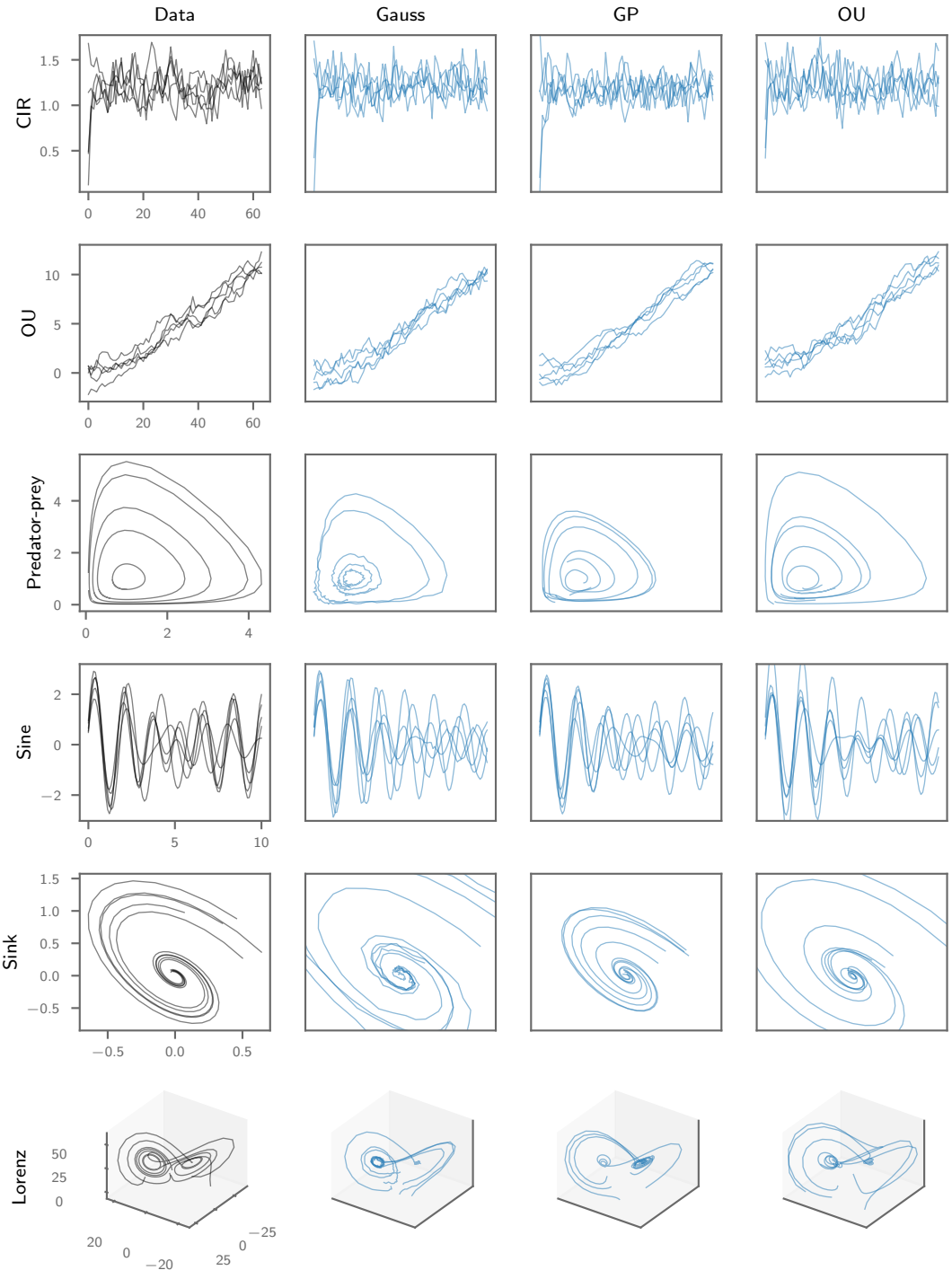


Figure 6.3: Real data (left column) and 5 samples generated from DSPD using Gaussian noise (corresponds to DDPM), OU process, and Gaussian process noise, respectively.

		Electricity	Exchange	Solar
TimeGrad	NRMSE	0.064±0.007	0.013±0.003	0.799±0.096
	Energy score	8425±613	0.057±0.002	**150±17**
Ours	NRMSE	**0.045±0.002**	**0.012±0.001**	**0.757±0.026**
	Energy score	**7079±164**	**0.031±0.002**	166±12

Table 6.3: NRMSE (top rows) and energy score (bottom rows) on real-world forecasting data.

Results. We quantitatively compare the generative power of our model with the established baselines for irregular time series modeling, namely, latent ODEs [229] and CTFP [56] (more details are in Appendix D.2.1). In short, after training a single generative model we use it to sample new data. The original and generated data is used to train a new model that learns to discriminate between them. If the discriminator cannot be trained, that is, its prediction is not better than a random guess (having accuracy of around 50%), we say the generative model captures the true distribution.

Table 6.2 compares our model with the baselines and demonstrates that we produce samples that are indistinguishable to a powerful transformer-based discriminator. The same does not hold for the competing methods.

6.4.2 Forecasting

We test our model as defined in Section 6.3.1 and Figure 6.2 against TimeGrad [223] on three established real-world datasets: Electricity, Exchange and Solar [147]. Due to the limitations of the CRPS-sum metric [141], we report the NRMSE and the energy score [81] averaged over five runs, but we note that the rank of the model's performance does not change when using other metrics as well. Table 6.3 shows that our method outperforms TimeGrad even though we predict over the complete forecast horizon at once, and Figure 6.4 demonstrates the prediction quality alongside the uncertainty estimate.

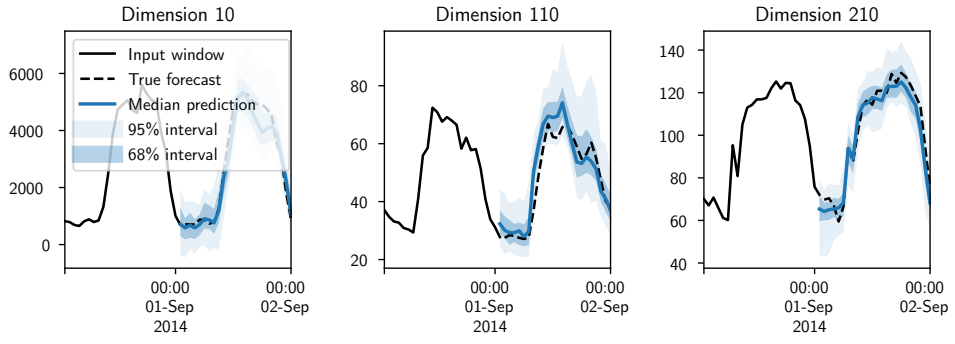


Figure 6.4: Forecast and uncertainty intervals on Electricity.

Missing ratio:	10%	50%	90%
CSDI	0.520±0.055	**0.644±0.024**	0.818±0.02
DSPD-GP (Our)	**0.498±0.036**	**0.644±0.029**	**0.815±0.019**

Table 6.4: Imputation RMSE on Physionet data with varying amounts of missingness.

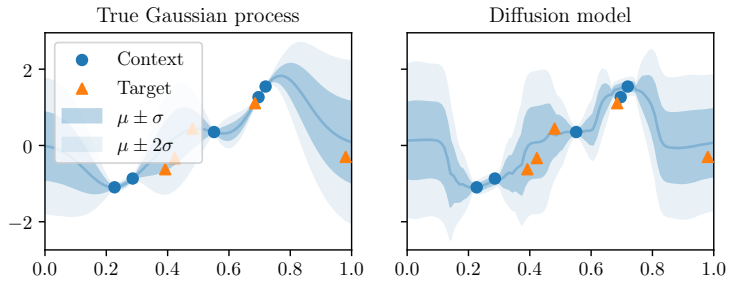


Figure 6.5: Sampled curves given a set of points.

6.4.3 Neural process

We construct a dataset where each time series $\boldsymbol{X}$ comes from a different stochastic process, by sampling from Gaussian processes with varying kernel parameters and time series lengths. This is a standard training setting in neural process literature [76]. In our denoising network, we modify the attention-like layer to make it stationary (see Appendix D.2.2) and train as described in Section 6.3.2. Due to the use of tanh activations in the final layers, combined with its stationary, our model extrapolates well, that is, when tanh saturates the mean and the variance fall to zero-one. This is the same behaviour we see in the GP with an RBF kernel, for example. The quantile loss of the unobserved data under the true GP model is 0.845 while we achieve 0.737 which indicates we capture the true process, which can also be seen in Figure 6.5. We remark that the attentive neural process [132] does not produce the correct uncertainty.

Finally, in Figure D.1 (Appendix D.2.2) we compare the OU and GP, across different kernels. It shows that the noise process is connected to the final sample *smoothness*, however, the marginal distributions are the same and correct.

6.4.4 Imputation

We compare to the CSDI [262], introduced in Section 6.3.3, on an imputation task. To this end, we use exactly the same training setup, including the random seeds and model architecture, but change the noise source to a Gaussian process. Following Tashiro et al. [262], we use Physionet dataset [245] which is a collection of medical time series collected at an hourly rate. It already contains missing values but for testing purposes, we choose varying degrees of missingness and report the results on the test set. We update the loss and sampling accordingly, as in Section 6.2. Table 6.4 shows that we outperform the original CSDI model even though we only changed the noise, and the dataset we used has

regular time sampling. Appendix D.2.3 contains more details, including the statistical test that shows our results have statistical significance.

6.5 Discussion

In this chapter, we introduced a novel generative model for continuous functions. It can also be viewed as a neural stochastic process or a generative model for solutions to stochastic differential equations. We also demonstrate how it can be used in conventional (both regular and irregularly-sampled) time series tasks such as forecasting, interpolation, and imputation. In the experiments we showed that the improvements over the previous works come from using the stochastic process as the noise source, and using the model that takes in the whole time series at once. The results demonstrate the practical utility of our method and validate our motivation.

We used bare-bones diffusion without extensive tuning to demonstrate the modeling potential and make a fair comparison to other methods. However, it should be straightforward to improve upon our models by implementing recent advances in diffusion models [e.g., 192]. In case we have a large number of points, we can consider replacing the current sampling strategies with more scalable variants, such as switching to a sparse GP [213]. Additionally, one can train a latent diffusion model [228] by first learning the time series encoder-decoder which might be helpful for high-dimensional data, such as those we encountered in the forecasting task. It would be interesting to explore different architecture choices, for example, implement improvements in conditioning models via learned activations [217]. Finally, we can also apply the presented methods to other areas outside the time series domain, such as modeling point clouds or even images, as we have demonstrated that our method is competitive on regular grids.

7 Conclusion

In this thesis we answered important questions regarding irregularly-sampled time series; namely, how to process such data using modern machine learning methods, and how to define deep generative models that produce new realizations and forecast with uncertainty. The main takeaway is that combining the expressive power of neural networks with the formal frameworks that implement constraints on the model results in state-of-the-art performance. Neural flows are a good example of this: we design a neural network in such a way that it always defines a flow—a solution to an ordinary differential equation—which allows us to evaluate queries very quickly while having enough expressive power to learn complicated dynamics. We expand on this architecture to design irregular time series encoders and decoders and demonstrate the usefulness on various applications.

The main goal of the thesis was to define generative models for irregular sequences, both in the case of event-based data and continuous measurements. The former is the problem of modeling arrival times in an irregular sequence which is traditionally approached with temporal point processes. We frame the task of learning the intensity as a neural density estimation problem which enables evaluating exact likelihood and straightforward sampling of new sequences. Since events can have different types, we classify them with a model conditioned on the arrival time. Our model can change the prediction over time by evolving the distribution over the simplex. This allows us to assign an amount of certainty in the prediction including the case where the model predicts that no event will occur. Finally, we tackle the case of sequences which are measured continuously; the assumption is that the data follows an underlying continuous function so we augment the denoising diffusion approach to include this constraint. The resulting model can be seen as a neural process as it samples functions as new realizations.

7.1 Retrospection

The approaches and results we presented in the thesis have been published across several years which is why we now discuss our contributions in the broader research context.

Encoder architectures for irregular time series are an important part of each of the proposed approaches (Chapters 3-6) that could be investigated in more detail, but we focused on highlighting other main contributions in these chapters. Additionally, the state-of-the-art is constantly evolving and some new approaches have been introduced in the meantime; our neural flows are one such example since they were not available as the encoder architecture in Chapters 3 and 4. The choice of the encoder is orthogonal to the conclusions we want to show, in particular, Chapter 3 demonstrates how, given an

RNN encoder, the density parameterization improves upon the intensity models. It is reasonable to expect that the same result will hold if we replace an RNN with a neural flow, which is what we finally show in Section 5.3.

Nevertheless, we will take a look at some of the recent advances in time series modeling. One example are transformers (Section 2.1.3) which can be successfully used as a drop in replacement for RNNs [154, 164, 283, 296, 300]. Shukla and Marlin [243] propose a modification of the basic architecture to encode irregular samples. They introduce a set of reference time points that correspond to queries, similar to pseudo points in our uncertainty model (Chapter 4), and additionally, in the attention layer, they use actual arrival times as keys and observations as values. We use transformers for time series in Chapter 6 and design a modified similarity function that resembles a radial basis kernel.

A different type of encoder is based on neural controlled differential equations [130, 184]. The data is assumed to follow a continuous underlying function, like in Chapter 6, so it is interpolated with piecewise polynomial functions called splines. The network is a continuous analogue of the RNN since it takes values from a continuous function and continuously updates the hidden state with the passage of time. We [16] propose an alternative where the operations on the interpolated data are done in a functional form until the very last layer where we collect the actual values in order to forecast or classify. Although both approaches show some improvements on irregular time series tasks, transformers and RNNs are the preferred option for simplicity of implementation and lower computational requirements.

Temporal point processes. A natural way to define a TPP is via autoregressive intensity, which is a common approach in both classical [99] and neural TPP models [61, 180]. We did the same in Chapter 3, replacing the intensity with density. The drawback of this method is an inherent sequential nature of computation so sampling new sequences is done by sampling points one by one. Shchur et al. [241] extend on our approach and define a normalizing flow model—the observed sequence is transformed in parallel to a base process, corresponding to a homogenenous Poisson process. The reverse transformation is also done in parallel and corresponds to sampling new sequences. The method is build on top of the random time change theorem (Section 2.2.1) combined with triangular maps that preserve the causality. Although less flexible by design, Shchur et al. [241] show that such a model can match the performance of autoregressive models at a fraction of a cost, while supporting new applications such as modeling discrete-state systems.

Our log-normal model (Section 3.2.2) has proved to be useful in many applications, including distributed systems [253], cybersecurity [183], imputation [94], retrieval [95], anomaly detection [239], learning temporal knowledge graphs [205], human activity prediction [93], healthcare [68], and so on.

Uncertainty. The two models that we introduced in Chapter 4 use the same type of loss: an expectation over the categorical distribution. The parameterized output is also known as the epistemic distribution [31]. Our models are specifically designed for the irregular time series, therefore, we output the evolution of this distribution across time. As already

hinted in the results from Section 4.3, Dirichlet seems to be a more robust model choice compared to the one based on a Gaussian process. It is also more interpretable as the mean and the variance, together with the weight, correspond to the time, confidence range and confidence in the prediction, respectively. We avoid using out-of-distribution samples and introduce regularization to obtain higher uncertainty on parts where we did not observe data. A followup work by Charpentier et al. [31] uses a similar Dirichlet parameterization together with our uncertainty cross entropy loss and additionally adds the entropy regularizer. They introduce a learnable posterior distribution which allows evaluating the confidence in the prediction. This is similar in spirit to our original paper but it is extended for general classification, beyond time series.

Historically, Chapter 4 preceded Chapter 3, setting up the ground works for parameterizing the irregular time series decoders. Since we are dealing with one-dimensional outputs in both cases, we opt for simple parametric functions. In Chapter 4, for example, we output the mixture of Gaussians as a function of time which corresponds to the intensity in the TPP approach. However, our preferred TPP approach uses normalized functions (densities) over the positive reals, as it gives an equivalent but simpler implementation. For higher dimensional continuous dynamics, we propose a solution in Chapter 5.

Neural flows. Just as neural differential equations grew in popularity [35, 156, 246], the interest in learning their closed-form solutions was becoming ever larger, including our work presented in Chapter 5. Recently, Hasani et al. [98] investigated a continuous-time neural network defined with a specific differential equation and derived a closed-form approximation, again avoiding the expensive numerical solvers. Song et al. [250] aim to produce a diffusion model without iterative generative procedure by defining a function that stays on the probability flow curve, a property which they call consistency. They also enforce an initial condition so the final model resembles our ResNet flow (Equation 5.1).

Schirmer et al. [236] redefine a continuous RNN by evolving the hidden state between the observations with a linear SDE. This exactly corresponds to a Kalman filter, therefore, the predict and update steps can be easily computed, unlike in GRU-ODE [51]. Previously, Bézenac et al. [13] proposed a similar approach to model regular time series coupled with normalizing flows as the emission function, making the inference tractable. Such a linear flow is less expressive than a general neural flow, however, the hidden dimension can be arbitrarily larger than the input data so it might still capture the correct dynamics [281].

In Section 5.4.2, we show that neural flows cannot guarantee solutions to autonomous dynamics, without explicitly implementing this additional constraint. The constraint is, however, limiting the expressiveness of the network so not all possible configurations can be modeled. Despite this, Zhi et al. [298] model ODEs in a similar manner as normalizing flows—a base ODE (for example, linear) is mapped to the target space via diffeomorphism. The drawback of the method is that it preserves the topology of the original space, which was not an issue in density estimation, but becomes problematic in dynamical systems.

Another approach is to learn solutions to differential equations in the Laplace domain [104]. This can capture a large family of equations, beyond ODEs, such as delay differential equations, integro-differential, forced and stiff differential equations. Holt et al. [104] show

that our neural flows can, without any modifications, beat ODE-based methods on these exotic problems, but they lack in performance compared to the Laplace approach. We find the conclusion that flows outperform ODEs satisfying as that was the main focus of our comparison. It might be possible to extend the neural flows to these problems by introducing further inductive biases in the architecture.

It is worth noting that neural ODEs still have their place, especially when the domain knowledge can aid us in constructing the equation [214]. This argument can again be flipped, that is, in physics-informed neural networks we use known PDEs to guide learning the solution [215] and in neural operators we utilize the boundary condition to build a solver [158]. Although differential equations offer a compact way to write down laws of nature, the neural version is far from easy to parse. In addition, many fields actually only care about solutions, for example, fluid dynamics is governed by known equations which are notoriously hard to solve. A lot of the effort is put into solving these equations as efficiently as possible and some recent works use machine learning to achieve this [158, 25].

As a final remark, we recall that we used a continuous normalizing flow, an ODE-based generative model, to capture invariant densities (Section 3.4.1). This was an example of a model that was simpler to define through vector fields. However, Bose and Kobyzev [24] recently show that the equivariance in transformation can be achieved with discrete normalizing flows, such as residual-based flows.

In the end, choosing between flows and differential equations is the matter of convenience. For some problems, specifying an ODE is easy but learning it will be expensive so there will always be an incentive to optimize for best computation performance while incorporating the most domain knowledge. This is a setting in which neural flows thrive. As we have seen, neural flows and, more generally, much of the recent research surrounding neural networks uses known primitives (for example, fully connected layers and attention) and incorporates the constraints, such as invertibility, to make known frameworks (like TPPs and ODEs) more flexible. Combining known frameworks and neural networks in such a principled way seems to be the way to create powerful models that solve hard problems.

Denoising diffusion. Our generative model from Chapter 6 operates on functions instead of discrete points. Concurrently, Kerrigan et al. [126] propose a very similar approach of modeling functions with diffusion by defining a Gaussian measure on Hilbert spaces. This formalizes our ideas using the results from measure theory. Another concurrent work [64] views diffusion on functions as neural processes, similar to our formulation in Section 6.3.2. Subsequent work by Lim et al. [162] replaces the Gaussian process noise with a Gaussian random field, an extension which generalizes to higher dimensions. Together with neural operators, this allows them to model solutions to PDEs, in particular, they generate solutions to the Navier-Stokes equation. This can also be applied to sample images or to generate functional data on non-Euclidean spaces. Finally, Pidstrigach et al. [208] use an empirical covariance matrix, calculated from data, to achieve certain amount of smoothness of the generated functions. This answers the question of how to choose hyperparameters of GP and OU covariances from Chapter 6.

7.2 Open questions

In this section we take a look at the future of machine learning in irregularly-sampled time series. The questions from the introduction have been answered through the thesis and although we cannot say there is one correct answer to any of them, we believe that the approaches we presented in Chapters 3-6 will stand the test of time. In favor of this statement are all the followup works that we presented in the previous section. In particular, modeling TPPs with density or intensity with closed-form integral is a natural improvement over the unconstrained neural TPPs. The method still relies on the decades of TPP research but brings additional flexibility in the advent of big data. Capturing uncertainty with the distribution over the categorical distribution can be seen as a Bayesian way to make predictions. Modeling closed-form solutions with neural networks is becoming increasingly popular, and in more general, incorporating domain specific constraints is a powerful way to create flexible and accurate models. Generative diffusion has a simple philosophy—predict the added noise to remove it from corrupted data. It is similar to VAEs and normalizing flows but iterative, and it keeps a simple single scalar loss unlike GANs that play an adversarial game which is harder to train. Further, our application to functions can be expanded to other spaces such as images and manifolds. Taking all of this into consideration, we believe that the presented ideas are here to stay; so we now turn to the remaining unanswered questions.

Representing time series. Even though we have spent quite some time looking at different irregular time series encoders, it seems there is no *single one* that should always be used across all tasks. As we have seen, there are various trade offs that we can make, for example, we can choose between fast RNNs with linear evolution of the hidden state and neural CDEs that interpolate data, basically oversampling across time. Choosing the best encoder is the problem of Pareto optimality, but we cannot rule out that new better architectures could dominate across multiple relevant properties. Many datasets that we considered do not have very long dependencies so it is possible to use transformers, especially with modern hardware and their proven track record on other domains. A different approach is using random untrained convolutional layers that extract random features which are then fed into a simple linear layer [53]. Similarly, reservoir computing [168, 115] is common in modeling dynamical systems—randomly initialized transformations are used to extract features which are then used as an input to a simpler, often linear, network. These approaches hint that extracting many low level features (wider networks) is preferable to extracting hierarchical features (deeper networks). Although this works on some datasets, it is not hard to imagine that such an approach would not scale to more complicated tasks. In this thesis, we have shown that continuous time equivalents of RNNs (Chapter 5) achieve competitive results.

The problem of showing that some encoder is preferred to others directly connects to a lack of common benchmarks for irregular time series, which are present in many other data types [54, 275]. It is therefore harder to demonstrate if some method achieves better results overall or only on some specific subset of problems. The main issue could actually be a bit more subtle: time series come from very different domains and it might not be

reasonable to expect that a single architecture excels on all, compared to many specialized models. After all, we have shown that introducing domain specific constraints improves performance. Thus, it might only make sense to talk about the need for benchmarking in TPPs, or in dynamical systems, but not both at the same time.

On the other hand, large language models have recently shown impressive results on a vast variety of natural language tasks, exhibiting few-shot behavior [26]. Could we build a similar universal model for time series, even when dealing with irregularly-sampled data? Using periodic activations to embed irregular time [124, 243] promises unification between regular and irregular methods, at least for fixed-sized vector representation of the whole time series. Although pre-trained networks for time series exist [92, 123, 170, 287], we are yet to see wider adoption of this approach, especially compared to how omnipresent it is in natural language processing.

Applications. We have considered many different data sources, including healthcare, social media, smart houses and location data, besides others. This demonstrates that irregularly-sampled time series can be naturally found in many domains and we can also have different applications per domain: forecasting, classification, imputation etc. Beyond direct application of our methods to other data sources, we can also consider using ideas and applying them to other tasks; just like Chapter 4 inspired uncertainty in general classification [31], the ideas of Chapter 6 have already been applied to PDEs [162] and can be used in image modeling. Lastly, adapting our methods for real-world use would require satisfying different requirements, from those imposed by data itself to legal compliance. We will discuss different ways to inspect deep models in the rest of this section.

Broader impact of black box models. Software development often prefers deploying quickly at the expense of verifiability. This should not apply to all software—self driving cars are an obvious example. However, the impact of even seemingly innocent technologies can be unexpectedly catastrophic.[1] As the models slowly shift from research domain to real-world usage, it becomes our responsibility to provide safety guarantees that the models align with what we want from them. Improving upon a single scalar objective[2] is what allows deep learning to thrive but it also allows for perverse solutions that find shortcuts in data [187]. The reasoning[3] that models use in such solutions would certainly be regarded as unacceptable, if it was accessible for inspection in the first place. This is because models are not transparent, they are black boxes so it is not possible to anticipate their behavior from the structure and parameters alone.

[1] Social media that optimizes for engagement can amplify hate speech.

[2] Simple objectives are bad in general, this fact is expressed through different names: perverse incentive, Campbell's [28] or Goodhart's law [86], or the cobra effect. The last name comes from an attempt at eradicating cobras in Delhi using a financial incentive for each captured cobra which resulted in breeding cobras to earn from the program.

[3] Here, and in much of the remaining text we use human traits to describe inanimate. Since we already used the term *neural* to describe all of our models, although there is not much connection to the biological brain, such personification should mostly be seen as a stylistic choice.

Explaining the prediction post-hoc is one way to enable users inspect models. Some works use backpropagation to highlight parts of an input that contribute the most to a particular prediction [254, 276, 299]. A different feature importance extraction method builds a surrogate model that locally explains the current prediction [166, 225], which has been successfully applied to time series [237]. Shapelets build a representative set of time series subsequences that are then used in prediction [289]. That is, shapelets are learned shapes which are used as a signal for prediction when found in a particular time series. There exists an extension for irregular data [131], whereas we propose a deeper architecture that uses spline convolution layers which are interpretable in a similar way to shapelets [16].

Adversarial attacks are purposefully changed inputs designed to fool the network and get a behavior different than expected. For example, one can modify images with a small amount of noise to change the predicted class [258] or change the structure of the graph yielding the same undesirable outcome [304]. Unsurprisingly, time series models can also be attacked by such methods [122]. A way of defending against adversaries is by making models robust. One method that works on time series is to obtain the robustness certificates [291] from randomized smoothing [44]—majority vote under input perturbations.

Interpretability and robustness of irregular time series have not been studied in detail but there is no reason to believe that the outcome would be any different than with regular time series, or other data types. However, since healthcare measurements often contain nonuniform sampling, it will be necessary to study these questions in more detail in order to have trustworthy algorithms that can be applied in real-world environments.

Another important topic in machine learning and, consequently, in irregular time series is mitigating biases. A decision-maker that is biased can be considered unfair, meaning it will base the decision on properties that should not be relevant. One example is discriminating based on race or sex which already falls under illegal activity in many countries.[4] Therefore, one has to take such issues into account when building real-world applications, if for no other reason, to comply with local laws. Unfortunately, it is very easy to introduce biases into models from an early stage of gathering data, by adding or omitting features [42, 186], using inappropriate sampling and aggregation [256, 274]. The question is how can we avoid generating further discrimination using machine learning.

There are various definitions of fairness [179], such as achieving equal odds or equal opportunity regardless of whether data comes from a protected group or not. Note that it is not possible to satisfy multiple fairness criteria at once [137] which further complicates things. An example of trying to resolve representation issue is data augmentation. For example, to deal with the problem of gender representation in natural language datasets, one can augment data to create a gender-swapped corpora [297]. It is not immediately clear how this would translate to time series data since there might be complicated features that reveal the protected group and allow the model to discriminate based on it.

Models themselves can also exhibit bias and there have been approaches that try to mitigate this [5, 82, 178], for example, by modifying the training procedure [12, 233].

[4] In Germany, *Allgemeines Gleichbehandlungsgesetz* is an act that aims to prevent discrimination.

Lastly, one can apply post-processing techniques to convert a potentially unfair predictor to a fair one [97]. Since we treat the model as a black box here, we can readily apply this to time series models. Fairness in time series [301] benefits a lot from the general fairness research so it is reasonable to believe it will be used more in practice as the field progresses. Since irregular data can have special properties (for example, continuity) it will be interesting to see how the above mentioned methods can adapt to this.

In many applications, large amounts of sensitive data are used for training which raises the questions of safe storage but also the prevention of leakage from a trained model [249]. Differential privacy [65] is a field that investigates disclosing datasets and aggregates from data without disclosing any private information. Again, these methods are quite general but time series data comes with its own challenges which should be addressed [221].

Finally, we also have to consider model safety, especially when running in critical applications such as healthcare or critical infrastructure like powerplants. AI safety is an umbrella term which connects, besides other, robustness and interpretability. Recently, the term alignment grew in popularity, which denotes aligning model's given objective with an intended objective. A part of the problem can be ascribed to optimizing simple objectives without any constraints, as we have mentioned before. Another potential issue that some researchers consider is that building larger models can lead to unexpected consequences [75, 191]. So far this is mostly concerning large (language) models, and while we do not believe the same issues can arise in the methods that we considered in this thesis due to the model sizes and the tasks we tackle, it is important to keep such considerations in mind going forward.

Ideas presented in this thesis are not inherently good nor bad. We anticipate that using marked point processes might lead to unfair models if marks are not representative of real population or if they contain unnecessary sensitive information. For example, a spatial model trained on crime data [15] can suggest increased policing in perceived dangerous areas which results in more reported crimes—creating a feedback loop [108]. It is clear that the type of fairness is dictated by the task at hand, so there is no single off-the-shelf solution that fits all problems. We hope that one of the main future applications will be in healthcare. The importance of making the effort to make models safe cannot be overstated since the consequences of poorly designed models can lead to undesirable outcomes both on an individual level but also creating further social disparity [172]. Incorporating all of the above requirements, such as safety and privacy, might not be easy but we believe that the final products can impact the world in a positive way that will be measured in the number of saved lives.

Bibliography

[1] Namrata Anand and Tudor Achim. "Protein Structure and Sequence Generation with Equivariant Denoising Diffusion Probabilistic Models". In: *arXiv 2205.15019* (2022) (cited on page 76).

[2] Brian D.O. Anderson. "Reverse-time Diffusion Equation Models". In: *Stochastic Processes and their Applications* 12.3 (1982), pp. 313–326 (cited on page 23).

[3] Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. "Structured Denoising Diffusion Models in Discrete State-Spaces". In: *Neural Information Processing Systems (NeurIPS)*. 2021 (cited on page 76).

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer Normalization". In: *arXiv 1607.06450* (2016) (cited on page 11).

[5] Arturs Backurs, Piotr Indyk, Krzysztof Onak, Baruch Schieber, Ali Vakilian, and Tal Wagner. "Scalable Fair Clustering". In: *International Conference on Machine Learning (ICML)*. 2019 (cited on page 97).

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *International Conference on Learning Representations (ICLR)*. 2015 (cited on page 10).

[7] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. "Deep Equilibrium Models". In: *Neural Information Processing Systems (NeurIPS)*. 2019 (cited on page 68).

[8] Stefan Banach. "Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales". In: *Fund. math* 3.1 (1922), pp. 133–181 (cited on page 7).

[9] Ron Begleiter, Ran El-Yaniv, and Golan Yona. "On Prediction Using Variable Order Markov Models". In: *J. Artif. Int. Res.* 22.1 (2004) (cited on page 48).

[10] Jens Behrmann, Will Grathwohl, Ricky TQ Chen, David Duvenaud, and Jörn-Henrik Jacobsen. "Invertible Residual Networks". In: *International Conference on Machine Learning (ICML)*. 2019 (cited on page 57).

[11] Chris Bender, Juan Jose Garcia, Kevin O'Connor, and Junier Oliva. "Exchangeable Generative Models with Flow Scans". In: *AAAI Conference on Artificial Intelligence*. 2020 (cited on page 38).

[12] Richard Berk, Hoda Heidari, Shahin Jabbari, Matthew Joseph, Michael Kearns, Jamie Morgenstern, Seth Neel, and Aaron Roth. "A convex framework for fair regression". In: *arXiv preprint arXiv:1706.02409* (2017) (cited on page 97).

[13] Emmanuel de Bézenac, Syama Sundar Rangapuram, Konstantinos Benidis, Michael Bohlke-Schneider, Richard Kurle, Lorenzo Stella, Hilaf Hasson, Patrick Gallinari, and Tim Januschowski. "Normalizing Kalman Filters for Multivariate Time Series Analysis". In: *Neural Information Processing Systems (NeurIPS)*. 2020 (cited on pages 81, 93).

[14] Marin Biloš, Bertrand Charpentier, and Stephan Günnemann. "Uncertainty on Asynchronous Time Event Prediction". In: *Neural Information Processing Systems (NeurIPS)*. 2019 (cited on pages 2, 39, 61).

[15] Marin Biloš and Stephan Günnemann. "Scalable Normalizing Flows for Permutation Invariant Densities". In: *International Conference on Machine Learning (ICML)*. 2021 (cited on pages 11, 38, 98).

[16] Marin Biloš, Emanuel Ramneantu, and Stephan Günnemann. "Irregularly-Sampled Time Series Modeling with Spline Networks". In: *Workshop on the Continuous Time Methods for Machine Learning, International Conference on Machine Learning (ICML)*. 2022 (cited on pages 92, 97).

[17] Marin Biloš, Kashif Rasul, Anderson Schneider, Yuriy Nevmyvaka, and Stephan Günnemann. "Modeling Temporal Data as Continuous Functions with Process Diffusion". In: *International Conference on Machine Learning (ICML)*. 2023 (cited on page 2).

[18] Marin Biloš, Andrei Smirdin, and Stephan Günnemann. "Modeling Solutions to Ordinary and Partial Differential Equations with Continuous Initial Value Networks". In: *Workshop on the Continuous Time Methods for Machine Learning, International Conference on Machine Learning (ICML)*. 2022 (cited on pages 70, 71).

[19] Marin Biloš, Johanna Sommer, Syama Sundar Rangapuram, Tim Januschowski, and Stephan Günnemann. "Neural Flows: Efficient Alternative to Neural ODEs". In: *Neural Information Processing Systems (NeurIPS)*. 2021 (cited on page 2).

[20] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. "Pyro: Deep Universal Probabilistic Programming". In: *Journal of Machine Learning Research* (2018) (cited on page 126).

[21] Christopher M Bishop. "Mixture Density Networks". In: *Aston University* (1994) (cited on pages 25, 31, 33).

[22] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. "Variational Inference: A Review for Statisticians". In: *Journal of the American statistical Association* 112.518 (2017), pp. 859–877 (cited on page 5).

[23] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. "Weight Uncertainty in Neural Network". In: *International Conference on Machine Learning (ICML)*. 2015 (cited on page 49).

[24] Avishek Joey Bose and Ivan Kobyzev. "Equivariant Discrete Normalizing Flows". In: *arXiv 2110.08649* (2021) (cited on page 94).

[25] Johannes Brandstetter, Daniel Worrall, and Max Welling. “Message Passing Neural PDE Solvers”. In: *International Conference on Learning Representations (ICLR)*. 2022 (cited on pages 71, 94).

[26] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. “Language Models are Few-shot Learners”. In: *Neural Information Processing Systems (NeurIPS)*. 2020 (cited on page 96).

[27] Craig Calcaterra and Axel Boldt. “Approximating with Gaussians”. In: *arXiv 0805.3795* (2008) (cited on page 45).

[28] Donald T Campbell. “Assessing the Impact of Planned Social Change”. In: *Evaluation and program planning* 2.1 (1979), pp. 67–90 (cited on page 96).

[29] Oscar Celma. “Music Recommendation”. In: *Music recommendation and discovery*. Springer, 2010, pp. 43–85 (cited on page 128).

[30] Bo Chang, Lili Meng, Eldad Haber, Lars Ruthotto, David Begert, and Elliot Holtham. “Reversible Architectures for Arbitrarily Deep Residual Neural Networks”. In: *AAAI Conference on Artificial Intelligence*. 2018 (cited on page 68).

[31] Bertrand Charpentier, Daniel Zügner, and Stephan Günnemann. “Posterior Network: Uncertainty Estimation Without OOD Samples via Density-based Pseudo-counts”. In: *Neural Information Processing Systems (NeurIPS)* (2020) (cited on pages 92, 93, 96).

[32] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. “Recurrent Neural Networks for Multivariate Time Series with Missing Values”. In: *Scientific reports* 8.1 (2018), pp. 1–12 (cited on page 59).

[33] Liming Chen, Chris D Nugent, Jit Biswas, and Jesse Hoey. *Activity Recognition in Pervasive Intelligent Environments*. Vol. 4. Springer Science & Business Media, 2011 (cited on page 50).

[34] Ricky T. Q. Chen, Brandon Amos, and Maximilian Nickel. “Neural Spatio-Temporal Point Processes”. In: *International Conference on Learning Representations (ICLR)*. 2021 (cited on pages 56, 62, 63, 66, 67, 145, 146, 150, 153).

[35] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. “Neural Ordinary Differential Equations”. In: *Neural Information Processing Systems (NeurIPS)*. 2018 (cited on pages 7, 18, 55, 60, 62, 68, 73, 84, 93, 145, 159).

[36] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. “Training Deep Nets with Sublinear Memory Cost”. In: *arXiv 1604.06174* (2016) (cited on page 68).

[37] Ilya Chevyrev and Andrey Kormilitzin. “A Primer on the Signature Method in Machine Learning”. In: *arXiv 1603.03788* (2016) (cited on page 70).

[38] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. “On the Properties of Neural Machine Translation: Encoder-decoder Approaches”. In: *Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST-8)*. 2014 (cited on pages 48, 57, 141).

[39] Hyungjin Chung, Byeongsu Sim, and Jong Chul Ye. "Come-closer-diffuse-faster: Accelerating Conditional Diffusion Models for Inverse Problems through Stochastic Contraction". In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2022 (cited on page 76).

[40] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *arXiv 1412.3555* (2014) (cited on pages 9, 160).

[41] Marco Ciccone, Marco Gallieri, Jonathan Masci, Christian Osendorfer, and Faustino Gomez. "NAIS-Net: Stable Deep Networks from Non-Autonomous Differential Equations". In: *Neural Information Processing Systems (NeurIPS)*. 2018 (cited on page 68).

[42] Kevin A Clarke. "The Phantom Menace: Omitted Variable Bias in Econometric Research". In: *Conflict management and peace science* 22.4 (2005), pp. 341–352 (cited on page 97).

[43] Earl A Coddington and Norman Levinson. *Theory of Ordinary Differential Equations*. Tata McGraw-Hill Education, 1955 (cited on page 7).

[44] Jeremy Cohen, Elan Rosenfeld, and Zico Kolter. "Certified Adversarial Robustness via Randomized Smoothing". In: *International Conference on Machine Learning (ICML)*. 2019 (cited on page 97).

[45] William Jay Conover. *Practical Nonparametric Statistics*. Vol. 350. john wiley & sons, 1999 (cited on page 161).

[46] George Cybenko. "Approximation by Superpositions of a Sigmoidal Function". In: *Mathematics of control, signals and systems* 2.4 (1989) (cited on pages 5, 31).

[47] Daryl J Daley and David Vere-Jones. *An Introduction to the Theory of Point Processes: Volume I: Elementary Theory and Methods*. Springer Science & Business Media, 2007 (cited on page 14).

[48] Daryl J Daley and David Vere-Jones. *An Introduction to the Theory of Point Processes: Volume II: General Theory and Structure*. Springer Science & Business Media, 2007 (cited on pages 5, 15).

[49] Hennie Daniels and Marina Velikova. "Monotone and Partially Monotone Neural Networks". In: *IEEE Transactions on Neural Networks* 21.6 (2010), pp. 906–917 (cited on page 31).

[50] Anirban DasGupta. *Asymptotic Theory Of Statistics and Probability*. Springer Science & Business Media, 2008 (cited on page 31).

[51] Edward De Brouwer, Jaak Simm, Adam Arany, and Yves Moreau. "GRU-ODE-Bayes: Continuous Modeling of Sporadically-observed Time Series". In: *Neural Information Processing Systems (NeurIPS)*. 2019 (cited on pages 55–57, 60, 61, 65, 66, 93, 143, 145, 151).

[52] Bruno De Finetti. "La prévision: ses lois logiques, ses sources subjectives". In: *Annales de l'institut Henri Poincaré*. Vol. 7. 1. 1937, pp. 1–68 (cited on page 83).

[53] Angus Dempster, François Petitjean, and Geoffrey I. Webb. "ROCKET: Exceptionally Fast and Accurate Time Series Classification using Random Convolutional Kernels". In: *Data Min. Knowl. Discov.* 34.5 (2020) (cited on page 95).

[54] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "Imagenet: A Large-scale Hierarchical Image Database". In: *IEEE conference on computer vision and pattern recognition*. 2009 (cited on page 95).

[55] Ruizhi Deng, Marcus A Brubaker, Greg Mori, and Andreas Lehrmann. "Continuous Latent Process Flows". In: *Neural Information Processing Systems (NeurIPS)* 34 (2021) (cited on page 85).

[56] Ruizhi Deng, Bo Chang, Marcus A Brubaker, Greg Mori, and Andreas Lehrmann. "Modeling Continuous Stochastic Processes with Dynamic Normalizing Flows". In: *Neural Information Processing Systems (NeurIPS)* 33 (2020), pp. 7805–7815 (cited on pages 70, 73, 77, 85, 159).

[57] Prafulla Dhariwal and Alexander Quinn Nichol. "Diffusion Models Beat GANs on Image Synthesis". In: *Neural Information Processing Systems (NeurIPS)*. 2021 (cited on page 76).

[58] Laurent Dinh, David Krueger, and Yoshua Bengio. "NICE: Non-linear Independent Components Estimation". In: *International Conference on Learning Representations (ICLR) Workshop* (2015) (cited on page 17).

[59] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. "Density Estimation Using Real NVP". In: *International Conference on Learning Representations (ICLR)*. 2017 (cited on pages 17, 28, 58, 63, 125, 126, 147, 159).

[60] John R Dormand and Peter J Prince. "A family of embedded Runge-Kutta formulae". In: *Journal of computational and applied mathematics* 6.1 (1980), pp. 19–26 (cited on pages 7, 63, 148).

[61] Nan Du, Hanjun Dai, Rakshit Trivedi, Utkarsh Upadhyay, Manuel Gomez-Rodriguez, and Le Song. "Recurrent Marked Temporal Point Processes: Embedding Event History to Vector". In: *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016 (cited on pages 25–27, 30, 32–35, 39, 48, 49, 59, 61, 92, 122, 125, 128, 131).

[62] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. "Augmented Neural ODEs". In: *Neural Information Processing Systems (NeurIPS)*. 2019 (cited on page 59).

[63] Conor Durkan, Artur Bekasov, Iain Murray, and George Papamakarios. "Neural Spline Flows". In: *Neural Information Processing Systems (NeurIPS)*. 2019 (cited on page 18).

[64] Vincent Dutordoir, Alan Saul, Zoubin Ghahramani, and Fergus Simpson. "Neural Diffusion Processes". In: *arXiv 2206.03992* (2022) (cited on pages 84, 94).

[65] Cynthia Dwork. "Differential Privacy: A Survey of Results". In: *Theory and Applications of Models of Computation: 5th International Conference, TAMC 2008, Xi'an, China, April 25-29, 2008. Proceedings 5*. Springer. 2008, pp. 1–19 (cited on page 98).

[66] Emil Eirola and Amaury Lendasse. "Gaussian Mixture Models for Time Series Modelling, Forecasting, and Interpolation". In: *International Symposium on Intelligent Data Analysis*. Springer. 2013, pp. 162–173 (cited on page 33).

[67] Stefanos Eleftheriadis, Tom Nicholson, Marc Deisenroth, and James Hensman. "Identification of Gaussian Process State Space Models". In: *Neural Information Processing Systems (NeurIPS)*. 2017 (cited on page 49).

[68] Joseph Enguehard, Dan Busbridge, Adam Bozson, Claire Woodcock, and Nils Hammerla. "Neural Temporal Point Processes for Modelling Electronic Health Records". In: *Machine Learning for Health*. 2020 (cited on page 92).

[69] Dhivya Eswaran, Stephan Günnemann, and Christos Faloutsos. "The Power of Certainty: A Dirichlet-Multinomial Model for Belief Propagation". In: *SDM*. 2017, pp. 144–152 (cited on page 49).

[70] Patrick E Farrell, David A Ham, Simon W Funke, and Marie E Rognes. "Automated Derivation of the Adjoint of High-level Transient Finite Element Programs". In: *SIAM Journal on Scientific Computing* 35.4 (2013), pp. C369–C393 (cited on page 68).

[71] Chris Finlay, Jörn-Henrik Jacobsen, Levon Nurbekyan, and Adam M Oberman. "How to Train your Neural ODE". In: *International Conference on Machine Learning (ICML)*. 2020 (cited on pages 55, 68).

[72] Adriaan Daniël Fokker. "Die mittlere Energie rotierender elektrischer Dipole im Strahlungsfeld". In: *Annalen der Physik* 348.5 (1914), pp. 810–820 (cited on page 70).

[73] Meire Fortunato, Charles Blundell, and Oriol Vinyals. "Bayesian Recurrent Neural Networks". In: *arXiv 1704.02798* (2017) (cited on page 49).

[74] Sylvia Frühwirth-Schnatter. *Finite Mixture and Markov Switching Models*. Springer Science & Business Media, 2006 (cited on page 29).

[75] Iason Gabriel. "Artificial intelligence, values, and alignment". In: *Minds and machines* 30.3 (2020), pp. 411–437 (cited on page 98).

[76] Marta Garnelo, Jonathan Schwarz, Dan Rosenbaum, Fabio Viola, Danilo J Rezende, SM Eslami, and Yee Whye Teh. "Neural Processes". In: *ICML 2018 workshop on Theoretical Foundations and Applications of Deep Generative Models*. 2018 (cited on pages 73, 74, 83, 88).

[77] Carl Friedrich Gauss. *Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem Ambientium auctore Carolo Friderico Gauss*. Frid. Perthes et IH Besser, 1809 (cited on page 1).

[78] Felix A Gers and Jürgen Schmidhuber. “Recurrent Nets that Time and Count”. In: *International Joint Conference on Neural Networks (IJCNN)*. Vol. 3. IEEE. 2000 (cited on page 9).

[79] Amir Gholami, Kurt Keutzer, and George Biros. “ANODE: Unconditionally Accurate Memory-efficient Gradients for Neural ODEs”. In: *IJCAI*. 2019 (cited on page 68).

[80] Arnab Ghosh, Harkirat Singh Behl, Emilien Dupont, Philip HS Torr, and Vinay Namboodiri. “STEER: Simple Temporal Regularization For Neural ODEs”. In: *Neural Information Processing Systems (NeurIPS)*. 2020 (cited on pages 64, 68).

[81] Tilmann Gneiting and Adrian E Raftery. “Strictly Proper Scoring Rules, Prediction, and Estimation”. In: *Journal of the American Statistical Association* 102.477 (2007), pp. 359–378 (cited on page 87).

[82] Naman Goel, Mohammad Yaghini, and Boi Faltings. “Non-discriminatory machine learning through convex fairness criteria”. In: *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*. 2018, pp. 116–116 (cited on page 97).

[83] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. “PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals”. In: *Circulation* 101.23 (2000), e215–e220 (cited on pages 65, 152).

[84] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, 2016 (cited on page 5).

[85] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. “Generative Adversarial Nets”. In: *Neural Information Processing Systems (NeurIPS)* (2014) (cited on pages 16, 33, 76).

[86] Charles AE Goodhart. *Problems of Monetary Management: The UK Experience*. Springer, 1984 (cited on page 96).

[87] Henry Gouk, Eibe Frank, Bernhard Pfahringer, and Michael J Cree. “Regularisation of Neural Networks By Enforcing Lipschitz Continuity”. In: *Machine Learning* 110.2 (2021), pp. 393–416 (cited on page 57).

[88] Will Grathwohl, Ricky TQ Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. “FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models”. In: *International Conference on Learning Representations (ICLR)*. 2019 (cited on pages 146, 147).

[89] Will Grathwohl, Dami Choi, Yuhuai Wu, Geoffrey Roeder, and David Duvenaud. “Backpropagation through the Void: Optimizing Control Variates for Black-Box Gradient Estimation”. In: *International Conference on Learning Representations (ICLR)* (2018) (cited on page 127).

[90] Alex Graves. "Generating Sequences with Recurrent Neural Networks". In: *arXiv 1308.0850* (2013) (cited on page 33).

[91] Ruocheng Guo, Jundong Li, and Huan Liu. "INITIATOR: Noise-contrastive Estimation for Marked Temporal Point Process". In: *International Joint Conference on Artificial Intelligence*. 2018 (cited on page 33).

[92] Priyanka Gupta, Pankaj Malhotra, Jyoti Narwariya, Lovekesh Vig, and Gautam Shroff. "Transfer Learning for Clinical Time Series Analysis using Deep Neural Networks". In: *Journal of Healthcare Informatics Research* 4.2 (2020), pp. 112–137 (cited on page 96).

[93] Vinayak Gupta and Srikanta Bedathur. "ProActive: Self-attentive Temporal Point Process Flows for Activity Sequences". In: *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2022 (cited on page 92).

[94] Vinayak Gupta, Srikanta Bedathur, Sourangshu Bhattacharya, and Abir De. "Learning Temporal Point Processes with Intermittent Observations". In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2021 (cited on page 92).

[95] Vinayak Gupta, Srikanta Bedathur, and Abir De. "Learning Temporal Point Processes for Efficient Retrieval of Continuous Time Event Sequences". In: *AAAI Conference on Artificial Intelligence (AISTATS)*. Vol. 36. 2022 (cited on page 92).

[96] Eldad Haber and Lars Ruthotto. "Stable Architectures for Deep Neural Networks". In: *Inverse Problems* 34.1 (2017), p. 014004 (cited on page 68).

[97] Moritz Hardt, Eric Price, and Nati Srebro. "Equality of Opportunity in Supervised Learning". In: *Neural Information Processing Systems (NeurIPS)* (2016) (cited on page 97).

[98] Ramin Hasani, Mathias Lechner, Alexander Amini, Lucas Liebenwein, Aaron Ray, Max Tschaikowski, Gerald Teschl, and Daniela Rus. "Closed-form Continuous-time Neural Networks". In: *Nature Machine Intelligence* (2022), pp. 1–12 (cited on page 93).

[99] Alan G Hawkes. "Spectra of Some Self-Exciting and Mutually Exciting Point Processes". In: *Biometrika* 58.1 (1971), pp. 83–90 (cited on pages 14, 26, 33, 48, 92).

[100] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *IEEE CCVPR*. 2016 (cited on pages 6, 57).

[101] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups". In: *IEEE Signal processing magazine* 29.6 (2012), pp. 82–97 (cited on page 1).

[102] Jonathan Ho, Ajay Jain, and Pieter Abbeel. "Denoising Diffusion Probabilistic Models". In: *Neural Information Processing Systems (NeurIPS)* (2020) (cited on pages 24, 74, 75, 79, 81, 85, 155, 156).

[103] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780 (cited on pages 9, 48, 60).

[104] Samuel I Holt, Zhaozhi Qian, and Mihaela van der Schaar. "Neural Laplace: Learning Diverse Classes of Differential Equations in the Laplace Domain". In: *International Conference on Machine Learning (ICML)*. 2022 (cited on page 93).

[105] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. "Meta-learning in Neural Networks: A survey". In: *IEEE transactions on pattern analysis and machine intelligence* 44.9 (2021), pp. 5149–5169 (cited on page 83).

[106] Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville. "Neural Autoregressive Flows". In: *International Conference on Machine Learning (ICML)*. 2018, pp. 2078–2087 (cited on pages 28, 31).

[107] Hengguang Huang, Hao Wang, and Brian Mak. "Recurrent Poisson Process Unit for Speech Recognition". In: *AAAI Conference on Artificial Intelligence*. 2019 (cited on pages 26, 27).

[108] Priscillia Hunt, John S Hollywood, and Jessica M Saunders. *Evaluation of the Shreveport Predictive Policing Experiment*. Rand Corporation Santa Monica, 2014 (cited on page 98).

[109] Michael F Hutchinson. "A Stochastic Estimator of the Trace of the Influence Matrix for Laplacian Smoothing Splines". In: *Communications in Statistics-Simulation and Computation* 18.3 (1989) (cited on page 19).

[110] Rob Hyndman, Anne Koehler, Keith Ord, and Ralph Snyder. *Forecasting with exponential smoothing. The state space approach*. Springer Science & Business Media, 2008 (cited on page 59).

[111] Aapo Hyvärinen. "Estimation of Non-normalized Statistical Models by Score Matching". In: *Journal of Machine Learning Research* 6.4 (2005) (cited on page 22).

[112] Valerie Isham and Mark Westcott. "A Self-correcting Point Process". In: *Stochastic Processes and Their Applications* 8.3 (1979), pp. 335–347 (cited on pages 14, 26, 33, 48).

[113] Aleksei Grigorevich Ivakhnenko and Valentin Grigorevich Lapa. *Cybernetics and Forecasting Techniques*. American Elsevier Publishing Company, 1967 (cited on page 1).

[114] Jörn-Henrik Jacobsen, Arnold W. M. Smeulders, and Edouard Oyallon. "i-RevNet: Deep Invertible Networks". In: *International Conference on Learning Representations (ICLR)*. 2018 (cited on page 68).

[115] Herbert Jaeger and Harald Haas. "Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication". In: *science* 304.5667 (2004), pp. 78–80 (cited on page 95).

[116] Priyank Jaini, Kira A. Selby, and Yaoliang Yu. "Sum-of-Squares Polynomial Flow". In: *International Conference on Machine Learning (ICML)*. 2019 (cited on pages 28, 29, 31).

[117] Eric Jang, Shixiang Gu, and Ben Poole. "Categorical Reparameterization with Gumbel-softmax". In: *International Conference on Learning Representations (ICLR)* (2017) (cited on pages 30, 127).

[118] Junteng Jia and Austin R. Benson. "Neural Jump Stochastic Differential Equations". In: *Neural Information Processing Systems (NeurIPS)*. 2019 (cited on pages 56, 61, 62, 66).

[119] Alistair Johnson, Lucas Bulgarelli, Tom Pollard, Steven Horng, Leo Anthony Celi, and Roger Mark. "MIMIC-IV (version 1.0)". In: *PhysioNet* (2021) (cited on pages 65, 152).

[120] Alistair Johnson, Tom Pollard, Lu Shen, H Lehman Li-Wei, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G Mark. "MIMIC-III, a Freely Accessible Critical Care Database". In: *Scientific data* 3.1 (2016), pp. 1–9 (cited on pages 65, 152).

[121] Alexia Jolicoeur-Martineau, Ke Li, Rémi Piché-Taillefer, Tal Kachman, and Ioannis Mitliagkas. "Gotta go Fast when Generating Data with Score-Based Models". In: *arXiv 2105.14080* (2021) (cited on page 76).

[122] Fazle Karim, Somshubra Majumdar, and Houshang Darabi. "Adversarial attacks on time series". In: *IEEE transactions on pattern analysis and machine intelligence* 43.10 (2020) (cited on page 97).

[123] Kathan Kashiparekh, Jyoti Narwariya, Pankaj Malhotra, Lovekesh Vig, and Gautam Shroff. "Convtimenet: A Pre-trained Deep Convolutional Neural Network for Time Series Classification". In: *International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2019 (cited on page 96).

[124] Seyed Mehran Kazemi, Rishab Goel, Sepehr Eghbali, Janahan Ramanan, Jaspreet Sahota, Sanjay Thakur, Stella Wu, Cathal Smyth, Pascal Poupart, and Marcus Brubaker. "Time2vec: Learning a Vector Representation of Time". In: *arXiv 1907.05321* (2019) (cited on page 96).

[125] Jacob Kelly, Jesse Bettencourt, Matthew J. Johnson, and David Duvenaud. "Learning Differential Equations that are Easy to Solve". In: *Neural Information Processing Systems (NeurIPS)*. 2020 (cited on pages 55, 68).

[126] Gavin Kerrigan, Justin Ley, and Padhraic Smyth. "Diffusion Generative Models in Infinite Dimensions". In: *International Conference on Artificial Intelligence and Statistics (AISTATS)* (2023) (cited on page 94).

[127] Patrick Kidger, Ricky TQ Chen, and Terry Lyons. ""Hey, That's not an ODE": Faster ODE Adjoints via Seminorms". In: *International Conference on Machine Learning (ICML)*. 2021 (cited on pages 63, 68).

[128] Patrick Kidger, James Foster, Xuechen Li, and Terry J Lyons. "Neural SDEs as Infinite-dimensional GANs". In: *International Conference on Machine Learning (ICML)*. 2021, pp. 5453–5463 (cited on pages 70, 73, 85).

[129] Patrick Kidger and Terry Lyons. "Signatory: Differentiable Computations of the Signature and Logsignature Transforms, on both CPU and GPU". In: *International Conference on Learning Representations (ICLR)*. 2021 (cited on page 70).

[130] Patrick Kidger, James Morrill, James Foster, and Terry J. Lyons. "Neural Controlled Differential Equations for Irregular Time Series". In: *Neural Information Processing Systems (NeurIPS)*. 2020 (cited on page 92).

[131] Patrick Kidger, James Morrill, and Terry Lyons. "Generalised Interpretable Shapelets for Irregular Time Series". In: *arXiv 2005.13948* (2020) (cited on page 97).

[132] Hyunjik Kim, Andriy Mnih, Jonathan Schwarz, Marta Garnelo, S. M. Ali Eslami, Dan Rosenbaum, Oriol Vinyals, and Yee Whye Teh. "Attentive Neural Processes". In: *International Conference on Learning Representations (ICLR), (ICLR)*. 2019 (cited on page 88).

[133] Diederik Kingma, Tim Salimans, Ben Poole, and Jonathan Ho. "Variational Diffusion Models". In: *Neural Information Processing Systems (NeurIPS)*. 2021 (cited on page 76).

[134] Diederik P Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations (ICLR)*. 2015 (cited on pages 63, 129, 141, 153).

[135] Diederik P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. "Improved Variational Inference with Inverse Autoregressive Flow". In: *Neural Information Processing Systems (NeurIPS)*. 2016 (cited on pages 17, 18, 147).

[136] Diederik P Kingma and Max Welling. "Auto-encoding Variational Bayes". In: *International Conference on Learning Representations (ICLR)*. 2014 (cited on pages 83, 84).

[137] Jon Kleinberg, Sendhil Mullainathan, and Manish Raghavan. "Inherent Trade-offs in the Fair Determination of Risk Scores". In: *arXiv preprint arXiv:1609.05807* (2016) (cited on page 97).

[138] I. Kobyzev, S. Prince, and M. Brubaker. "Normalizing Flows: An Introduction and Review of Current Methods". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020) (cited on page 159).

[139] Zhifeng Kong, Wei Ping, Jiaji Huang, Kexin Zhao, and Bryan Catanzaro. "DiffWave: A Versatile Diffusion Model for Audio Synthesis". In: *International Conference on Learning Representations (ICLR)*. 2021 (cited on pages 76, 82).

[140] Alireza Koochali, Andreas Dengel, and Sheraz Ahmed. "If You Like It, GAN It—Probabilistic Multivariate Times Series Forecast with GAN". In: *Engineering Proceedings* 5.1 (2021) (cited on page 81).

[141] Alireza Koochali, Peter Schichtel, Andreas Dengel, and Sheraz Ahmed. "Random Noise vs. State-of-the-Art Probabilistic Forecasting Methods: A Case Study on CRPS-Sum Discrimination Ability". In: *Applied Sciences* 12.10 (2022), p. 5104 (cited on page 87).

[142] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Neural Information Processing Systems (NeurIPS)*. 2012 (cited on page 1).

[143] Srijan Kumar, Xikun Zhang, and Jure Leskovec. "Predicting Dynamic Embedding Trajectory in Temporal Interaction Networks". In: *ACM SIGKDD international conference on Knowledge discovery and data mining*. 2019 (cited on pages 33, 66, 128, 153).

[144] Hiroshi Kunita. "Stochastic Differential Equations and Stochastic Flows of Diffeomorphisms". In: *Ecole d'été de probabilités de Saint-Flour XII-1982*. Springer, 1984, pp. 143–303 (cited on page 70).

[145] Hiroshi Kunita. *Stochastic Flows and Jump-diffusions*. Springer, 2019 (cited on page 70).

[146] Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. "Artificial Neural Networks for Solving Ordinary and Partial Differential Equations". In: *IEEE transactions on neural networks* 9.5 (1998), pp. 987–1000 (cited on page 68).

[147] Guokun Lai, Wei-Cheng Chang, Yiming Yang, and Hanxiao Liu. "Modeling Long- and Short-Term Temporal Patterns with Deep Neural Networks". In: *ACM SIGIR Conference on Research & Development in Information Retrieval*. 2018 (cited on page 87).

[148] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. "Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles". In: *Neural Information Processing Systems (NeurIPS)*. 2015 (cited on page 49).

[149] Mathias Lechner and Ramin Hasani. "Learning Long-term Dependencies in Irregularly-sampled Time Series". In: *Neural Information Processing Systems (NeurIPS)*. 2020 (cited on page 60).

[150] Yann LeCun, Sumit Chopra, Raia Hadsell, M Ranzato, and Fujie Huang. "A Tutorial on Energy-based Learning". In: *Predicting structured data* (2006) (cited on page 22).

[151] Jin Sub Lee and Philip M Kim. "ProteinSGM: Score-based Generative Modeling for de novo Protein Design". In: *bioRxiv* (2022) (cited on page 76).

[152] John M Lee. *Introduction to Smooth Manifolds*. Springer, 2012 (cited on page 56).

[153] Qianxiao Li, Ting Lin, and Zuowei Shen. "Deep Learning via Dynamical Systems: An Approximation Perspective". In: *arXiv 1912.10382* (2019) (cited on page 59).

[154] Shiyang Li, Xiaoyong Jin, Yao Xuan, Xiyou Zhou, Wenhu Chen, Yu-Xiang Wang, and Xifeng Yan. "Enhancing the Locality and Breaking the Memory Bottleneck of Transformer on Time Series Forecasting". In: *Neural Information Processing Systems (NeurIPS)* (2019) (cited on page 92).

[155] Shuang Li, Shuai Xiao, Shixiang Zhu, Nan Du, Yao Xie, and Le Song. "Learning Temporal Point Processes via Reinforcement Learning". In: *Neural Information Processing Systems (NeurIPS)*. 2018 (cited on pages 27, 33).

[156] Xuechen Li, Ting-Kam Leonard Wong, Ricky TQ Chen, and David Duvenaud. "Scalable Gradients for Stochastic Differential Equations". In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2020 (cited on pages 73, 84, 85, 93).

[157] Yang Li, Nan Du, and Samy Bengio. "Time-Dependent Representation for Neural Event Sequence Prediction". In: *International Conference on Learning Representations (ICLR) Workshop*. 2018 (cited on page 48).

[158] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. "Fourier neural operator for parametric partial differential equations". In: *International Conference on Learning Representations (ICLR)*. 2021 (cited on pages 71, 94).

[159] Li Jianyu, Luo Siwei, Qi Yingjian, and Huang Yaping. "Numerical Solution of Differential Equations by Radial Basis Function Neural Networks". In: *International Joint Conference on Neural Networks (IJCNN)*. 2002 (cited on page 68).

[160] Qianli Liao and Tomaso Poggio. "Bridging the Gaps between Residual Learning, Recurrent Neural Networks and Visual Cortex". In: *arXiv 1604.03640* (2016) (cited on pages 6, 68).

[161] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, et al. "Elastic Machine Learning Algorithms in Amazon Sagemaker". In: *ACM SIGMOD International Conference on Management of Data*. 2020 (cited on page 63).

[162] Jae Hyun Lim, Nikola B Kovachki, Ricardo Baptista, Christopher Beckham, Kamyar Azizzadenesheli, Jean Kossaifi, Vikram Voleti, Jiaming Song, Karsten Kreis, Jan Kautz, et al. "Score-based Diffusion Models in Function Space". In: *arXiv 2302.07400* (2023) (cited on pages 94, 96).

[163] Hongzhou Lin and Stefanie Jegelka. "ResNet with One-neuron Hidden Layers is a Universal Approximator". In: *Neural Information Processing Systems (NeurIPS)*. 2018 (cited on page 59).

[164] Shizhan Liu, Hang Yu, Cong Liao, Jianguo Li, Weiyao Lin, Alex X Liu, and Schahram Dustdar. "Pyraformer: Low-complexity Pyramidal Attention for Long-Range Time Series Modeling and Forecasting". In: *International Conference on Learning Representations (ICLR)*. 2021 (cited on page 92).

[165] Yiping Lu, Aoxiao Zhong, Quanzheng Li, and Bin Dong. "Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations". In: *International Conference on Machine Learning (ICML)*. 2018 (cited on page 68).

[166] Scott M Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions". In: *Neural Information Processing Systems (NeurIPS)* (2017) (cited on page 97).

[167] Zhaoyang Lyu, Xudong Xu, Ceyuan Yang, Dahua Lin, and Bo Dai. "Accelerating Diffusion Models via Early Stop of the Diffusion Process". In: *arXiv 2205.12524* (2022) (cited on page 76).

[168] Wolfgang Maass, Thomas Natschläger, and Henry Markram. "Real-time Computing without Stable States: A new Framework for Neural Computation Based on Perturbations". In: *Neural computation* 14.11 (2002), pp. 2531–2560 (cited on page 95).

[169] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of machine learning research* 9.Nov (2008), pp. 2579–2605 (cited on page 36).

[170] Pankaj Malhotra, Vishnu TV, Lovekesh Vig, Puneet Agarwal, and Gautam Shroff. "TimeNet: Pre-trained Deep Recurrent Neural Network for Time Series Classification". In: *arXiv 1706.08838* (2017) (cited on page 96).

[171] Andrey Malinin and Mark Gales. "Predictive Uncertainty Estimation via Prior Networks". In: *Neural Information Processing Systems (NeurIPS)*. 2018 (cited on pages 47, 49, 52, 136).

[172] Arjun K Manrai, Birgit H Funke, Heidi L Rehm, Morten S Olesen, Bradley A Maron, Peter Szolovits, David M Margulies, Joseph Loscalzo, and Isaac S Kohane. "Genetic Misdiagnoses and the Potential for Health Disparities". In: *New England Journal of Medicine* 375.7 (2016), pp. 655–665 (cited on page 98).

[173] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015 (cited on page 6).

[174] Warren S McCulloch and Walter Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133 (cited on page 1).

[175] Patrick L. McDermott and Christopher K. Wikle. "Bayesian Recurrent Neural Network Models for Forecasting and Quantifying Uncertainty in Spatial-Temporal Data". In: *Entropy 21(2): 184 (2019)* (2019) (cited on page 49).

[176] Geoffrey McLachlan and David Peel. *Finite Mixture Models*. John Wiley & Sons, 2004 (cited on page 29).

[177] Andrew J Meade Jr and Alvaro A Fernandez. "The Numerical Solution of Linear Ordinary Differential Equations by Feedforward Neural Networks". In: *Mathematical and Computer Modelling* 19.12 (1994), pp. 1–25 (cited on page 68).

[178] Ninareh Mehrabi, Fred Morstatter, Nanyun Peng, and Aram Galstyan. "Debiasing Community Detection: The Importance of Lowly Connected Nodes". In: *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. 2019 (cited on page 97).

[179] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. "A Survey on Bias and Fairness in Machine Learning". In: *ACM Computing Surveys (CSUR)* 54.6 (2021), pp. 1–35 (cited on page 97).

[180] Hongyuan Mei and Jason M Eisner. "The Neural Hawkes Process: A Neurally Self-Modulating Multivariate Point Process". In: *Neural Information Processing Systems (NeurIPS)*. 2017 (cited on pages 25, 26, 32–34, 48, 49, 61, 92).

[181] Sebastian Meyer, Johannes Elias, and Michael Höhle. "A Space-time Conditional Intensity Model for Invasive Meningococcal Disease Occurrence". In: *Biometrics* 68.2 (2012), pp. 607–616 (cited on page 62).

[182] Shakir Mohamed, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. "Monte Carlo Gradient Estimation in Machine Learning". In: *arXiv 1906.10652* (2019) (cited on pages 30, 127, 133).

[183] Kristen Moore, Cody James Christopher, David Liebowitz, Surya Nepal, and Renee Selvey. "Modelling Direct Messaging Networks with Multiple Recipients for Cyber Deception". In: *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2022 (cited on page 92).

[184] James Morrill, Cristopher Salvi, Patrick Kidger, and James Foster. "Neural Rough Differential Equations for Long Time Series". In: *International Conference on Machine Learning (ICML)*. 2021 (cited on page 92).

[185] Marek Musiela and Marek Rutkowski. *Martingale Methods in Financial Modelling*. Springer, 2005 (cited on page 70).

[186] David B Mustard. "Reexamining Criminal Behavior: The Importance of Omitted Variable Bias". In: *Review of Economics and Statistics* 85.1 (2003), pp. 205–211 (cited on page 97).

[187] Akhila Narla, Brett Kuprel, Kavita Sarin, Roberto Novoa, and Justin Ko. "Automated Classification of Skin Lesions: From Pixels to Practice". In: *Journal of Investigative Dermatology* 138.10 (2018), pp. 2108–2110 (cited on page 96).

[188] Charlie Nash, Yaroslav Ganin, SM Ali Eslami, and Peter Battaglia. "Polygen: An autoregressive generative model of 3d meshes". In: *International Conference on Machine Learning (ICML)*. 2020 (cited on page 38).

[189] Radford Neal. "MCMC Using Hamiltonian Dynamics". In: *Handbook of Markov Chain Monte Carlo* (2010) (cited on page 22).

[190] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. "Phased LSTM: Accelerating Recurrent Network Training for Long or Event-Based Sequences". In: *Neural Information Processing Systems (NeurIPS)*. 2016 (cited on pages 39, 48, 59).

[191] Richard Ngo. "The Alignment Problem from a Deep Learning Perspective". In: *arXiv 2209.00626* (2022) (cited on page 98).

[192] Alexander Quinn Nichol and Prafulla Dhariwal. "Improved Denoising Diffusion Probabilistic Models". In: *International Conference on Machine Learning (ICML)*. 2021 (cited on pages 76, 89).

[193] Alexander Norcliffe, Cristian Bodnar, Ben Day, Nikola Simidjievski, and Pietro Liò. "On Second Order Behaviour in Augmented Neural ODEs". In: *Neural Information Processing Systems (NeurIPS)*. 2020 (cited on page 59).

[194] NVIDIA. *CUDA Programming Guide. Version 1.0*. 2007 (cited on page 1).

[195] Yosihiko Ogata and David Vere-Jones. "Inference for Earthquake Models: A Self-correcting Model". In: *Stochastic processes and their applications* 17.2 (1984), pp. 337–347 (cited on page 62).

[196] Maya Okawa, Tomoharu Iwata, Takeshi Kurashima, Yusuke Tanaka, Hiroyuki Toda, and Naonori Ueda. "Deep Mixture Point Processes: Spatio-temporal Event Prediction with Rich Contextual Information". In: (2019) (cited on page 33).

[197] Bernt Øksendal. *Stochastic Differential Equations*. Springer, 2003 (cited on page 70).

[198] Bernt Øksendal. *Stochastic Differential Equations: an Introduction with Applications*. Springer Science & Business Media, 2013 (cited on page 83).

[199] Takahiro Omi, Naonori Ueda, and Kazuyuki Aihara. "Fully Neural Network Based Model for General Temporal Point Processes". In: *Neural Information Processing Systems (NeurIPS)*. 2019 (cited on pages 25, 27, 32–34, 66, 123, 125, 127, 153).

[200] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. "Wavenet: A Generative Model for Raw Audio". In: *SSW*. 2016 (cited on pages 33, 59).

[201] Katharina Ott, Prateek Katiyar, Philipp Hennig, and Michael Tiemann. "ResNet After All? Neural ODEs and Their Numerical Solution". In: *International Conference on Learning Representations (ICLR)* (2021) (cited on pages 63, 148).

[202] George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. "Normalizing Flows for Probabilistic Modeling and Inference". In: *arXiv 1912.02762* (2019) (cited on pages 5, 19).

[203] George Papamakarios, Theo Pavlakou, and Iain Murray. "Masked Autoregressive Flow for Density Estimation". In: *Neural Information Processing Systems (NeurIPS)*. 2017 (cited on page 17).

[204] Giorgio Parisi. "Correlation Functions and Computer Simulations". In: *Nuclear Physics B* 180.3 (1981), pp. 378–384 (cited on page 22).

[205] Namyong Park, Fuchen Liu, Purvanshi Mehta, Dana Cristofor, Christos Faloutsos, and Yuxiao Dong. "Evokg: Jointly Modeling Event Time and Network Structure for Reasoning over Temporal Knowledge Graphs". In: *Fifteenth ACM International Conference on Web Search and Data Mining*. 2022, pp. 794–803 (cited on page 92).

[206] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "On the Difficulty of Training Recurrent Neural Networks". In: *International Conference on Machine Learning (ICML)*. 2013, pp. 1310–1318 (cited on page 8).

[207] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "PyTorch: An Imperative Style, High-performance Deep Learning Library". In: *Neural Information Processing Systems (NeurIPS)* (2019) (cited on pages 1, 6, 19, 124).

[208] Jakiw Pidstrigach, Youssef Marzouk, Sebastian Reich, and Sven Wang. "Infinite-Dimensional Diffusion Models for Function Spaces". In: *arXiv 2302.10130* (2023) (cited on page 94).

[209] Maria Laura Piscopo, Michael Spannowsky, and Philip Waite. "Solving Differential Equations with Neural Networks: Applications to the Calculation of Cosmological Phase Transitions". In: *Phys. Rev. D* (2019) (cited on page 68).

[210] VM Planck. "Über einen Satz der statistischen Dynamik und seine Erweiterung in der Quantentheorie". In: *Sitzungsberichte der Königlich-Preussischen Akademie der Wissenschaften zu Berlin* (1917) (cited on page 70).

[211] Michael Poli, Stefano Massaroli, Atsushi Yamashita, Hajime Asama, and Jinkyoo Park. "Hypersolvers: Toward Fast Continuous-Depth Models". In: *Neural Information Processing Systems (NeurIPS)*. 2020 (cited on page 68).

[212] Lev Semenovich Pontryagin. *Mathematical Theory of Optimal Processes*. CRC press, 1987 (cited on page 7).

[213] Joaquin Quiñonero-Candela and Carl Edward Rasmussen. "A Unifying View of Sparse Approximate Gaussian Process Regression". In: *Journal of Machine Learning Research* 6.65 (2005), pp. 1939–1959 (cited on page 89).

[214] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, Ali Ramadhan, and Alan Edelman. "Universal Differential Equations for Scientific Machine Learning". In: *arXiv 2001.04385* (2020) (cited on page 94).

[215] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. "Physics-informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations". In: *Journal of Computational physics* 378 (2019), pp. 686–707 (cited on pages 71, 94).

[216] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. "Hierarchical Text-Conditional Image Generation with CLIP Latents". In: *arXiv 2204.06125* (2022) (cited on page 76).

[217] Alberto Gil Couto Pimentel Ramos, Abhinav Mehrotra, Nicholas Donald Lane, and Sourav Bhattacharya. "Conditioning Sequence-to-sequence Networks with Learned Activations". In: *International Conference on Learning Representations (ICLR)*. 2022 (cited on page 89).

[218] Syama Sundar Rangapuram, Matthias W Seeger, Jan Gasthaus, Lorenzo Stella, Yuyang Wang, and Tim Januschowski. "Deep State Space Models for Time Series Forecasting". In: *Neural Information Processing Systems (NeurIPS)*. 2018 (cited on page 59).

[219] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2005 (cited on pages 42, 78).

[220] Jakob Gulddahl Rasmussen. "Temporal Point Processes: The Conditional Intensity Function". In: *Lecture Notes, Jan* (2011) (cited on pages 27, 123).

[221] Vibhor Rastogi and Suman Nath. "Differentially Private Aggregation of Distributed Time-series with Transformation and Encryption". In: *ACM SIGMOD International Conference on Management of data*. 2010 (cited on page 98).

[222] Kashif Rasul, Calvin Seward, Ingmar Schuster, and Roland Vollgraf. "Autoregressive Denoising Diffusion Models for Multivariate Probabilistic Time Series Forecasting". In: *International Conference on Machine Learning (ICML)*. 2021 (cited on pages 76, 81, 82).

[223] Kashif Rasul, Abdul-Saboor Sheikh, Ingmar Schuster, Urs M Bergmann, and Roland Vollgraf. "Multivariate Probabilistic Time Series Forecasting via Conditioned Normalizing Flows". In: *International Conference on Learning Representations (ICLR)*. 2021 (cited on pages 81, 87).

[224] Danilo Jimenez Rezende and Shakir Mohamed. "Variational Inference with Normalizing Flows". In: *International Conference on Machine Learning (ICML)* (2015) (cited on pages 18, 25, 33).

[225] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "" Why Should I Trust You?" Explaining the Predictions of Any Classifier". In: *ACM SIGKDD international conference on knowledge discovery and data mining*. 2016 (cited on page 97).

[226] Hippolyt Ritter, Aleksandar Botev, and David Barber. "A Scalable Laplace Approximation for Neural Networks". In: *International Conference on Learning Representations (ICLR)*. 2018 (cited on page 49).

[227] Christian P Robert, George Casella, and George Casella. *Monte Carlo Statistical Methods*. Vol. 2. Springer, 1999 (cited on page 20).

[228] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. "High-Resolution Image Synthesis with Latent Diffusion Models". In: *arXiv 2112.10752* (2021) (cited on pages 76, 89).

[229] Yulia Rubanova, Ricky T. Q. Chen, and David K Duvenaud. "Latent Ordinary Differential Equations for Irregularly-Sampled Time Series". In: *Neural Information Processing Systems (NeurIPS)*. Vol. 32. 2019 (cited on pages 56, 59, 60, 64, 65, 84, 85, 145, 150, 151).

[230] Peter Sadowski and Pierre Baldi. *Neural Network Regression with Beta, Dirichlet, and Dirichlet-Multinomial Outputs*. 2019 (cited on page 49).

[231] David Salinas, Michael Bohlke-Schneider, Laurent Callot, Roberto Medico, and Jan Gasthaus. "High-dimensional Multivariate Forecasting with Low-rank Gaussian Copula Processes". In: *Neural Information Processing Systems (NeurIPS)*. 2019 (cited on page 81).

[232] David Salinas, Valentin Flunkert, Jan Gasthaus, and Tim Januschowski. "DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks". In: *International Journal of Forecasting* 36.3 (2020), pp. 1181–1191 (cited on pages 1, 59, 81).

[233] Samira Samadi, Uthaipon Tantipongpipat, Jamie H Morgenstern, Mohit Singh, and Santosh Vempala. "The Price of Fair PCA: One Extra Dimension". In: *Neural Information Processing Systems (NeurIPS)* (2018) (cited on page 97).

[234] Simo Särkkä and Arno Solin. *Applied Stochastic Differential Equations*. Institute of Mathematical Statistics Textbooks. Cambridge University Press, 2019 (cited on pages 80, 157).

[235] Anton Maximilian Schäfer and Hans Georg Zimmermann. "Recurrent Neural Networks are Universal Approximators". In: *International Conference on Artificial Neural Networks*. Springer. 2006, pp. 632–640 (cited on page 31).

[236] Mona Schirmer, Mazin Eltayeb, Stefan Lessmann, and Maja Rudolph. "Modeling Irregular Time Series with Continuous Recurrent Units". In: *International Conference on Machine Learning (ICML)*. 2022 (cited on page 93).

[237] Udo Schlegel, Hiba Arnout, Mennatallah El-Assady, Daniela Oelke, and Daniel A Keim. "Towards a Rigorous Evaluation of XAI Methods on Time Series". In: *International Conference on Computer Vision Workshop (ICCVW)*. IEEE. 2019 (cited on page 97).

[238] Mike Schuster. "Better Generative Models for Sequential Data Problems: Bidirectional Recurrent Mixture Density Networks". In: *Neural Information Processing Systems (NeurIPS)*. 2000 (cited on page 33).

[239] Karishma Sharma, Yizhou Zhang, Emilio Ferrara, and Yan Liu. "Identifying Coordinated Accounts on Social Media Through Hidden Influence and Group Behaviours". In: *arXiv 2008.11308* (2020) (cited on page 92).

[240] Oleksandr Shchur, Marin Biloš, and Stephan Günnemann. "Intensity-Free Learning of Temporal Point Processes". In: *International Conference on Learning Representations (ICLR)*. 2020 (cited on pages 2, 5, 61, 66).

[241] Oleksandr Shchur, Nicholas Gao, Marin Biloš, and Stephan Günnemann. "Fast and Flexible Temporal Point Processes with Triangular Maps". In: *Neural Information Processing Systems (NeurIPS)*. 2020 (cited on pages 14, 20, 92).

[242] Oleksandr Shchur, Ali Caner Turkmen, Tim Januschowski, Jan Gasthaus, and Stephan Günnemann. "Detecting Anomalous Event Sequences with Temporal Point Processes". In: *Neural Information Processing Systems (NeurIPS)* (2021) (cited on page 14).

[243] Satya Narayan Shukla and Benjamin M Marlin. "Multi-time Attention Networks for Irregularly Sampled Time Series". In: *International Conference on Learning Representations (ICLR)*. 2021 (cited on pages 92, 96).

[244] Hava T Siegelmann and Eduardo D Sontag. "On the Computational Power of Neural Nets". In: *Workshop on Computational learning theory*. ACM. 1992 (cited on page 31).

[245] Ikaro Silva, George Moody, Daniel J Scott, Leo A Celi, and Roger G Mark. "Predicting In-hospital Mortality of ICU Patients: The Physionet/computing in Cardiology Challenge 2012". In: *2012 Computing in Cardiology*. IEEE. 2012, pp. 245–248 (cited on pages 64, 88, 151).

[246] Justin Sirignano and Konstantinos Spiliopoulos. "DGM: A Deep Learning Algorithm for Solving Partial Differential Equations". In: *Journal of computational physics* 375 (2018), pp. 1339–1364 (cited on page 93).

[247] Edward Snelson and Zoubin Ghahramani. "Sparse Gaussian Processes using Pseudo-inputs". In: *Neural Information Processing Systems (NeurIPS)*. 2006 (cited on page 49).

[248] Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. "Deep Unsupervised Learning using Nonequilibrium Thermodynamics". In: *International Conference on Machine Learning (ICML)*. 2015, pp. 2256–2265 (cited on pages 74, 75).

[249] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. "Machine learning models that remember too much". In: *ACM SIGSAC Conference on computer and communications security*. 2017 (cited on page 98).

[250] Yang Song, Prafulla Dhariwal, Mark Chen, and Ilya Sutskever. *Consistency Models*. 2023 (cited on page 93).

[251] Yang Song, Sahaj Garg, Jiaxin Shi, and Stefano Ermon. "Sliced Score Matching: A Scalable Approach to Density and Score Estimation". In: *Uncertainty in Artificial Intelligence*. 2020, pp. 574–584 (cited on page 22).

[252] Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. "Score-Based Generative Modeling through Stochastic Differential Equations". In: *International Conference on Learning Representations (ICLR)*. 2021 (cited on pages 23, 74, 76, 80, 85, 157).

[253] Markus Steinbach, Anshul Jindal, Mohak Chadha, Michael Gerndt, and Shajulin Benedict. "Tppfaas: Modeling Serverless Functions Invocations via Temporal Point Processes". In: *IEEE Access* 10 (2022) (cited on page 92).

[254] Nils Strodthoff and Claas Strodthoff. "Detecting and Interpreting Myocardial Infarction using Fully Convolutional Neural Networks". In: *Physiological measurement* 40.1 (2019) (cited on page 97).

[255] Yongbin Sun, Yue Wang, Ziwei Liu, Joshua Siegel, and Sanjay Sarma. "Pointgrow: Autoregressively Learned Point Cloud Generation with Self-attention". In: *IEEE WACV*. 2020 (cited on page 38).

[256] Harini Suresh and John Guttag. "A Framework for Understanding Sources of Harm Throughout the Machine Learning Life Cycle". In: *Equity and access in algorithms, mechanisms, and optimization*. 2021 (cited on page 97).

[257] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to Sequence Learning with Neural Networks". In: *Neural Information Processing Systems (NeurIPS)* 27 (2014) (cited on pages 1, 10).

[258] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. "Intriguing Properties of Neural Networks". In: *International Conference on Learning Representations (ICLR)*. 2014 (cited on page 97).

[259] Esteban G Tabak and Cristina V Turner. "A Family of Nonparametric Density Estimation Algorithms". In: *Communications on Pure and Applied Mathematics* 66.2 (2013), pp. 145–164 (cited on page 33).

[260] Behzad Tabibian, Isabel Valera, Mehrdad Farajtabar, Le Song, Bernhard Schölkopf, and Manuel Gomez-Rodriguez. "Distilling Information Reliability and Source Trustworthiness from Digital Traces". In: *International Conference on World Wide Web*. 2017, pp. 847–855 (cited on page 33).

[261] Matthew A Taddy, Athanasios Kottas, et al. "Mixture Modeling for Marked Poisson Processes". In: *Bayesian Analysis* 7.2 (2012) (cited on page 33).

[262] Yusuke Tashiro, Jiaming Song, Yang Song, and Stefano Ermon. "CSDI: Conditional Score-based Diffusion Models for Probabilistic Time Series Imputation". In: *Neural Information Processing Systems (NeurIPS)*. 2021 (cited on pages 84, 88, 161).

[263] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. "Deepmind Control Suite". In: *arXiv 1801.00690* (2018) (cited on page 64).

[264] Takeshi Teshima, Isao Ishikawa, Koichi Tojo, Kenta Oono, Masahiro Ikeda, and Masashi Sugiyama. "Coupling-based Invertible Neural Networks Are Universal Diffeomorphism Approximators". In: *Neural Information Processing Systems (NeurIPS)*. 2020 (cited on pages 19, 59).

[265] Takeshi Teshima, Koichi Tojo, Masahiro Ikeda, Isao Ishikawa, and Kenta Oono. "Universal Approximation Property of Neural Ordinary Differential Equations". In: *Neural Information Processing Systems (NeurIPS) 2020 Workshop on Differential Geometry meets Deep Learning*. 2020 (cited on pages 19, 59).

[266] The New York Times. *Coronavirus (Covid-19) Data in the United States*. 2020. URL: `https://github.com/nytimes/covid-19-data` (cited on pages 67, 153).

[267] George Tucker, Andriy Mnih, Chris J Maddison, John Lawson, and Jascha Sohl-Dickstein. "Rebar: Low-variance, Unbiased Gradient Estimates for Discrete Latent Variable Models". In: *Neural Information Processing Systems (NeurIPS)*. 2017 (cited on page 127).

[268] Ali Caner Türkmen, Yuyang Wang, and Alexander J Smola. "FastPoint: Scalable Deep Point Processes". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. 2019 (cited on page 33).

[269] Ryan Turner, Marc Deisenroth, and Carl Rasmussen. "State-Space Inference and Learning with Gaussian Processes". In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2010 (cited on page 49).

[270] U.S. Geological Survey. *Earthquake Catalogue (accessed May 15, 2021)*. 2020. URL: `https://earthquake.usgs.gov/earthquakes/search/` (cited on pages 67, 153).

[271] Utkarsh Upadhyay, Abir De, and Manuel Gomez Rodriguez. "Deep Reinforcement Learning of Marked Temporal Point Processes". In: *Neural Information Processing Systems (NeurIPS)*. 2018 (cited on pages 27, 30, 33, 122, 125, 130).

[272] Utkarsh Upadhyay and Manuel Gomez Rodriguez. *Temporal Point Processes. Lecture Notes for Human-Centered ML*. 2019 (cited on pages 32, 121).

[273] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Neural Information Processing Systems (NeurIPS)*. 2017 (cited on pages 6, 10, 11, 62, 82, 83, 159, 160).

[274] Clifford H Wagner. "Simpson's Paradox in Real Life". In: *The American Statistician* 36.1 (1982), pp. 46–48 (cited on page 97).

[275] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding". In: *International Conference on Learning Representations (ICLR)*. 2019 (cited on page 95).

[276] Zhiguang Wang, Weizhong Yan, and Tim Oates. "Time Series Classification from Scratch with Deep Neural Networks: A Strong Baseline". In: *International joint conference on neural networks (IJCNN)*. IEEE. 2017 (cited on page 97).

[277] E Weinan. "A Proposal on Machine Learning via Dynamical Systems". In: *Communications in Mathematics and Statistics* 5.1 (2017), pp. 1–11 (cited on pages 6, 68).

[278] Max Welling and Yee W Teh. "Bayesian Learning via Stochastic Gradient Langevin Dynamics". In: *International Conference on Machine Learning (ICML)*. 2011 (cited on page 22).

[279] Junfeng Wen, Negar Hassanpour, and Russell Greiner. "Weighted Gaussian Process for Estimating Treatment Effect". In: *Neural Information Processing Systems (NeurIPS)*. 2016 (cited on page 49).

[280] Andreas Wienke. *Frailty Models in Survival Analysis*. Chapman and Hall/CRC, 2010 (cited on page 122).

[281] Matthew O Williams, Ioannis G Kevrekidis, and Clarence W Rowley. "A Data-driven Approximation of The Koopman Operator: Extending Dynamic Mode Decomposition". In: *Journal of Nonlinear Science* 25 (2015) (cited on page 93).

[282] Ronald J Williams. "Simple Statistical Gradient-following Algorithms for Connectionist Reinforcement Learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256 (cited on page 126).

[283] Haixu Wu, Jiehui Xu, Jianmin Wang, and Mingsheng Long. "Autoformer: Decomposition Transformers with Auto-Correlation for Long-Term Series Forecasting". In: *Neural Information Processing Systems (NeurIPS)* (2021) (cited on page 92).

[284] Shuai Xiao, Mehrdad Farajtabar, Xiaojing Ye, Junchi Yan, Le Song, and Hongyuan Zha. "Wasserstein Learning of Deep Generative Point Process Models". In: *Neural Information Processing Systems (NeurIPS)*. 2017 (cited on page 33).

[285] Shuai Xiao, hongteng Xu, Junchi Yan, Mehrdad Farajtabar, Xiaokang Yang, Le Song, and Hongyuan Zha. "Learning Conditional Generative Models for Temporal Point Processes". In: *AAAI Conference on Artificial Intelligence*. 2018 (cited on page 33).

[286] Junchi Yan, Xin Liu, Liangliang Shi, Changsheng Li, and Hongyuan Zha. "Improving Maximum Likelihood Estimation of Temporal Point Process via Discriminative and Adversarial Learning". In: *International Joint Conference on Artificial Intelligence*. 2018 (cited on page 33).

[287] Chao-Han Huck Yang, Yun-Yun Tsai, and Pin-Yu Chen. "Voice2series: Reprogramming Acoustic Models for Time Series Classification". In: *International Conference on Machine Learning (ICML)*. 2021 (cited on page 96).

[288] Guandao Yang, Xun Huang, Zekun Hao, Ming-Yu Liu, Serge Belongie, and Bharath Hariharan. "Pointflow: 3d Point Cloud Generation with Continuous Normalizing Flows". In: *International Conference on Computer Vision (ICCV)*. 2019 (cited on page 38).

[289] Lexiang Ye and Eamonn Keogh. "Time Series Shapelets: A Novel Technique that Allows Accurate, Interpretable and Fast Classification". In: *Data mining and knowledge discovery* 22 (2011), pp. 149–182 (cited on page 97).

[290] Kyongmin Yeo, Igor Melnyk, Nam Nguyen, and Eun Kyung Lee. "DE-RNN: Forecasting the Probability Density Function of Nonlinear Time Series". In: *2018 IEEE International Conference on Data Mining (ICDM)*. 2018 (cited on page 49).

[291] TaeHo Yoon, Youngsuk Park, Ernest K Ryu, and Yuyang Wang. "Robust Probabilistic Time Series Forecasting". In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*. PMLR. 2022 (cited on page 97).

[292] Jiaxuan You, Rex Ying, Xiang Ren, William L Hamilton, and Jure Leskovec. "GraphRNN: Generating Realistic Graphs with Deep Auto-Regressive Models". In: *International Conference on Machine Learning (ICML)*. 2018 (cited on page 38).

[293] Baichuan Yuan, Xiaowei Wang, Jianxin Ma, Chang Zhou, Andrea L. Bertozzi, and Hongxia Yang. "Variational Autoencoders for Highly Multivariate Spatial Point Processes Intensities". In: *International Conference on Learning Representations (ICLR)*. 2020 (cited on page 38).

[294] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. "Deep Sets". In: *Neural Information Processing Systems (NeurIPS)*. 2017 (cited on page 83).

[295] Han Zhang, Xi Gao, Jacob Unterman, and Tom Arodz. "Approximation Capabilities of Neural ODEs and Invertible Residual Networks". In: *International Conference on Machine Learning (ICML)*. 2020, pp. 11086–11095 (cited on page 59).

[296] Yunhao Zhang and Junchi Yan. "Crossformer: Transformer Utilizing Cross-Dimension Dependency for Multivariate Time Series Forecasting". In: *International Conference on Learning Representations (ICLR)*. 2023 (cited on page 92).

[297] Jieyu Zhao, Tianlu Wang, Mark Yatskar, Vicente Ordonez, and Kai-Wei Chang. "Gender Bias in Coreference Resolution: Evaluation and Debiasing Methods". In: *arXiv 1804.06876* (2018) (cited on page 97).

[298] Weiming Zhi, Tin Lai, Lionel Ott, Edwin V. Bonilla, and Fabio Ramos. "Learning Efficient and Robust Ordinary Differential Equations via Invertible Neural Networks". In: *International Conference on Machine Learning (ICML)*. 2022 (cited on page 93).

[299] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. "Learning Deep Features for Discriminative Localization". In: *IEEE conference on computer vision and pattern recognition*. 2016 (cited on page 97).

[300] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. "Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting". In: *AAAI Conference on Artificial Intelligence*. 2021 (cited on page 92).

[301] Quan Zhou, Jakub Marecek, and Robert N Shorten. "Fairness in forecasting and learning linear dynamical systems". In: *AAAI Conference on Artificial Intelligence*. 2021 (cited on page 98).

[302] Juntang Zhuang, Nicha Dvornek, Xiaoxiao Li, Sekhar Tatikonda, Xenophon Papademetris, and James Duncan. "Adaptive Checkpoint Adjoint Method for Gradient Estimation in Neural ODE". In: *International Conference on Machine Learning (ICML)*. 2020 (cited on page 68).

[303] Zachary M Ziegler and Alexander M Rush. "Latent Normalizing Flows for Discrete Sequences". In: *International Conference on Machine Learning (ICML)* (2019) (cited on page 33).

[304] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. "Adversarial Attacks on Neural Networks for Graph Data". In: *ACM SIGKDD international conference on knowledge discovery & data mining*. 2018 (cited on page 97).

A Appendix for Chapter 3

A.1 Intensity function of flow and mixture models

CDF and conditional intensity function of proposed models. The cumulative distribution function (CDF) of a **normalizing flow** model can be obtained in the following way. If z has a CDF $Q(z)$ and $\tau = g(z)$, then the CDF $F(\tau)$ of τ is obtained as:

$$F(\tau) = Q(g^{-1}(\tau)).$$

Since for both SOSFlow and DSFlow we can evaluate g^{-1} in closed-form, $F(\tau)$ is also easy to compute. For the **log-normal mixture model**, CDF is by definition equal to:

$$F(\tau) = \sum_{k=1}^{K} w_k \Phi\left(\frac{\log \tau - \mu_k}{s_k}\right),$$

where $\Phi(\cdot)$ is the CDF of a standard normal distribution.

Given the conditional PDF and CDF, we can compute the conditional intensity $\lambda^*(t)$ and the cumulative intensity $\Lambda^*(\tau)$ for each model as

$$\lambda^*(t) = \frac{p^*(t - t_{i-1})}{1 - F^*(t - t_{i-1})}, \qquad \Lambda^*(\tau_i) := \int_0^{\tau_i} \lambda^*(t_{i-1} + s)ds = -\log(1 - F^*(\tau_i)),$$

where t_{i-1} is the arrival time of most recent event before t.

Merging two independent processes. We replicate the setup from Upadhyay and Rodriguez [272] and consider what happens if we merge two *independent* TPPs with intensity functions $\lambda_1^*(t)$ and $\lambda_2^*(t)$ (and respectively, cumulative intensity functions $\Lambda_1^*(\tau)$ and $\Lambda_2^*(\tau)$). According to [272], the intensity function of the new process is $\lambda^*(t) = \lambda_1^*(t) + \lambda_2^*(t)$. Therefore, the cumulative intensity function of the new process is

$$\begin{aligned}
\Lambda^*(\tau) &= \int_0^{\tau} \lambda^*(t_{i-1} + s)ds \\
&= \int_0^{\tau} \lambda_1^*(t_{i-1} + s)ds + \int_0^{\tau} \lambda_2^*(t_{i-1} + s)ds \\
&= \Lambda_1^*(\tau) + \Lambda_2^*(\tau).
\end{aligned}$$

Using the previous result, we can obtain the CDF of the merged process as

$$\begin{aligned} F^*(\tau) &= 1 - \exp(-\Lambda^*(\tau)) \\ &= 1 - \exp(-\Lambda_1^*(\tau) - \Lambda_2^*(\tau)) \\ &= 1 - \exp(\log(1 - F_1^*(\tau)) + \log(1 - F_2^*(\tau))) \\ &= 1 - (1 + F_1^*(\tau)F_2^*(\tau) - F_1^*(\tau) - F_2^*(\tau)) \\ &= F_1^*(\tau) + F_2^*(\tau) - F_1^*(\tau)F_2^*(\tau). \end{aligned}$$

The PDF of the merged process is obtained by simply differentiating the CDF w.r.t. τ.

This means that by using either normalizing flows or mixture distributions, and thus directly modeling PDF or CDF, we are not losing any benefits of the intensity parametrization.

A.2 Discussion of constant & exponential intensity models

Constant intensity model as exponential distribution. The conditional intensity function of the constant intensity model [271] is defined as $\lambda^*(t_i) = \exp(\boldsymbol{v}^T \boldsymbol{h}_i + b)$, where $\boldsymbol{h}_i \in \mathbb{R}^H$ is the history embedding produced by an RNN, and $b \in \mathbb{R}$ is a learnable parameter. By setting $c = \exp(\boldsymbol{v}^T \boldsymbol{h}_i + b)$, it's easy to see that the PDF of the constant intensity model $p^*(\tau) = c \exp(-c)$ corresponds to an exponential distribution.

Exponential intensity model as Gompertz distribution. PDF of a Gompertz distribution [280] is defined as:

$$p(\tau|\alpha, \beta) = \alpha \exp\left(\beta\tau - \frac{\alpha}{\beta}\exp(\beta t) + \frac{\alpha}{\beta}\right).$$

for $\alpha, \beta > 0$. The two parameters α and β define its shape and rate, respectively. For any choice of its parameters, Gompertz distribution is unimodal and light-tailed. The mean of the Gompertz distribution can be computed as $\mathbb{E}[\tau] = \frac{1}{\beta}\exp\left(\frac{\alpha}{\beta}\right)\mathrm{Ei}(-\frac{\alpha}{\beta})$, where $\mathrm{Ei}(z) = \int_{-z}^{\infty} \exp(-v)/v \; dv$ is the exponential integral function (that can be approximated numerically).

The conditional intensity function of the exponential intensity model [61] is defined as $\lambda^*(t_i) = \exp(w(t_i - t_{i-1}) + \boldsymbol{v}^T \boldsymbol{h}_i + b)$, where $\boldsymbol{h}_i \in \mathbb{R}^H$ is the history embedding produced by an RNN, and $\boldsymbol{v} \in \mathbb{R}^H, b \in \mathbb{R}, w \in \mathbb{R}_+$ are learnable parameters. By defining $d = \boldsymbol{v}^T \boldsymbol{h}_i + b$, we obtain the PDF of the exponential intensity model [61, Equation 12] as:

$$p(\tau|w, d) = \exp\left(w\tau + d - \frac{1}{w}\exp(w\tau + d) + \frac{1}{w}\exp(d)\right).$$

By setting $\alpha = \exp(d)$ and $\beta = w$ we see that the exponential intensity model is equivalent to a Gompertz distribution.

A.3 Discussion of the FullyNN model

Summary The main idea of the approach by [199] is to model the integrated conditional intensity function:

$$\Lambda^*(\tau) = \int_0^\tau \lambda^*(t_{i-1} + s)ds.$$

using a feedforward neural network with non-negative weights:

$$\Lambda^*(\tau) := f(\tau) = \zeta(\boldsymbol{W}^{(3)} \tanh(\boldsymbol{W}^{(2)} \tanh(\boldsymbol{W}^{(1)}\tau + \tilde{\boldsymbol{b}}^{(1)}) + \boldsymbol{b}^{(2)}) + \boldsymbol{b}^{(3)}), \tag{A.1}$$

where $\tilde{\boldsymbol{b}}^{(1)} = \boldsymbol{V}\boldsymbol{h} + \boldsymbol{b}^{(0)}$, $\boldsymbol{h} \in \mathbb{R}^H$ is the history embedding, $\boldsymbol{W}^{(1)} \in \mathbb{R}_+^{D\times 1}$, $\boldsymbol{W}^{(2)} \in \mathbb{R}_+^{D\times D}$, $\boldsymbol{W}^{(3)} \in \mathbb{R}_+^{1\times D}$ are non-negative weight matrices, and $\boldsymbol{V} \in \mathbb{R}^{D\times H}$, $\boldsymbol{b}^{(0)} \in \mathbb{R}^D$, $\boldsymbol{b}^{(2)} \in \mathbb{R}^D$, $\boldsymbol{b}^{(3)} \in \mathbb{R}$ are the remaining model parameters.

FullyNN as a normalizing flow Let $z \sim$ Exponential(1), that is

$$F(z) = 1 - \exp(-z), \qquad p(z) = \exp(-z).$$

We can view $f : \mathbb{R}_+ \to \mathbb{R}_+$ as a transformation that maps τ to z

$$z = f(\tau) \iff \tau = f^{-1}(z).$$

We can now use the change of variables formula to obtain the conditional CDF and PDF of τ. Alternatively, we can obtain the conditional intensity as:

$$\lambda^*(\tau) = \frac{\partial}{\partial\tau}\Lambda^*(\tau) = \frac{\partial}{\partial\tau}f(\tau),$$

and use the fact that $p^*(\tau_i) = \lambda^*(t_{i-1} + \tau_i)\exp\left(-\int_0^{\tau_i}\lambda^*(t_{i-1} + s)ds\right)$. Both approaches lead to the same conclusion:

$$F^*(\tau) = 1 - \exp(-f(\tau)), \qquad p^*(\tau) = \exp(-f(\tau))\frac{\partial}{\partial\tau}f(\tau).$$

However, the first approach also provides intuition on how to draw samples $\tilde{\tau}$ from the resulting distribution $p^*(\tau)$ — an approach known as the inverse method [220]

1. Sample $\tilde{z} \sim$ Exponential(1),
2. Obtain $\tilde{\tau}$ by solving $f(\tau) - \tilde{z} = 0$ for τ (using e.g. bisection method).

Similarly to other flow-based models, sampling from the FullyNN model cannot be done exactly and requires a numerical approximation.

Shortcomings of the FullyNN model

1. The PDF defined by the FullyNN model doesn't integrate to 1.

   By definition of the CDF, the condition that the PDF integrates to 1 is equivalent to $\lim_{\tau\to\infty} F^*(\tau) = 1$, which in turn is equivalent to $\lim_{\tau\to\infty} \Lambda^*(\tau) = \infty$. However, because of saturation of tanh activations (i.e. $\sup_{x\in\mathbb{R}} |\tanh(x)| = 1$) in Equation A.1

   $$\lim_{\tau\to\infty} \Lambda^*(\tau) = \lim_{\tau\to\infty} f(\tau) < \zeta\left(\sum_{d=1}^{D} |w_d^{(3)}| + b^{(3)}\right) < \infty$$

   . Therefore, the PDF doesn't integrate to 1.

2. The FullyNN model assigns a non-zero amount of probability mass to the $(-\infty, 0)$ interval, which violates the assumption that inter-event times are strictly positive.

   Since the inter-event times τ are assumed to be strictly positive almost surely, it must hold that $\text{Prob}(\tau \leq 0) = F^*(0) = 0$, or equivalently $\Lambda^*(0) = 0$. However, we can see that

   $$\Lambda^*(0) = f(0) = \zeta(\boldsymbol{W}^{(3)} \tanh(\boldsymbol{W}^{(2)} \tanh(\tilde{\boldsymbol{b}}^{(1)}) + \boldsymbol{b}^{(2)}) + \boldsymbol{b}^{(3)}) > 0,$$

   which means that the FullyNN model permits negative inter-event times.

A.4 Implementation details

A.4.1 Shared architecture

We implement SOSFlow, DSFlow and LogNormMix, together with baselines: RMTPP (Gompertz distribution), exponential distribution and a FullyNN model. All of them share the same pipeline, from the data preprocessing to the parameter tuning and model selection, differing only in the way we calculate $p^*(\tau)$. This way we ensure a fair evaluation. Our implementation uses Pytorch [207].

From arrival times t_i we calculate the inter-event times $\tau_i = t_i - t_{i-1}$. Since they can contain very large values, RNN takes log-transformed and centered inter-event time and produces $\boldsymbol{h}_i \in \mathbb{R}^H$. In case we have marks, we additionally input m_i — the index of the mark class from which we get mark embedding vector $\boldsymbol{m}_i$. In some experiments we use extra conditional information, such as metadata $\boldsymbol{y}_i$ and sequence embedding $\boldsymbol{e}_j$, where j is the index of the sequence.

As illustrated in Section 3.2.3 we generate the parameters $\boldsymbol{\theta}$ of the distribution $p^*(\tau_i)$ from $[\boldsymbol{h}_i||\boldsymbol{y}_i||\boldsymbol{e}_j]$ using an affine layer. We apply a transformation of the parameters to enforce the constraints, if necessary.

All decoders are implemented using a common framework relying on normalizing flows. By defining the base distribution $q(z)$ and the inverse transformation $(g_1^{-1} \circ \cdots \circ g_M^{-1})$ we can evaluate the PDF $p^*(\tau)$ at any τ, which allows us to train with maximum likelihood (Section 3.2.1).

A.4.2 Log-normal mixture

The log-normal mixture distribution is defined in Equation 3.7. We generate the parameters of the distribution $\boldsymbol{w} \in \mathbb{R}^K, \boldsymbol{\mu} \in \mathbb{R}^K, \boldsymbol{s} \in \mathbb{R}^K$ (subject to $\sum_k w_k = 1, w_k \geq 0$ and $s_k > 0$), using an affine transformation (Equation 3.11). The log-normal mixture is equivalent to the following normalizing flow model

$$\begin{aligned} z_1 &\sim \text{GaussianMixture}(\boldsymbol{w}, \boldsymbol{\mu}, \boldsymbol{s}) \\ z_2 &= az_1 + b \\ \tau &= \exp(z_2) \end{aligned}$$

By using the affine transformation $z_2 = az_1 + b$ before the exp transformation, we obtain a better initialization, and thus faster convergence. This is similar to the batch normalization flow layer [59], except that $b = \frac{1}{N}\sum_{i=1}^N \log \tau_i$ and $a = \sqrt{\frac{1}{N}\sum_{i=1}^N (\log \tau_i - b)}$ are estimated using the entire dataset, not using batches.

Forward direction samples a value from a Gaussian mixture, applies an affine transformation and applies exp. In the *backward* direction we apply log-transformation to an observed data, center it with an affine layer and compute the density under the Gaussian mixture.

A.4.3 Baselines

We implement FullyNN model [199] as described in Appendix A.3, using the official implementation as a reference[1]. The model uses feed-forward neural network with non-negative weights (enforced by clipping values at 0 after every gradient step). Output of the network is a cumulative intensity function $\Lambda^*(\tau)$ from which we can easily get intensity function $\lambda^*(\tau)$ as a derivative w.r.t. τ using automatic differentiation in Pytorch. We get the PDF as $p^*(\tau) = \lambda^*(\tau)\exp(-\Lambda^*(\tau))$.

We implement RMTPP / Gompertz distribution [61] and the exponential distribution [271] models as described in Appendix A.2.[2]

All of the above methods define the distribution $p^*(\tau)$. Since the inter-event times may come at very different scales, we apply a linear scaling $\tilde{\tau} = a\tau$, where $a = \frac{1}{N}\sum_{i=1}^N \tau_i$ is estimated from the data. This ensures a good initialization for all models and speeds up training.

A.4.4 Deep sigmoidal flow

A single layer of DSFlow model is defined as:

$$f_{\boldsymbol{\theta}}^{DSF}(x) = \sigma^{-1}\left(\sum_{k=1}^K w_k \sigma\left(\frac{x - \mu_k}{s_k}\right)\right),$$

[1] `https://github.com/omitakahiro/NeuralNetworkPointProcess`
[2] `https://github.com/musically-ut/tf_rmtpp`

with parameters $\boldsymbol{\theta} = \{\boldsymbol{w} \in \mathbb{R}^K, \boldsymbol{\mu} \in \mathbb{R}^K, \boldsymbol{s} \in \mathbb{R}^K\}$ (subject to $\sum_k w_k = 1, w_k \geq 0$ and $s_k > 0$). We obtain the parameters of each layer using Equation 3.11.

We define $p(\tau)$ through the inverse transformation $(g_1^{-1} \circ \cdots \circ g_M^{-1})$, as described in Section 3.2.1.

$$\begin{aligned} z_M &= g_M^{-1}(\tau) = \log \tau \\ &\vdots \\ z_m &= g_m^{-1}(z_{m+1}) = f_{\boldsymbol{\theta}_m}^{DSF}(z_{m+1}) \\ &\vdots \\ z_1 &= \sigma(z_2) \\ z_1 &\sim q_1(z_1) = \text{Uniform}(0, 1) \end{aligned}$$

We use the batch normalization flow layer [59] between every pair of consecutive layers, which significantly speeds up convergence.

A.4.5 Sum-of-squares polynomial flow

A single layer of SOSFlow model is defined as:

$$f^{SOS}(x) = a_0 + \sum_{k=1}^{K} \sum_{p=0}^{R} \sum_{q=0}^{R} \frac{a_{p,k} a_{q,k}}{p+q+1} x^{p+q+1}.$$

There are no constraints on the polynomial coefficients $\boldsymbol{a} \in \mathbb{R}^{(R+1)\times K}$. We obtain $\boldsymbol{a}$ similarly to Equation 3.11 as $\boldsymbol{a} = \boldsymbol{V_a c} + \boldsymbol{b_a}$, where $\boldsymbol{c}$ is the context vector.

We define $p(\tau)$ by through the inverse transformation $(g_1^{-1} \circ \cdots \circ g_M^{-1})$, as described in Section 3.2.1.

$$\begin{aligned} z_M &= g_M^{-1}(\tau) = \log \tau \\ &\vdots \\ z_m &= g_m^{-1}(z_{m+1}) = f_{\boldsymbol{\theta}_m}^{SOS}(z_{m+1}) \\ &\vdots \\ z_1 &= \sigma(z_2) \\ z_1 &\sim q_1(z_1) = \text{Uniform}(0, 1) \end{aligned}$$

Same as for DSFlow, we use the batch normalization flow layer between every pair of consecutive layers. When implementing SOSFlow, we used Pyro [20] for reference.

A.4.6 Reparameterization sampling

Using a log-normal mixture model allows us to sample with reparameterization (see also Section 2.2.3) which proves to be useful, e.g. when imputing missing data (Section 3.3.4).

In a score function estimator [282] given a random variable $x \sim p_\theta(x)$, where θ are parameters, we can compute $\nabla_\theta \mathbb{E}_{x\sim p_\theta(x)}[f(x)]$ as $\mathbb{E}_{x\sim p_\theta(x)}[f(x)\nabla_\theta \log p_\theta(x)]$. This is an unbiased estimator of the gradients but it often suffers from high variance. If the function f is differentiable, we can obtain an alternative estimator using the reparameterization trick: $\epsilon \sim q(\epsilon), x = g_\theta(\epsilon)$. Thanks to this reparameterization, we can compute $\nabla_\theta \mathbb{E}_{x\sim p_\theta(x)}[f(x)] = \mathbb{E}_{\epsilon\sim q(\epsilon)}[\nabla_\theta f(g_\theta(\epsilon))]$. Such reparameterization estimator typically has lower variance than the score function estimator [182]. In both cases, we estimate the expectation using Monte Carlo.

To sample with reparameterization from the mixture model we use the Straight-Through Gumbel Estimator [117]. We first obtain a relaxed sample $\boldsymbol{z}^* = \text{softmax}((\log \boldsymbol{w} + \boldsymbol{o})/T)$, where each o_i is sampled i.i.d. from a Gumbel distribution with zero mean and unit scale, and T is the temperature parameter. Finally, we get a one-hot sample $\boldsymbol{z} = \text{onehot}(\arg\max_k z_k^*)$. While a discrete $\boldsymbol{z}$ is used in the forward pass, during the backward pass the gradients will flow through the differentiable $\boldsymbol{z}^*$.

The gradients obtained by the Straight-Through Gumbel Estimator are slightly biased, which in practice doesn't have a significant effect on the model's performance. There exist alternatives [267, 89] that provide unbiased gradients, but are more expensive to compute.

A.5 Dataset statistics

A.5.1 Synthetic data

Synthetic data is generated according to Omi et al. [199] using well known point processes. We sample 64 sequences for each process, each sequence containing 1024 events.

- **Poisson.** Conditional intensity function for a homogeneous (or stationary) Poisson point process is given as $\lambda^*(t) = 1$. Constant intensity corresponds to exponential distribution.

- **Renewal.** A stationary process defined by a log-normal probability density function $p(\tau)$, where we set the parameters to be $\mu = 1.0$ and $\sigma = 6.0$. Sequences appear clustered.

- **Self-correcting.** Unlike the previous two, this point process depends on the history and is defined by a conditional intensity function $\lambda^*(t) = \exp(t - \sum_{t_i<t} 1)$. After every new event the intensity suddenly drops, inhibiting the future points. The resulting point patterns appear regular.

- **Hawkes.** We use a self-exciting point process with a conditional intensity function given as $\lambda^*(t) = \mu + \sum_{t_i<t} \sum_{j=1}^{M} \alpha_j \beta_j \exp(-\beta_j(t - t_i))$. As per Omi et al. [199], we create two different datasets: **Hawkes1** with $M = 1$, $\mu = 0.02$, $\alpha_1 = 0.8$ and $\beta_1 = 1.0$; and **Hawkes2** with $M = 2$, $\mu = 0.2$, $\alpha_1 = 0.4$, $\beta_1 = 1.0$, $\alpha_2 = 0.4$ and $\beta_2 = 20$. For the imputation experiment we use Hawkes1 to generate the data and remove some of the events.

A.5.2 Real-world data

In addition we use real-world datasets that are described bellow. Table A.1 shows their summary. All datasets have a large amount of unique sequences and the number of events per sequence varies a lot. Using marked temporal point processes to predict the type of an event is feasible for some datasets (e.g. when the number of classes is low), and is meaningless for other.

- **LastFM Celma [29].** The dataset contains sequences of songs that selected users listen over time. Artists are used as an event type.
- **Reddit.**[3] On this social network website users submit posts to subreddits. In the dataset, most active subreddits are selected, and posts from the most active users on those subreddits are recodered. Each sequence corresponds to a list of submissions a user makes. The data contains 984 unique subreddits that we use as classes in mark prediction.
- **Stack Overflow.**[4] Users of a question-answering website get rewards (called badges) over time for participation. A sequence contains a list of rewards for each user. Only the most active users are selected and only those badges that users can get more than once.
- **MOOC.** Contains the interaction of students with an online course system. An interaction is an event and can be of various types (97 unique types), e.g. watching a video, solving a quiz etc.
- **Wikipedia.** A sequence corresponds to edits of a Wikipedia page. The dataset contains most edited pages and users that have an activity (number of edits) above a certain threshold.
- **Yelp.**[5] We use the data from the review forum and consider the reviews for the 300 most visited restaurants in Toronto. Each restaurant then has a corresponding sequence of reviews over time.

A.6 Additional discussion of the experiments

A.6.1 Event time prediction using history

Detailed setup. Each dataset consists of multiple sequences of inter-event times. We consider 10 train/validation/test splits of the sequences (of sizes 60%/20%/20%). We train all model parameters by minimizing the negative log-likelihood (NLL) of the training sequences, defined as $\mathcal{L}_{time}(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N} \log p^*_{\boldsymbol{\theta}}(\tau_i)$. After splitting the data into the

[3] `https://github.com/srijankr/jodie/` [143]
[4] `https://archive.org/details/stackexchange` preprocessed according to Du et al. [61]
[5] `https://www.yelp.com/dataset/challenge`

Dataset name	Number of sequences	Number of events
LastFM	929	1268385
Reddit	10000	672350
Stack Overflow	6633	480414
MOOC	7047	396633
Wikipedia	1000	157471
Yelp	300	215146

Table A.1: Dataset statistics.

3 sets, we break down long training sequences into sequences of length at most 128. Optimization is performed using Adam [134] with learning rate 10^{-3}. We perform training using mini-batches of 64 sequences. We train for up to 2000 epochs (1 epoch = 1 full pass through all the training sequences). For all models, we compute the validation loss at every epoch. If there is no improvement for 100 epochs, we stop optimization and revert to the model parameters with the lowest validation loss.

We select hyperparameter configuration for each model that achieves the lowest average loss on the validation set. For each model, we consider different values of L_2 regularization strength $C \in \{0, 10^{-5}, 10^{-3}\}$. Additionally, for SOSFlow we tune the number of transformation layers $M \in \{1, 2, 3\}$ and for DSFlow $M \in \{1, 2, 3, 5, 10\}$. We have chosen the values of K such that the mixture model has approximately the same number of parameters as a 1-layer DSFlow or a 1-layer FullyNN model. More specifically, we set $K = 64$ for LogNormMix, DSFlow and FullyNN. We found all these models to be rather robust to the choice of K, as can be seen in Table A.2 for LogNormMix. For SOSFlow we used $K = 4$ and $R = 3$, resulting in a polynomial of degree 7 (per each layer). Higher values of R led to unstable training, even when using batch normalization.

Additional discussion. In this experiment, we only condition the distribution $p^*(\tau_i)$ on the history embedding $\boldsymbol{h}_i$. We don't learn sequence embeddings $\boldsymbol{e}_j$ since they can only be learned for the training sequences, and not fore the validation/test sets.

There are two important aspects related to the NLL loss values that we report. First, the absolute loss values can be arbitrarily shifted by rescaling the data. Assume, that we have a distribution $p(\tau)$ that models the distribution of τ. Now assume that we are interested in the distribution $q(x)$ of $x = a\tau$ (for $a > 0$). Using the change of variables formula, we obtain $\log q(x) = \log p(\tau) + \log a$. This means that by simply scaling the data we can arbitrarily offset the log-likelihood score that we obtain. Therefore, the absolute values of of the (negative) log-likelihood $\mathcal{L}$ for different models are of little interest — all that matters are the differences between them.

The loss values are dependent on the train/val/test split. Assume that model 1 achieves loss values $\mathcal{L}_1 = \{1.0, 3.0\}$ on two train/val/test splits, and model 2 achieves $\mathcal{L}_2 = \{2.0, 4.0\}$ on the same splits. If we first aggregate the scores and report the average $\hat{\mathcal{L}}_1 = 2.0 \pm 1.0$, $\hat{\mathcal{L}}_2 = 3.0 \pm 1.0$, it may seem that the difference between the two models is not significant. However, if we first compute the differences and then aggregate

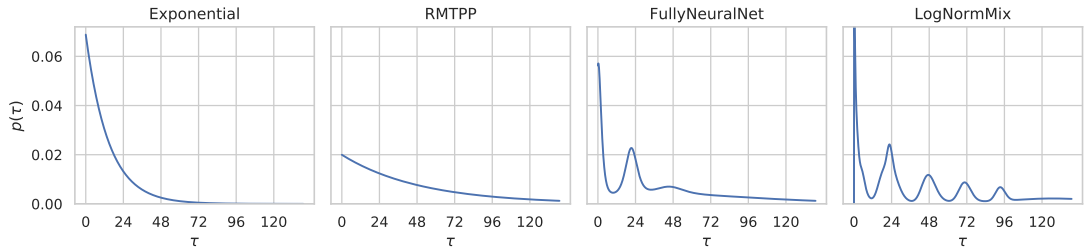


Figure A.1: Models learn different conditional distribution $p(\tau|\mathcal{H})$ on Yelp dataset. Since check-ins occur during the opening hours, true distribution of the next check-in resembles the one on the right.

$(\mathcal{L}_2 - \mathcal{L}_1) = 1.0 \pm 0.0$ we see a different picture. Therefore, we use the latter strategy in Figure 3.4. For completeness, we also report the numbers obtained using the first strategy in Table A.3.

As a baseline, we also considered the constant intensity / exponential distribution model [271]. However, we excluded the results for it from Figure 3.4, since it consistently achieved the worst loss values and had high variance. We still include the results for the constant intensity model in Table A.3. We also performed all the experiments on the synthetic datasets (Appendix A.5.1). The results are shown in Table A.4, together with NLL scores under the true model. We see that LogNormMix and DSFlow, besides achieving the best results, recover the true distribution.

Finally, in Figure A.1 we plot the conditional distribution $p(\tau|\mathcal{H})$ with models trained on Yelp dataset. The events represent check-ins into a specific restaurant. Since check-ins mostly happen during the opening hours, the inter-event time is likely to be on the same day (0h), next day (24h), the day after (48h), etc. LogNormMix can fully recover this behavior from data while others either cannot learn multimodal distributions (e.g., RMTPP) or struggle to capture it (e.g. FullyNN).

K	2	4	8	16	32	64
Reddit	10.239	10.208	10.189	10.185	10.191	10.192
LastFM	-2.828	-2.879	-2.881	-2.880	-2.877	-2.860
MOOC	6.246	6.053	6.055	6.055	6.050	5.660
Stack Overflow	14.461	14.438	14.435	14.435	14.436	14.428
Wikipedia	8.399	8.389	8.385	8.384	8.384	8.386
Yelp	13.169	13.103	13.058	13.045	13.032	13.024
Poisson	1.006	0.992	0.991	0.991	0.990	0.991
Renewal	0.256	0.254	0.254	0.254	0.256	0.259
Self-correcting	0.831	0.785	0.782	0.783	0.784	0.784
Hawkes1	0.530	0.523	0.532	0.532	0.523	0.523
Hawkes2	0.036	0.026	0.024	0.024	0.026	0.024

Table A.2: Performance of LogNormMix model for different numbers K of mixture components.

	Reddit	LastFM	MOOC	Stack Overflow	Wikipedia	Yelp
LogNormMix	**10.19 ± 0.078**	**-2.88 ± 0.147**	**6.03 ± 0.092**	**14.44 ± 0.013**	**8.39 ± 0.079**	**13.02 ± 0.070**
DSFlow	10.20 ± 0.074	**-2.88 ± 0.148**	**6.03 ± 0.090**	**14.44 ± 0.019**	8.40 ± 0.090	13.09 ± 0.065
SOSFlow	10.27 ± 0.106	-2.56 ± 0.133	6.27 ± 0.058	14.47 ± 0.049	8.44 ± 0.120	13.21 ± 0.068
FullyNN	10.23 ± 0.072	-2.84 ± 0.179	6.83 ± 0.152	14.45 ± 0.014	8.40 ± 0.086	13.04 ± 0.073
LogNormal	10.38 ± 0.077	-2.60 ± 0.140	6.53 ± 0.016	14.62 ± 0.013	8.52 ± 0.078	13.44 ± 0.074
RMTPP	10.88 ± 0.293	-1.30 ± 0.164	10.65 ± 0.023	14.51 ± 0.014	10.02 ± 0.085	13.36 ± 0.056
Exponential	11.07 ± 0.070	-1.28 ± 0.152	10.64 ± 0.026	18.48 ± 3.257	10.03 ± 0.083	13.78 ± 1.250

Table A.3: Time prediction test NLL on real-world data.

	Poisson	Renewal	Self-correcting	Hawkes1	Hawkes2
True model	0.999	0.254	0.757	0.453	-0.043
LogNormMix	**0.99 ± 0.006**	**0.25 ± 0.010**	**0.78 ± 0.003**	**0.52 ± 0.047**	**0.02 ± 0.049**
DSFlow	**0.99 ± 0.006**	**0.25 ± 0.010**	**0.78 ± 0.002**	**0.52 ± 0.047**	**0.02 ± 0.050**
SOSFlow	1.00 ± 0.013	**0.25 ± 0.010**	0.88 ± 0.011	0.59 ± 0.056	0.06 ± 0.046
FullyNN	1.00 ± 0.006	0.28 ± 0.013	**0.78 ± 0.004**	0.55 ± 0.047	0.06 ± 0.047
LogNormal	1.08 ± 0.008	**0.25 ± 0.010**	1.03 ± 0.006	0.55 ± 0.047	0.06 ± 0.049
RMTPP	**0.99 ± 0.006**	1.01 ± 0.023	**0.78 ± 0.003**	0.74 ± 0.057	0.69 ± 0.058
Exponential	**0.99 ± 0.006**	1.00 ± 0.023	0.94 ± 0.002	0.74 ± 0.055	0.69 ± 0.054

Table A.4: Time prediction test NLL on synthetic data.

A.6.2 Learning with marks

Detailed setup. We use the same setup as in Section A.6.1, except two differences. For learning in a marked temporal point process, we mimic the architecture from Du et al. [61]. The RNN takes a tuple (τ_i, m_i) as input at each time step, where m_i is the mark. Moreover, the loss function now includes a term for predicting the next mark:

$$\mathcal{L}_{total}(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N}\left[\log p^*_{\boldsymbol{\theta}}(\tau_i) + \log p^*_{\boldsymbol{\theta}}(m_i)\right].$$

The next mark m_i at time t_i is predicted using a categorical distribution $p^*(m_i)$. The distribution is parameterized by the vector $\boldsymbol{\pi}_i$, where $\pi_{i,c}$ is the probability of event $m_i = c$. We obtain $\boldsymbol{\pi}_i$ using the history embedding $\boldsymbol{h}_i$ passed through a feedforward neural network: $\boldsymbol{\pi}_i = \operatorname{softmax}\left(\boldsymbol{V}_{\boldsymbol{\pi}}^{(2)} \tanh(\boldsymbol{V}_{\boldsymbol{\pi}}^{(1)}\boldsymbol{h}_i + \boldsymbol{b}_{\boldsymbol{\pi}}^{(1)}) + \boldsymbol{b}_{\boldsymbol{\pi}}^{(2)}\right)$, where $\boldsymbol{V}_{\boldsymbol{\pi}}^{(1)}, \boldsymbol{V}_{\boldsymbol{\pi}}^{(2)}\boldsymbol{b}_{\boldsymbol{\pi}}^{(1)}, \boldsymbol{b}_{\boldsymbol{\pi}}^{(2)}$ are the parameters of the neural network.

Additional discussion. In Figure 3.4 (right) we reported the differences in time NLL between different models $\mathcal{L}_{time}(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N}\log p^*_{\boldsymbol{\theta}}(\tau_i)$. In Table A.5 we additionally provide the total NLL $\mathcal{L}_{total}(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N}\left[\log p^*_{\boldsymbol{\theta}}(\tau_i) + \log p^*_{\boldsymbol{\theta}}(m_i)\right]$ averaged over multiple splits. Using marks as input to the RNN improves time prediction quality for all the models. However, since we assume that the marks are conditionally independent of the time given the history (as was done in earlier works), all models have similar mark prediction accuracy.

	Time NLL		Total NLL		Mark accuracy	
	Reddit	MOOC	Reddit	MOOC	Reddit	MOOC
LogNormMix	**10.28 ± 0.066**	**5.75 ± 0.040**	12.40 ± 0.094	7.58 ± 0.047	0.62±0.014	0.45±0.003
DSFlow	**10.28 ± 0.073**	5.78 ± 0.067	**12.39 ± 0.064**	**7.52 ± 0.074**	0.62±0.013	0.45±0.004
SOSFlow	10.35 ± 0.106	6.06 ± 0.084	12.49 ± 0.158	7.78 ± 0.107	0.62±0.013	**0.46±0.009**
FullyNN	10.41 ± 0.079	6.22 ± 0.224	12.51 ± 0.094	7.93 ± 0.230	**0.63±0.013**	**0.46±0.004**
LogNormal	10.42 ± 0.076	6.38 ± 0.019	12.51 ± 0.080	8.11 ± 0.026	0.62±0.013	0.42±0.005
RMTPP	11.15 ± 0.061	10.29 ± 0.209	13.26 ± 0.085	12.14 ± 0.220	0.62±0.014	0.41±0.006

Table A.5: Time and total NLL and mark accuracy when learning a marked TPP.

A.6.3 Learning with additional conditional information

Detailed setup. In the Yelp dataset, the task is to predict the time τ_i until the next customer check-in, given the history of check-ins up until the current time t_{i-1}. We want to verify our intuition that the distribution $p^*(\tau_i)$ depends on the current time t_{i-1}. For example, $p^*(\tau_i)$ might be different depending on whether it's a weekday and / or it's an evening hour. Unfortunately, a model that processes the history with an RNN cannot easily obtain this information. Therefore, we provide this information directly as a context vector $\boldsymbol{y}_i$ when modeling $p^*(\tau_i)$.

The first entry of context vector $\boldsymbol{y}_i \in \{0, 1\}^2$ indicates whether the previous event t_{i-1} took place on a weekday or a weekend, and the second entry indicates whether t_{i-1} was in the 5PM–11PM time window. To each of the four possibilities we assign a learnable 64-dimensional embedding vector. The distribution of $p^*(\tau_i)$ until the next event depends on the embedding vector of the time stamp t_{i-1} of the most recent event.

A.6.4 Missing data imputation

Detailed setup. The dataset for the experiment is generated as a two step process: 1) We generate a sequence of 100 events from the model used for Hawkes1 dataset (Appendix A.5.1) resulting in a sequence of arrival times $\{t_1, \dots t_N\}$, 2) We choose random t_i and remove all the events that fall inside the interval $[t_i, t_{i+k}]$ where k is selected such that the interval length is approximately $t_N/3$.

We consider three strategies for learning with missing data (shown in Figure 3.5 (left)):

a) **No imputation**. The missing block spans the time interval $[t_i, t_{i+k}]$. We simply ignore the missing data, i.e. training objective $\mathcal{L}_{time}$ will include an inter-event time $\tau = t_{i+k} - t_i$.

b) **Mean imputation**. We estimate the average inter-event time $\hat{\tau}$ from the observed data, and impute events at times $\{t_i + n\hat{\tau} \quad \text{for} \quad n \in \mathbb{N}, \text{ such that } \; t_i + n\hat{\tau} < t_{i+k}\}$. These imputed events are fed into the history-encoding RNN, but are not part of the training objective.

c) **Sampling.** The RNN encodes the history up to and including t_i and produces $\boldsymbol{h}_i$ that we use to define the distribution $p^*(\tau|\boldsymbol{h}_i)$. We draw a sample $\tau_j^{(imp)}$ form this distribution and feed it into the RNN. We keep repeating this procedure

until the samples get past the point t_{i+k}. The imputed inter-event times $\tau_j^{(imp)}$ are affecting the hidden state of the RNN (thus influencing the likelihood of future observed inter-event times $\tau_i^{(obs)}$). We sample multiple such sequences in order to approximate the *expected* log-likelihood of the observed inter-event times $\mathbb{E}_{\tau^{(imp)} \sim p^*} \left[\sum_i \log p^*(\tau_i^{(obs)}) \right]$. Since this objective includes an expectation that depends on p^*, we make use of reparameterization sampling to obtain the gradients w.r.t. the distribution parameters [182].

A.6.5 Sequence embedding

Detailed setup. When learning sequence embeddings, we train the model as described in Appendix A.6.1, besides one difference. First, we pre-train the sequence embeddings $\boldsymbol{e}_j$ by disabling the history embedding $\boldsymbol{h}_i$ and optimizing $-\frac{1}{N} \sum_i \log p_{\boldsymbol{\theta}}(\tau_i | \boldsymbol{e}_j)$. Afterwards, we enable the history and minimize $-\frac{1}{N} \sum_i \log p_{\boldsymbol{\theta}}(\tau_i | \boldsymbol{e}_j, \boldsymbol{h}_i)$.

In Figure 3.7 the top row shows samples generated using $\boldsymbol{e}_{SC}$, embedding of a self-correcting sequence, the bottom row was generated using $\boldsymbol{e}_{SC}$, embedding of a renewal sequence, and the middle row was generated using $1/2(\boldsymbol{e}_{SC} + \boldsymbol{e}_{RN})$, an average of the two embeddings.

B Appendix for Chapter 4

B.1 Approximation of the uncertainty cross-entropy for WGP-LN

Given true categorical distribution $\boldsymbol{p}_i^*$, and predicted $\boldsymbol{p}_i(\tau)$, the uncertainty cross-entropy can be calculated as in Equation 4.7. For the WGP-LN model, $\boldsymbol{p}_i(\tau) = \text{softmax}(\boldsymbol{z}_i(\tau))$, where $\boldsymbol{z}_i(\tau)$ are logits that come from a Gaussian process and follow a normal distribution $\mathcal{N}(\boldsymbol{\mu}_i(\tau), \boldsymbol{\Sigma}_i(\tau))$,. Therefore, $\exp(\boldsymbol{z}_i(\tau))$ follows a log-normal distribution. We will use this to derive an approximation of the loss. From now on, we omit τ from the equations. The mean and the variance of $\sum_c^C \exp(\boldsymbol{z}_{c_i})$ are then:

$$\begin{aligned} \mathbb{E}\left[\sum\nolimits_c^C \exp(\boldsymbol{z}_{c_i})\right] &= \sum\nolimits_c^C \exp(\boldsymbol{\mu}_{c_i} + \boldsymbol{\sigma}_{c_i}^2/2), \\ \mathbf{Var}\left[\sum\nolimits_{c_i}^C \exp(\boldsymbol{z}_{c_i})\right] &= \sum\nolimits_{c_i}^C (\exp(\sigma_{c_i}^2) - 1)\exp(2\boldsymbol{\mu}_{c_i} + \boldsymbol{\sigma}_{c_i}^2). \end{aligned} \tag{B.1}$$

The expectation of the cross entropy loss given that the logits are following a normal distribution is:

$$\mathcal{L}_i^{\text{UCE}} = \mathbb{E}[\mathcal{L}_i^{\text{CE}}] = \mathbb{E}[\log(\exp(\boldsymbol{z}_{c_i}))] - \mathbb{E}\left[\log\left(\sum\nolimits_c^C \exp(\boldsymbol{z}_{c_i})\right)\right]. \tag{B.2}$$

In general, given a random variable x, we can approximate expectation of $\log x$ by performing a second order Taylor expansion around the mean μ:

$$\begin{aligned} \mathbb{E}[\log x] &\approx \mathbb{E}\Big[\log\mu + \underbrace{\frac{(\log\mu)'}{1!}(x-\mu)}_{\mathbb{E}[x-\mu]=0} + \frac{(\log\mu)''}{2!}(x-\mu)^2\Big] \\ &\approx \mathbb{E}[\log\mu] - \frac{\mathbf{Var}[x]}{2\mu^2}. \end{aligned} \tag{B.3}$$

Using Equation B.3 together with Equation B.1 and plugging into Equation B.2 we get a closed-form solution for the loss for event i:

$$\mathcal{L}_i^{\text{UCE}} \approx \mu_{c_i}(\tau_i^*) - \log\Big(\sum_c^C \exp(\mu_c(\tau_i^*) + \sigma_c^2(\tau_i^*)/2)\Big) + \frac{\sum_c^C (\exp(\sigma_c^2(\tau_i^*)) - 1)\exp(2\mu_c(\tau_i^*) + \sigma_c^2(\tau_i^*))}{2\Big(\sum_c^C \exp(\mu_c(\tau_i^*) + \sigma_c^2(\tau_i^*)/2)\Big)^2}. \tag{B.4}$$

B.2 Comparison of the classical cross-entropy and the uncertainty cross-entropy

B.2.1 Simple classification task

In this section, we do not consider temporal data. The goal of this experiment is to show the benefit of the uncertainty cross-entropy compared to a classical cross-entropy loss on a simple classification task. As a consequence, we use a simple two layer neural network to predict the concentration parameters of a Dirichlet distribution from the input vector.

Setup. The set-up is similar to Malinin and Gales [171] and consists of two datasets of 1500 instances divided in three equidistant 2-D Gaussians. One dataset contains non-overlapping classes (**NOG**) whereas the other contains overlapping classes (**OG**). We train simple two layer neural networks to predict the concentration parameters of a Dirichlet distribution $\mathbf{Dir}(\alpha_1(x_i), \alpha_2(x_i), \alpha_3(x_i))$ which captures the uncertainty on the categorical distribution $\boldsymbol{p}(x_i)$, for a data point x_i. On each dataset, we train two neural networks. One neural network is trained with the classic cross-entropy loss $\mathcal{L}^{\text{CE}}$ which uses only the mean prediction $\bar{\boldsymbol{p}}(x_i)$. The second neural network is trained with the uncertainty cross-entropy loss together with a simple α-regularizer:

$$\mathcal{L}^{\text{UCE}} + \big|\alpha_0(x_i) - \sum_j \mathbf{1}_{x_j \in N_w(x_i)}\big|, \tag{B.5}$$

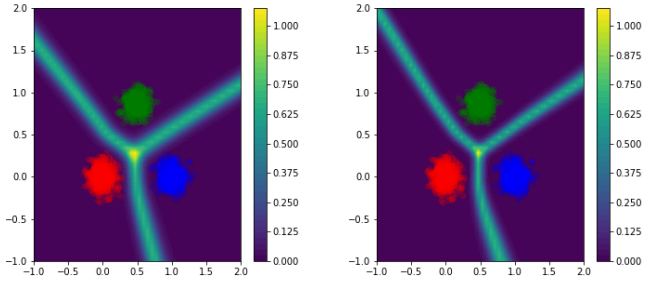
where x_i is the input 2-D vector and $N_w(x_i) = \{x', ||x' - x_i||_2^2 < w\}$ is its euclidean neighborhood of size w. We set $w = 10^{-5}$ for the non-overlapping Gaussians and $w = 10^{-2}$ for the overlapping Gaussians.

Results. The categorical entropy, equal to $-\sum_c p_c(x_i) \log p_c(x_i)$, is a good indicator that tells us how certain is the categorical distribution $\boldsymbol{p}(x_i)$ at point x_i. High entropy means that the categorical distribution is uncertain and vice versa. For non overlapping Gaussians shown in Figures B.1a and B.1b, we remark that both losses learn uncertain categorical distribution only on thin boundaries. However, for overlapping Gaussians as can be seen in Figures B.1c and B.1d, the uncertainty cross-entropy loss learns more uncertain categorical distributions due to its thicker boundaries.

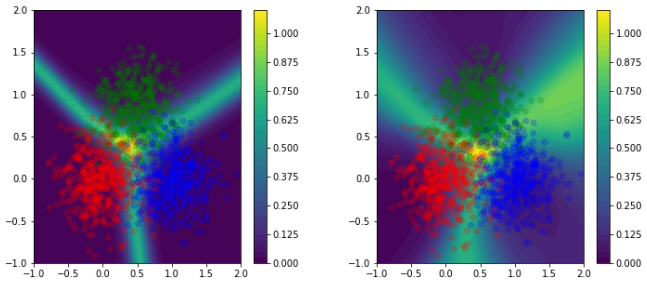
Another interesting result are the concentration parameters learned by the two models (Figure B.3 and Figure B.4). The classic cross-entropy loss learns very high values for $\alpha_1(x_i), \alpha_2(x_i), \alpha_3(x_i)$ which does match with the true distribution of the data. In contrast, the uncertainty cross-entropy learn meaningful alpha values for both datasets (delimiting the in-distribution areas for α_0 and centred around the classes for the others).

B.2.2 Irregularly-sampled Event Prediction

In this section, we consider temporal data. The goal of this experiment is again to show the benefit of the uncertainty cross-entropy compared to the classical cross-entropy in the case of irregular event prediction.



(a) NOG - CE **(b)** NOG - UCE



(c) OG - CE **(d)** OG - UCE

Figure B.1: Figures B.1a and B.1b plot the entropy of the categorical distribution learned on a classification task with three non-overlapping Gaussians. They show categorical entropy learned with the classical cross-entropy and with the uncertainty cross-entropy. Figures B.1c and B.1d plot the same for the dataset with three overlapping Gaussians.

Setup. For this purpose, we use the same set-up describe in the experiment with an anomaly detection and uncertainty. We trained the model FD-Dir with three different type of losses: (1) The classical cross-entropy (CE), (2) The classical cross-entropy with regularization described in section 4.1.3 (CE + reg) and (3) The classical uncertainty cross-entropy with regularization described in section 4.1.3 (UCE + reg).

Results. The results are shown in Figure B.2. The UCE loss with the regularization consistently improves the anomaly detection based on the distribution uncertainty.

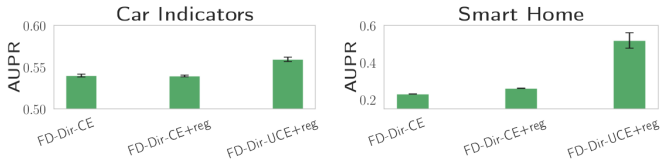
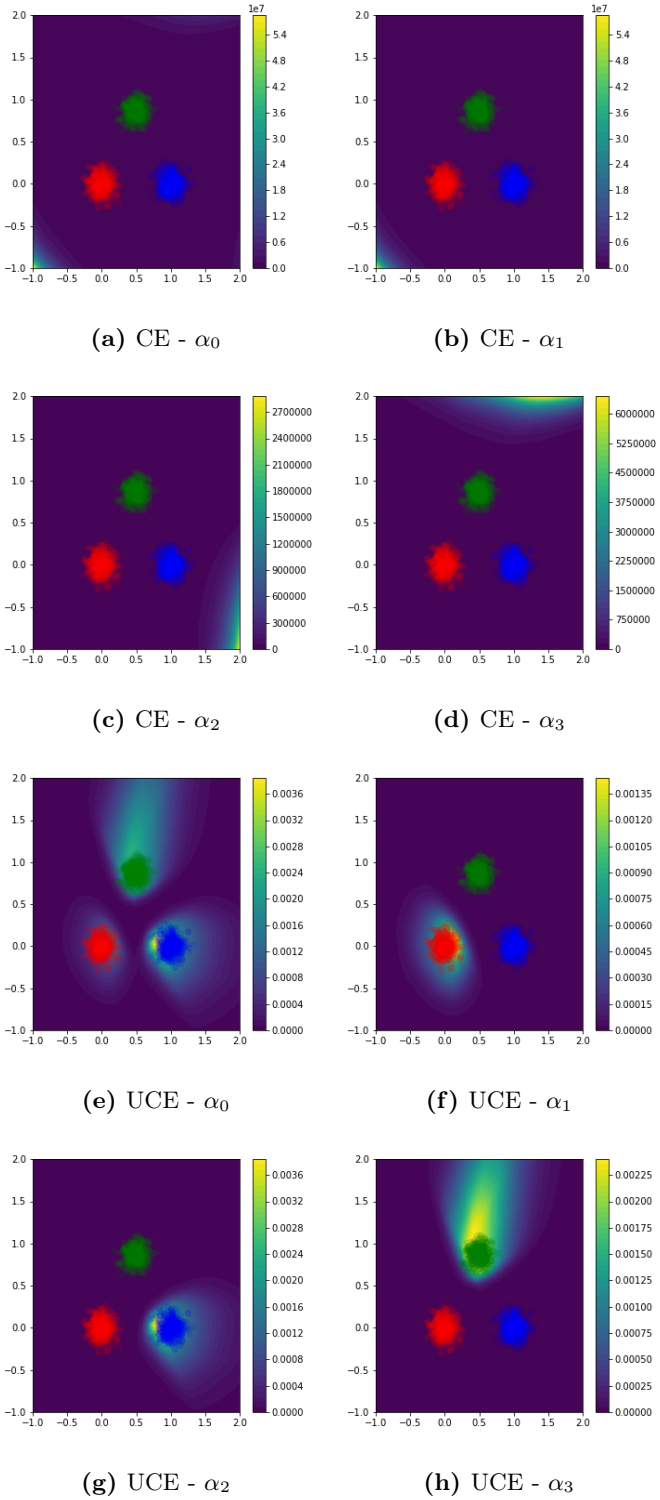


Figure B.2: Loss comparison in anomaly detection



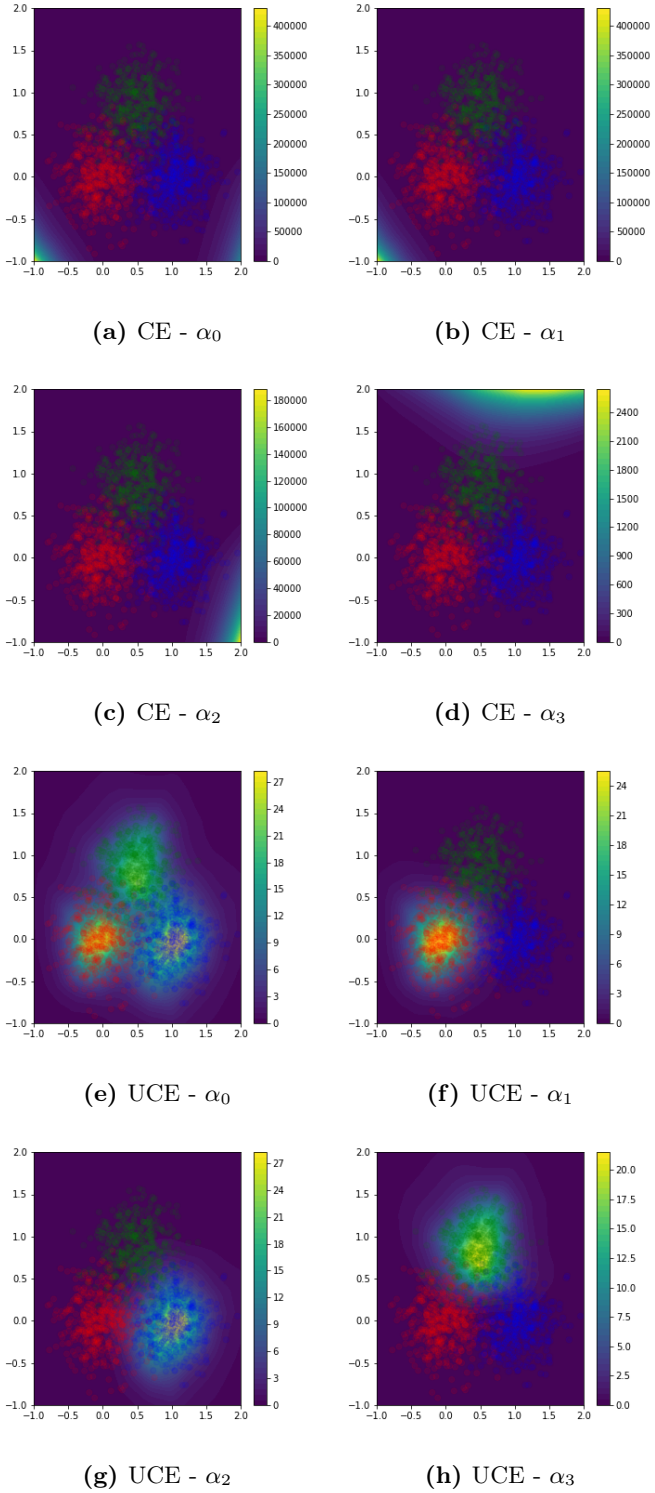
(a) CE - α_0 **(b)** CE - α_1

(c) CE - α_2 **(d)** CE - α_3

(e) UCE - α_0 **(f)** UCE - α_1

(g) UCE - α_2 **(h)** UCE - α_3

Figure B.3: Concentration parameters of the Dirichlet distribution on a classification task with three **non-overlapping** Gaussians. Figures B.3a, B.3b, B.3c, B.3d show α_0, α_1, α_2, α_3 learned with the classic cross-entropy. Figures B.3e, B.3f, B.3g, B.3h show α_0, α_1, α_2, α_3 learned with the uncertainty cross-entropy, respectively.



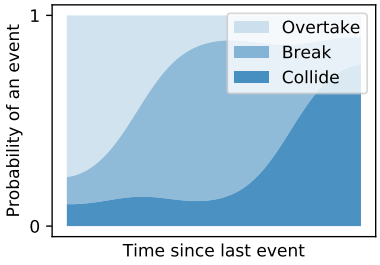
(a) CE - α_0 **(b)** CE - α_1

(c) CE - α_2 **(d)** CE - α_3

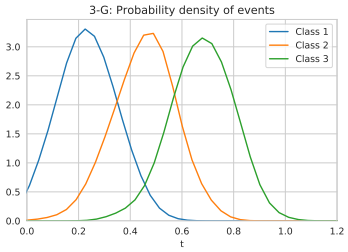
(e) UCE - α_0 **(f)** UCE - α_1

(g) UCE - α_2 **(h)** UCE - α_3

Figure B.4: Concentration parameters of the Dirichlet distribution on a classification task with three **overlapping** Gaussians. Figures B.4a, B.4b, B.3c, B.4d show α_0, α_1, α_2, α_3 learned with the classic cross-entropy. Figures B.4e, B.4f, B.3g, B.4h show α_0, α_1, α_2, α_3 learned with the uncertainty cross-entropy, respectively.



(a) Illustration of a car example, from the introduction of Chapter 4, where probabilities of the events change over time.



(b) Probability density of events in 3-G dataset. We can see that classes are independent of history.

Figure B.5: The motivating example (a) and the resulting synthetic dataset (b).

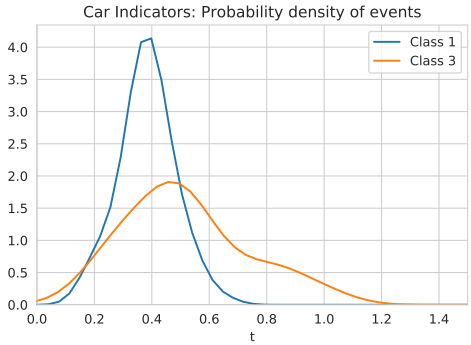


Figure B.6: Probability density of events in Car Indicators dataset for 2 selected classes. Time is log-transformed.

B.3 Datasets

In this section we describe the datasets in more detail. For the preprocessing step, the time gap between the two events $\tau_i^* = t_i - t_{i-1}$ is first log-transformed before applying min-max normalization: $\hat{\tau_i}^* = \frac{\tau_i' - \min(\tau_i^{*'})}{(\max(\tau_i^{*'}) - \min(\tau_i^{*'})}$ with $\tau_i^{*'} = \log(\tau_i^* + \epsilon)$, $\epsilon > 0$.

3-G. We set the number of classes $C = 3$ and draw from a normal distribution $P(\tau|c_i) = \mathcal{N}(i+1, 1.)$. This dataset tries to imitate the setting from Figure B.5a as explained in the introduction. We generate 1000 events. Probability density is shown in Figure B.5b. **Multi-G** dataset is created similarly.

Car Indicators. A sequence contains signals from a single car during one ride. We remove signals that are perfectly correlated giving 6 unique classes in the end. Top 3 classes make up 33%, 32%, and 16% of a total respectively. From Figure B.6 we can see that the setting is again irregular.

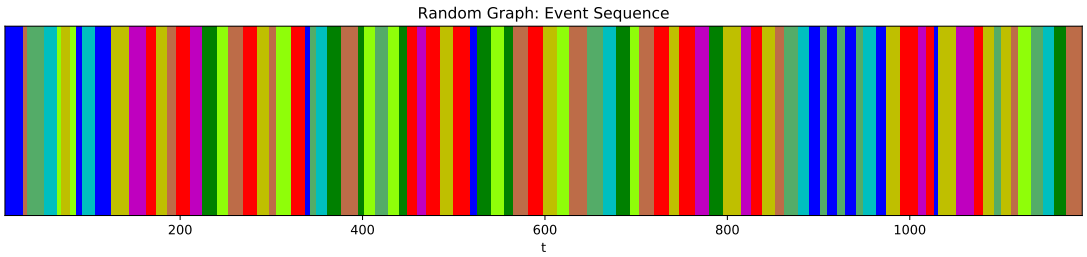


Figure B.7: Trace of events for a random graph. Different colors represent different classes and the width of a single column represents the time that has passed.

Graph. We generate graph G with 10 nodes and 48 edges between them. We assign variables μ and σ to each transition (edge) between events (nodes). The time it takes to make a transition between nodes i and j is drawn from normal distribution $\mathcal{N}(\mu_{ij}, \sigma_{ij}^2)$. By performing a random walk on the graph we create 10 thousand events. This dataset is similar to 3-G with the difference that a model needs to learn the relationship between events together with the time dependency. Parts of the trace are shown in Figure B.7.

B.4 Details of experiments

We test our models (WGP-LN, FD-Dir and FD-Dir-PP) against neural point process models (RMTPP and Hawkes) and simple baselines (RNN and LSTM—getting only history as an input, F-RNN and F-LSTM—having also the real time of the next event as an additional input; thus, they get a strong advantage!). We test on real-world (Stack Exchange, Car Indicators and Smart Home) and synthetic datasets (Graph). We show that our models consistently outperform all the other models when evaluated with class prediction accuracy and Time-Error.

B.4.1 Model selection

We apply the same tuning technique to all models. We split all datasets into train–validation–test sets ($60\% - 20\% - 20\%$), use the validation set to select a model and the test set to get the final scores. For Stack Exchange dataset we split on users. In all other datasets we split the trace based on time. We search over dimension of a hidden state $\{32, 64, 128, 256\}$, batch size $\{16, 32, 64\}$ and L_2 regularization parameter $\{0, 10^{-3}, 10^{-2}, 10^{-1}\}$. We use the same learning rate 0.001 for all models using an Adam optimizer [134], run each of them 5 times for maximum of 100 epochs with early stopping after 5 consecutive epochs without improvement in the validation loss. For the number of points M we pick 3 for WGP-LN and 20 for FD-Dir. WGP-LN and FD-Dir have additional regularization (Equation 4.10) with hyperparameters α and β. For both models we choose $\alpha = \beta = 10^{-3}$. Model with the highest mean accuracy on the validation set is selected. We use GRU cell [38] for both of our models.

	Car Indicators	Graph	Smart Home	Stack Exchange
FD-Dir	0.909 ± 0.005	**0.701 ± 0.002**	**0.522 ± 0.013**	**0.522 ± 0.001**
Dir-PP	**0.912 ± 0.006**	0.691 ± 0.006	0.415 ± 0.054	0.515 ± 0.002
WGP-LN	0.877 ± 0.010	0.685 ± 0.005	0.500 ± 0.017	0.519 ± 0.003
Hawkes	0.834 ± 0.022	0.585 ± 0.008	0.435 ± 0.017	0.513 ± 0.001
RMTPP	0.858 ± 0.004	0.257 ± 0.005	0.472 ± 0.016	0.492 ± 0.000
F-LSTM	0.855 ± 0.006	0.657 ± 0.002	0.411 ± 0.029	-
F-RNN	0.849 ± 0.013	0.615 ± 0.011	0.472 ± 0.035	-
LSTM	0.858 ± 0.010	0.251 ± 0.008	0.375 ± 0.026	-
RNN	0.838 ± 0.016	0.258 ± 0.008	0.437 ± 0.017	-

Table B.1: Class accuracy comparison for all models on all datasets

	Car Indicators	Graph	Smart Home	Stack Exchange
FD-Dir	**0.115 ± 0.040**	**0.101 ± 0.001**	**0.111 ± 0.011**	0.289 ± 0.019
WGP-LN	0.184 ± 0.047	0.120 ± 0.008	0.127 ± 0.010	**0.077 ± 0.016**
FD-Dir-PP	0.132 ± 0.031	0.106 ± 0.004	0.143 ± 0.022	0.375 ± 0.007
Hawkes	0.412 ± 0.091	0.158 ± 0.005	0.170 ± 0.035	0.507 ± 0.003
RMTPP	0.860 ± 0.004	0.257 ± 0.005	0.474 ± 0.016	0.721 ± 0.001
F-LSTM	0.277 ± 0.118	0.141 ± 0.002	0.209 ± 0.023	-
F-RNN	0.516 ± 0.105	0.146 ± 0.004	0.186 ± 0.011	-
LSTM	0.860 ± 0.010	0.251 ± 0.008	0.376 ± 0.026	-
RNN	0.841 ± 0.016	0.258 ± 0.008	0.439 ± 0.017	-

Table B.2: Time-Error comparison for all models on all datasets

Tables B.1 and B.2 show test results for *all* models on *all* datasets for Class accuracy and Time-Error.

B.4.2 Time prediction with point processes

TPP framework allows estimating the time $\hat{\tau}$ of the next event:

$$\hat{\tau} = \int_0^\infty t q(\tau) dt, \tag{B.6}$$

where $q(\tau) = \lambda_0(\tau) \exp\left(-\int_0^\tau \lambda_0(s) ds\right)$.

The usual way to evaluate the quality of this prediction is with a squared error. As we have already discussed in Section 4.3.3, this is not optimal for our use case. Nevertheless, we did compare our **FD-Dir-PP** model to RMTPP. On Car Indicators dataset our model has the mean MSE of 0.4783 while RMTPP achieves 0.4736. At the same time FD-Dir-PP outperforms RMTPP on other tasks (see Section 4.3).

C Appendix for Chapter 5

C.1 Theoretical background

C.1.1 Training loss for GRU-ODE-Bayes

De Brouwer et al. [51] define an objective that mimics the Bayesian filtering. It consists of two parts:

$$\mathcal{L} = \mathcal{L}_{\text{pre}} + \lambda \mathcal{L}_{\text{post}}, \tag{C.1}$$

where $\mathcal{L}_{\text{pre}}$ is masked negative log-likelihood and $\mathcal{L}_{\text{post}}$ is the Bayesian part of the loss. The model outputs the normal distribution for the observations, conditional on hidden state $\boldsymbol{h}(t)$. Since only some features are observed at a time, we mask out the missing values when calculating $\mathcal{L}_{\text{pre}}$. We denote our predicted distribution with p_{pre}, and predicted distribution after updating the state with p_{post}. Now, the Bayesian update can be written as $p_{\text{Bayes}} \propto p_{\text{pre}} \cdot p_{\text{obs}}$, with p_{obs} being the noise of the observations. $\mathcal{L}_{\text{post}}$ is defined as a KL-divergence between p_{Bayes} and p_{post}. This can be calculated in closed-form for normal distribution.

C.1.2 Proof of Theorem 2

Preliminaries. Function f has the Lipschitz constant L if $|f(x) - f(y)| \leq L|x - y|$, $\forall x, y$. We first derive a few useful inequalities.

For the sum of two Lipschitz functions $f + g$, the following holds:

$$\begin{aligned} |f(x) + g(x) - f(y) - g(y)| &\leq |f(x) - f(y)| + |g(x) - g(y)| \\ &\leq \text{Lip}(f)|x - y| + \text{Lip}(g)|x - y| \\ &= (\text{Lip}(f) + \text{Lip}(g))|x - y|, \end{aligned} \tag{C.2}$$

by the triangle inequality and the definition of the Lipschitz function. Similarly, for the product of two Lipschitz functions $f \cdot g$, the following holds:

$$\begin{aligned} |f(x)g(x) - f(y)g(y)| &= |f(x)g(x) + f(x)g(y) - f(x)g(y) - f(y)g(y)| \\ &= |f(x)(g(x) - g(y)) + g(y)(f(x) - f(y))| \\ &\leq |f(x)||g(x) - g(y)| + |g(y)||f(x) - f(y)| \\ &\leq |f(x)| \cdot \text{Lip}(g) \cdot |x - y| + |g(y)| \cdot \text{Lip}(f) \cdot |x - y| \\ &= (|f(x)| \cdot \text{Lip}(g) + |g(y)| \cdot \text{Lip}(f))|x - y|. \end{aligned} \tag{C.3}$$

If f and g are bounded, we can bound the above terms as well.

Let f be contractive function, $\text{Lip}(f) < 1$. Then, for the composition of functions $\sigma \circ f$, where $\sigma(x) = (1 + \exp(-x))^{-1}$ is the sigmoid activation, the following holds:

$$|\sigma(f(x)) - \sigma(f(y))| \leq \text{Lip}(\sigma)|f(x) - f(y)| = \frac{1}{4}|f(x) - f(y)| < \frac{1}{4}|x - y|,$$

where we used $\text{Lip}(\sigma) = \max(\sigma') = \frac{1}{4}$, by the mean value theorem. Similarly, we can derive: $\text{Lip}(\tanh) = 1$.

Proof. (**Theorem 2**)
Equation 5.2 defines GRU as: $\boldsymbol{z}_t \odot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{z}_t) \odot \boldsymbol{c}_t$. Since $\boldsymbol{z}_t$ is defined as $\sigma(f_c(\cdot))$, and acts as a gate, we can equivalently define GRU with: $(1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \boldsymbol{c}_t$. This will slightly simplify further calculations. Then, the GRU flow is defined as:

$$F(t, \boldsymbol{h}) = \boldsymbol{h} + \varphi(t) \odot z(t, \boldsymbol{h}) \odot (c(t, \boldsymbol{h}) - \boldsymbol{h}). \tag{5.5}$$

F is invertible when the second summand on the right hand side is a contractive map, i.e., has a Lipschitz constant smaller than one. Since $\varphi(t)$ is bounded to $[0, 1]$ and does not depend on $\boldsymbol{h}$, we only need to show that $z(t, \boldsymbol{h}) \odot (c(t, \boldsymbol{h}) - \boldsymbol{h})$ is contractive. From here, we denote with x and y the input to our functions.

Following Definition 1, let $r(x) = \beta \cdot \sigma(f_r(x))$, with $\text{Lip}(f_r) < 1$. Then we can write:

$$\begin{aligned} |r(x) - r(y)| &= |\beta \cdot \sigma(f_r(x)) - \beta \cdot \sigma(f_r(y))| \\ &\leq \beta|\sigma(f_r(x)) - \sigma(f_r(y))| \\ &\leq \frac{1}{4}\beta|f_r(x) - f_r(y)| \\ &< \frac{1}{4}\beta|x - y|. \end{aligned} \tag{C.4}$$

Similarly, for $z(x)$, where $z(x) = \alpha \cdot \sigma(f_z(x))$, and $\text{Lip}(f_z) < 1$:

$$|z(x) - z(y)| \leq |\alpha \cdot \sigma(f_z(x)) - \alpha \cdot \sigma(f_z(y))| < \frac{1}{4}\alpha|x - y|. \tag{C.5}$$

Then for $c(x) = \tanh(f_c(r(x) \cdot x))$, with $\text{Lip}(f_c) < 1$, we can write:

$$\begin{aligned} |c(x) - c(y)| &= |\tanh(f_c(r(x) \cdot x)) - \tanh(f_c(r(y) \cdot y))| \\ &\leq |f_c(r(x) \cdot x) - f_c(r(y) \cdot y)| \\ &< |r(x) \cdot x - r(y) \cdot y| \\ &< (\underbrace{|r(x)|}_{<\beta} \cdot \underbrace{\text{Lip}(\text{Id})}_{=1} + \underbrace{|x|}_{<1} \cdot \underbrace{\text{Lip}(r)}_{<\frac{1}{4}\beta})|x - y|, \end{aligned} \tag{C.6}$$

where we used Equation C.3 in the last line. Then $\text{Lip}(c) < \frac{5}{4}\beta$. Now, for $c(x) - x$, and using Equation C.2, we write:

$$|c(x) - x - c(y) + y| \leq (\text{Lip}(c) + 1)|x - y|, \tag{C.7}$$

meaning the whole term has Lipschitz constant $\frac{5}{4}\beta + 1$. Finally, for the term on the right hand side of Equation 5.5, the following holds:

$$\begin{aligned}
&|z(x)(c(x) - x) - z(y)(c(y) - y)| \\
&< (\underbrace{|z(x)|}_{<\alpha} \cdot \underbrace{\mathrm{Lip}(c(x) - x)}_{<\frac{5}{4}\beta+1} + \underbrace{|c(x) - x|}_{<2} \cdot \underbrace{\mathrm{Lip}(z(x))}_{<\frac{1}{4}\alpha} |x - y|.
\end{aligned}$$

If we set $\alpha = \frac{2}{5}$, $\beta = \frac{4}{5}$, then the Lipschitz constant is smaller than 1, as required. □

C.1.2.1 Properties of GRU flow

Our GRU flow has the same desired properties as GRU-ODE:

1. Boundedness: hidden state $\boldsymbol{h}$ stays within range $(-1, 1)$,
2. Continuity: the full transformation $\boldsymbol{h} + g(\boldsymbol{h})$ has Lipschitz constant $1 + \mathrm{Lip}(g) \leq 2$.

The gating mechanism in discrete GRU helps with gradient propagation to enable learning long-term dependencies. We emphasize that both GRU flow and GRU-ODE update the hidden state in two distinct ways: 1) with discrete GRU when the new observation arrives, and 2) with continuous GRU between observations. Thus, the gates $\boldsymbol{z}$ and $\boldsymbol{r}$ do not have the same interpretation in discrete GRUCell and in continuous GRU flow or GRU-ODE.

The same way, scalars α and β should not be interpreted as bounds to how much information can pass, but as a way to ensure invertibility. GRU flow has the ability to keep the old state $\boldsymbol{h}$, and does so at the initial condition $t = 0$, but can also overwrite it completely.

C.1.3 ODE reparameterization

The ODESolve operation is usually implemented such that it takes scalar start and end times, t_0 and t_1. However, we are often interested in processing the data in batches, to get speed-up from parallelism on modern hardware. When the previous works [35, 229, 51] received the vectors of start and end times, e.g., $\boldsymbol{t}_0 = [0, 0, 0]$ and $\boldsymbol{t}_1 = [5, 1, 4]$, they would concatenate all the values into a single vector and sort them to get a sequence of strictly ascending times, e.g., $[0, 1, 4, 5]$. The solver would then first solve $0 \to 1$, then $1 \to 4$, and finally $4 \to 5$. Note that for the element in the batch with the largest end time, this requires calling ODESolve multiple times (number of unique time values), instead of only once. Without this procedure, the adaptive solver could take larger steps then the ones imposed by the current batch, meaning we would get better performance.

Chen et al. [34] propose a reparameterization, such that, instead of solving the ODE on the interval $t \in [0, t_{\max}]$, they solve it on $s \in [0, 1]$, with $s = t/t_{\max}$. For the batch of size n, the joint system is:

$$\frac{\mathrm{d}}{\mathrm{d}s} \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_n \end{bmatrix} = \begin{bmatrix} t_1 f(st_1, \boldsymbol{x}_1) \\ t_2 f(st_2, \boldsymbol{x}_2) \\ \vdots \\ t_n f(st_n, \boldsymbol{x}_n) \end{bmatrix}.$$

This allows solving the system in parallel, in contrast to previous works. We used this reparameterization in all of our experiments.

C.1.4 Attentive normalizing flow

We follow the setup from Section 5.2.3, denoting times with $\boldsymbol{t} = (t_1, \ldots, t_n)$, and marks with $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$, $\boldsymbol{x}_i \in \mathbb{R}^d$. We define the self-attention layer as in Equation 2.20. Chen et al. [34], in their attentive CNF model, define the function f from Equation 2.38 for each $\boldsymbol{x}_i$, as the ith output of the attention function. It is important that elements $\boldsymbol{x}_j$, $j > i$, do not influence $\boldsymbol{x}_i$ to ensure we have a proper temporal model. This is achieved by placing $-\infty$ for values above the diagonal of the $\boldsymbol{Q}\boldsymbol{K}^T$ matrix so that softmax returns zero on these places.

Discrete normalizing flows cannot define the transformation using attention and have tractable determinant of the Jacobian at the same time. However, since we actually need an autoregressive model, i.e., the dependence is strictly on the past values, not future, we can define a model similar to attentive CNF. We use Equation 2.20 with diagonal masking to embed the history of all the elements that preceded $\boldsymbol{x}_i$: $\boldsymbol{h}_i = \text{SelfAttention}(\boldsymbol{X}_{1:i-1})$. This is in contrast to [34], who used $\boldsymbol{X}_{1:i}$. Then, the conditioning vector $\boldsymbol{h}_i$ is used as an additional input to neural networks u and v from Equation 5.6, essentially defining a conditional affine coupling normalizing flow.

C.1.5 Linear ODE and change of variables

Consider a linear ODE $f(t, \boldsymbol{z}(t)) = \boldsymbol{A}\boldsymbol{z}(t)$, with $\boldsymbol{z}(0) = \boldsymbol{z}$ and $\boldsymbol{z}(1) = \boldsymbol{x}$. Solving the ODE $0 \to 1$ is the same as calculating $\exp(\boldsymbol{A})\boldsymbol{z}$, where exp is the matrix exponential. Suppose that $\boldsymbol{z} \sim q(\boldsymbol{z})$, then the distribution $p(\boldsymbol{x})$ that we get by transforming $\boldsymbol{x}$ with an ODE is defined as:

$$\log p(\boldsymbol{x}) = \log q(\boldsymbol{z}) - \int_0^1 \operatorname{tr}\left(\frac{\partial f}{\partial \boldsymbol{z}(t)}\right) \mathrm{d}t = \log q(\boldsymbol{z}) - \operatorname{tr}(\boldsymbol{A}), \tag{C.8}$$

or simply: $p(\boldsymbol{x}) = q(\boldsymbol{z}) \exp(\operatorname{tr}(\boldsymbol{A}))^{-1}$.

When using a Hutchinson's trace estimator for the trace approximation we get the same result: $\mathbb{E}_{p(\boldsymbol{\epsilon})}[\int_0^1 \boldsymbol{\epsilon}^T \frac{\partial f}{\partial \boldsymbol{z}(t)} \boldsymbol{\epsilon} \, \mathrm{d}t] = \mathbb{E}_{p(\boldsymbol{\epsilon})}[\boldsymbol{\epsilon}^T \boldsymbol{A} \boldsymbol{\epsilon}] = \operatorname{tr}(\boldsymbol{A})$, where $\mathbb{E}(\boldsymbol{\epsilon}) = \boldsymbol{0}$ and $\operatorname{Cov}(\boldsymbol{\epsilon}) = \boldsymbol{I}$.

Similarly, applying the discrete change of variables, we get the same outcome when using the matrix exponential:

$$p(\boldsymbol{x}) = q(\boldsymbol{z}) |\det J_F(\boldsymbol{z})|^{-1} = q(\boldsymbol{z}) |\det \exp(\boldsymbol{A})|^{-1} = q(\boldsymbol{z}) \exp(\operatorname{tr}(\boldsymbol{A}))^{-1}, \tag{C.9}$$

proving the equivalence of the three different ways to define a CNF with a linear ODE.

C.1.6 Computation complexity of (continuous) normalizing flows

In general, evaluating the trace of the Jacobian of function $f : \mathbb{R}^d \to \mathbb{R}^d$ requires $O(d^2)$ operations. In CNFs, this operation has to be performed at every solver step. Since the number of steps can be very large for more complicated distributions [88], this becomes

Figure C.1: Sample trajectories for synthetic data.

prohibitively expensive. Because of this, Grathwohl et al. [88] introduce computing the approximation of the trace during training. This has the benefit of having a lower cost, $O(d)$. The issue with this method is that the training becomes noisier and after training we have to again rely on exact trace to get the exact density.

On the other hand, computing the determinant of the Jacobian is $O(d^3)$ operation in general. Because of this, regular normalizing flows do not use unconstrained functions f, but rather opt for those that produce triangular Jacobians, e.g., autoregressive [135] or coupling transformations [59], where the determinant is just the product of the diagonal elements, i.e., it is of linear cost $O(d)$.

C.2 Synthetic experiments

We first test the capabilities of our models on periodic signals:

- Sine: $f(t, x) = \cos(t)$ which corresponds to flow $F(t, x) = x + \sin(t)$, $x \in \mathbb{R}$,
- Sawtooth: $F(t, x) = x + t - \lfloor t \rfloor$,
- Square: $F(t, x) = x + \operatorname{sign}(\sin(t))$,
- Triangle: $F(t, x) = \int_0^t \operatorname{sign}(\sin(u))\,\mathrm{d}u$.

We sample initial values x uniformly in $(-2, 2)$ and set the time interval to $(0, 10)$. We additionally check how well the models extrapolate by extending the initial condition interval to $(-4, 4)$ and time to 30. We also use two datasets, generated as solutions to known ODEs:

- Sink: $f(t, \boldsymbol{x}) = \begin{bmatrix} -4 & 10 \\ -3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$,
- Ellipse: $f(t, \boldsymbol{x}) = \begin{bmatrix} \frac{2}{3}x_1 - \frac{2}{3}x_1 x_2 \\ x_1 x_2 - x_2 \end{bmatrix}$, which is a particular parametrization of Lotka-Volterra equations, also known as the predator-prey equations,

where we sample initial conditions $x_1, x_2 \in [0, 1]$ uniformly. For extrapolation, we use $x_1, x_2 \in [1, 2]$. Figure C.1 shows the generated trajectories for all synthetic datasets.

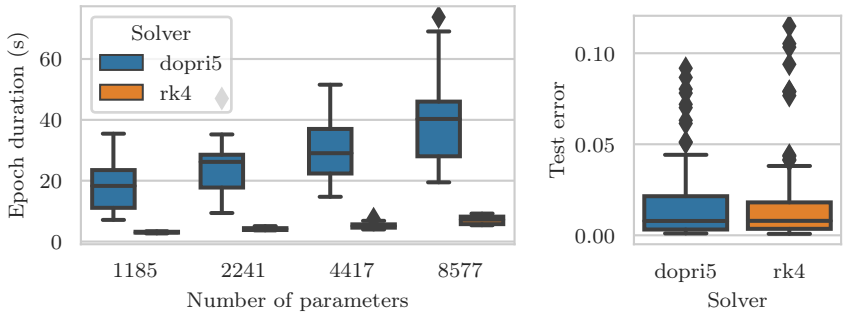


Figure C.2: Fixed solvers are faster to train on synthetic data (Left) but they still have similar accuracy compared to adaptive solvers (Right).

C.2.1 Comparing adaptive and fixed-step solvers

We ran an extensive hyperparameter search for Sine dataset. We test models with 2 or 3 hidden layers, each having dimension of 32 or 64, use tanh or ELU activations between them, and have tanh or identity as the final activation. For each of the model configurations we apply either no regularization or weigh the penalty term with 10^{-3}. Finally, we run each trial 5 times with different seeds and compare between Runge-Kutta fixed-step solver with 20 steps and an adaptive 5th order Dormand-Prince method [60].

As expected, the vast majority of the trials fit the data very well. However, as Figure C.2 shows, an adaptive solver always requires significantly longer training times, regardless of the size of the model, choice of the activations or regularization. We used default tolerance settings (rtol $= 10^{-7}$, atol $= 10^{-9}$) which is why we get such long training times. Therefore, in the other experiments, in the main text, whenever we use dopri5, we use rtol $= 10^{-3}$ and atol $= 10^{-4}$ to make training feasible. This once again shows the trade-off between speed and numerical accuracy.

From the results, one would expect that we can safely use fixed-step solvers and achieve similar or better results with smaller computational demand. However, as Ott et al. [201] showed, this can lead to overlapping trajectories which give non-unique solutions. Breaking the assumptions of our model can lead to misleading results in some cases. Here, we tackle density estimation with continuous normalizing flows as an example. We construct a synthetic 2-dim. dataset as a mixture of zero-centered normal distribution ($\sigma = 0.05$) and uniform points on the perimeter of a unit circle with small noise ($\sigma = 0.01$). We test adaptive dopri5 solver and Euler method with 20 steps.

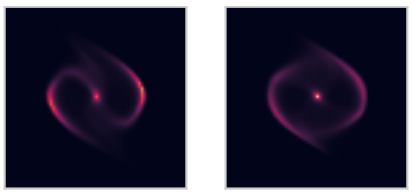


Figure C.3: Density learned with Euler and dopri5 solver. The estimated area under the curve for Euler method is 1.06, meaning it does not define a proper density.

The fixed solver achieves better results but Figure C.3 visually demonstrates that it is not really capturing the true distribution. In reality, it *cheats* by not defining a proper

MSE ($\times 10^{-2}$)	Ellipse	Sawtooth	Sink	Square	Triangle
Neural ODE	25.59±3.19	8.74±1.10	1.38±0.17	24.34±0.3	2.76±0.09
Coupling flow	14.16±4.80	**1.25±0.33**	0.50±0.06	**3.38±0.4**	0.19±0.02
ResNet flow	**9.48±2.64**	1.38±0.13	**0.40±0.04**	3.56±0.1	**0.0±0.0**

Table C.1: Test error on synthetic data, lower is better. Best results in bold.

MSE ($\times 10^{-2}$)	Ellipse	Sawtooth	Sink	Square	Triangle
Neural ODE	**19.82±1.34**	10.64±1.76	18.0±1.18	32.96±3.0	4.22±0.56
Coupling flow	515.8±555.6	**1.32±0.36**	**5.53±2.23**	**3.93±0.76**	**0.2±0.04**
ResNet flow	100.4±45.4	3.49±1.14	6.65±2.23	9.84±2.94	0.79±0.21

Table C.2: Error on trajectories that start at initial conditions out of training distribution. Some trials returned outliers that skew the results (e.g., coupling flow on ellipse dataset).

	Ellipse	Sawtooth	Sink	Square	Triangle
Neural ODE	9.3±0.88	8.25±0.33	8.78±0.81	7.81±0.34	7.91±0.35
Coupling flow	**0.7±0.11**	**0.46±0.22**	**0.6±0.05**	**0.49±0.14**	**0.58±0.16**
ResNet flow	1.05±0.04	1.01±0.15	1.24±0.13	0.98±0.04	1.01±0.09

Table C.3: Wall-clock time (in seconds) to run the last training epoch, using the same batch size.

density function that integrates to 1. Since it has more mass to distribute, it can achieve better likelihood results. This might be harder to detect in higher dimensions which is particularly problematic since most of the literature reports log-likelihood on test data. Even though we took Euler method as an extreme example, the same can happen in other solvers as well.

C.2.2 Comparing flow configurations

Similar to Appendix C.2.1, we compare different flow models on synthetic sine data. We try coupling and ResNet models with linear and tanh for φ, as well as an embedding with 8 Fourier features (bounded to $(0, 1)$ interval in ResNet model), see Section 5.1.1 for more details. Both models have either 2 or 4 stacked transformations, each with a two hidden layer neural network with 64 hidden dimensions. We run each configuration 5 times with and without weight regularization (10^{-3}).

All the models capture the data perfectly, except for the coupling flow with linear function of time φ which does not converge. This could be due to inability of neural networks to process large input values. The issue can be fixed with different initialization or normalizing the input time values.

Tables C.1, C.2 and C.3 show that neural flows outperform neural ODEs in forecasting, extrapolation with different initial values, and are faster during training.

C.3 Additional results

Table C.4 compares the training times for smoothing experiment. Neural ODE models use Euler method with 20 steps (the adaptive method is slower). Table C.5 shows the average wall-clock time to run a single epoch for different TPP models. We include ablations for flow and ODE models that use different continuous RNN encoders, and a model without an encoder. Table C.6 shows full negative log-likelihood results for the TPP experiment. Table C.7 shows the full NLL results for marked TPPs.

	Activity	MuJoCo	Physionet
Neural ODE	200.884±7.239	192.209±2.526	103.198±4.977
Coupling flow	**106.298±2.314**	**46.171±1.742**	**78.561±1.050**
ResNet flow	134.336±3.453	102.745±2.369	101.966±8.285

Table C.4: Average time (in seconds) to run a single epoch during training for different models, all other training parameters being the same.

		Poisson	Hawkes1	Hawkes2	Renewal	MOOC	Reddit	Wiki
Cont.	Neural ODE	96.7	129.8	208.6	111.2	844.2	612.8	157.9
	Coupling flow	10.8	11.2	10.8	11.1	180.8	113.1	31.7
	ResNet flow	7.1	7.1	7.2	7.3	130.0	83.8	19.9
Mixture	GRU-ODE	39.7	42.3	55.9	39.3	600.0	419.5	97.9
	ODE-LSTM	35.9	39.0	37.8	43.8	569.4	443.6	109.4
	Coupling flow	3.4	3.4	3.3	3.3	47.0	37.2	8.5
	ResNet flow	5.9	5.9	5.8	5.9	96.5	64.9	16.1
	GRU flow	3.6	3.5	3.3	3.7	52.8	36.4	9.7

Table C.5: Average time (in seconds) to run a single epoch during training for TPP models.

C.4 Data pre-processing

C.4.1 Encoder-decoder datasets

MuJoCo dataset. Using Deep Mind Control Suite and MuJoCo simulator, Rubanova et al. [229] generate 10000 sequences by sampling initial body position in $\mathbb{R}^2$ uniformly from $[0, 0.5]$, limbs from $[-2, 2]$, and velocities from $[-5, 5]$ interval. We use this dataset without any changes.

Activity dataset. Following [229], we round up the time measurements to 100ms intervals. This was done to reduce the size of the union of all the points when batching but is unnecessary when using our flow models, and also when using the reparameterization for ODEs [34].

Original labels are: 'walking', 'falling', 'lying down', 'lying', 'sitting down', 'sitting', 'standing up from lying', 'on all fours', 'sitting on the ground', 'standing up from sitting',

	Synthetic data	Poisson	Hawkes1	Hawkes2	Renewal
	Ground truth	0.9996	0.6405	0.1192	0.2667
	Without history	1.0046	0.7826	0.2354	0.2837
	Discrete GRU	1.0097±0.005	0.6424±0.006	0.1267±0.006	**0.2598±0.016**
Cont.	Jump ODE	**0.9945±0.016**	0.6461±0.009	0.2246±0.042	0.3124±0.022
Cont.	Coupling flow	1.0099±0.005	0.6441±0.007	0.1376±0.005	0.2720±0.017
Cont.	ResNet flow	1.0105±0.005	0.6426±0.007	0.1813±0.025	0.2851±0.018
Mix.	GRU-ODE	1.0100±0.005	**0.6419±0.007**	**0.1239±0.005**	0.2601±0.017
Mix.	ODE-LSTM	1.0108±0.005	0.6448±0.006	0.1253±0.005	0.2605±0.017
Mix.	Coupling flow	1.0103±0.005	0.6450±0.008	0.1254±0.006	0.2605±0.016
Mix.	GRU flow	1.0100±0.005	0.6439±0.007	0.1270±0.006	0.2608±0.016
Mix.	ResNet flow	1.0104±0.005	0.6443±0.006	0.1249±0.005	0.2603±0.017

	Real-word data	MOOC	Reddit	Wiki
	Without history	2.0623	1.5402	1.5813
	Discrete GRU	-0.4448±0.294	-0.9299±0.118	-0.5832±0.321
Cont.	Jump ODE	0.8710±0.157	0.1308±0.018	-0.3115±0.011
Cont.	Coupling flow	0.7694±0.172	-0.1263±0.273	-0.2807±0.500
Cont.	ResNet flow	**-1.2379±0.049**	**-1.2962±0.126**	-1.2907±0.045
Mix.	GRU-ODE	-0.2626±0.183	-1.0907±0.076	-1.3635±0.071
Mix.	ODE-LSTM	-0.2277±0.331	-1.0888±0.029	**-1.3727±0.327**
Mix.	Coupling flow	-0.4026±0.584	-1.0933±0.161	-1.2702±0.178
Mix.	GRU flow	-0.3509±0.220	-1.0605±0.113	-0.9852±0.105
Mix.	ResNet flow	-0.5664±0.278	-1.0291±0.174	-1.1937±0.048

Table C.6: Test negative log-likelihood (mean±standard deviation) for all TPP models.

'standing up from sitting on the ground'. Rubanova et al. [229] combine similar positions into one group resulting in 7 classes: 'walking', 'falling', 'lying', 'sitting', 'standing up', 'on all fours', 'sitting on the ground'. Data is split in train, validation and test set (75%–5%–20%).

Physionet dataset. We use PhysioNet Challenge 2012 [245], where the goal is to predict the mortality of patients upon being admitted to ICU. We process the data following [229] to exclude time-invariant features, and round the time stamps to one minute. Each feature is normalized to $[0, 1]$ interval. Data is split 60%–20%–20%.

When reporting MSE scores for the reconstruction task we scale the results by 10^2 for activity dataset and by 10^3 for others, following Rubanova et al. [229] to have better readability. This is equivalent to scaling the data beforehand.

C.4.2 MIMIC-III and MIMIC-IV

We follow [51] for processing **MIMIC-III** dataset. We process MIMIC-IV similarly.

		MOOC	Reddit	Wiki
	Discrete GRU	2.7563±0.141	1.8468±0.016	8.0527±0.170
Cont.	Jump ODE	4.6118±0.070	3.6654±0.000	10.6040±0.304
Cont.	Coupling flow	5.5494±0.413	3.6312±0.324	9.7214±0.101
Cont.	ResNet flow	2.9466±0.000	2.3932±0.131	10.4368±0.034
Mix.	GRU-ODE	3.5344±0.242	2.3078±0.033	**7.5537±0.065**
Mix.	ODE-LSTM	3.0723±0.114	1.9057±0.164	8.3187±0.231
Mix.	Coupling flow	**2.5877±0.176**	**1.6817±0.095**	8.8018±0.057
Mix.	ResNet flow	3.0005±0.081	1.9491±0.008	8.5489±0.267

Table C.7: Test negative log-likelihood (mean±standard deviation) for all marked TPP models.

The publicly available **MIMIC-IV** database provides clinical data of intensive care unit (ICU) patients at the tertiary academic medical center in Boston [119, 83]. It builds upon the MIMIC-III database and contains de-identified patient records from 2008 to 2019 [120]. We use version MIMIC-IV 1.0, which was released March 16th, 2021.

To preprocess the data, we first select the subset of patients who:

- are registered in the admissions table,
- stayed in the ICU for at least 2 days and no more than 30 days,
- are older than 15 years at the time of the admission,
- have chart-event data available,

which leaves us with 17874 patients.

There are four types of data sources available for ICU patients: chart-events, inputs, outputs and prescriptions. The chart-events table contains the patient's routine vital signs as well as any additional information such as laboratory tests. The input table documents drugs administered to the patient through, e.g., solutions and the prescription table stores information about medication given in any other form. Lastly, the outputs table contains any output data from, e.g., a catheter for the patient during their stay.

Because the medication in the input table is administered over time, the administered units and doses have to be unified and then split into entries which are spread out over time. We choose 30 minutes as our sampling window and, for all administered medications with duration longer than an hour, split them into fixed time injections.

For all other tables, we only keep the most commonly used entries:

- **Chart-events**: Alanine Aminotransferase, Albumin, Alkaline Phosphatase, Anion Gap, Asparate Aminotransferase, Base Excess, Basophils, Bicarbonate, Bilirubin, Calcium, Chloride, Creatinine, Eosinophils, Glucose, Hematocrit, Hemoglobin, Lactate, Lymphocytes, Magnesium, MCH, MCV, Monocytes, Neutrophils, pCO2, pH, pO2, Phosphate, Platelet Count, Potassium, PT, PTT, RDW, Red Blood Cells, Sodium, Specific Gravity, Total CO2, Urea Nitrogen and White Blood Cells.

- **Outputs**: Chest Tube, Emesis, Fecal Bag, Foley, Jackson Pratt, Nasogastric, OR EBL, OR Urine, Oral Gastric, Pre-Admission, Stool, Straight Cath, TF Residual, TF Residual Output and Void.
- **Prescriptions**: Acetaminophen, Aspirin, Bisacodyl, D5W, Docusate Sodium, Heparin, Humulin-R Insulin, Insulin, Magnesium Sulfate, Metoprolol Tartrate, Pantoprazole, Potassium Chloride and Sodium Chloride 0.9% Flush.

C.4.3 TPP datasets

We follow previous works [199, 143] to generate and pre-process temporal point process data, all of which have been already introduced in Chapter 3. We use four synthetic datasets: Poisson, Renewal, Hawkes1, and Hawkes2, generated the same way as described in Appendix A.5.1. We also use three real-world dataset: Reddit, MOOC, and Wiki, as described in Appendix A.5.2.

In our implementation, we use inter-event times $\tau_i = t_i - t_{i-1}$ and for real-world data, we normalize them by dividing them with the empirical mean $\bar{\tau}$ from the training set $\tau_i \mapsto \tau_i / \bar{\tau}$. This can still yield quite large values so for better numerical stability during training, we use log-transform $\tau \mapsto \log(\tau + 1)$. We can think of log-transformation as a change of variables and include it in the negative log-likelihood loss using the probability change of variables formula (see Section 5.2.3).

C.4.4 Spatial datasets

For spatial data used in time-dependent density estimation experiment, we used the datasets from Chen et al. [34] with the same pre-processing pipeline. See [34] for details.

Earthquakes contains events gathered between 1990 and 2020 in Japan, with the magnitude of at least 2.5 [270]. Each sequence has length of 30 days, with the gap of 7 days between them. There are 950 training sequences, and 50 validation and test sequences.

Covid data uses daily cases from March to July 2020 in New Jersey state [266]. The data is gathered on county level and dequantized. Each sequence covers 7 days. There are 1450 sequences in the training set, 100 in validation and 100 in test set.

Bikes contains rental events from a bike sharing service in New York using data from April to August 2019. Each sequence corresponds to a single day, starting at 5am. The data is split in training, test and validation set: 2440, 300, 320 sequences, respectively.

All the spatial values are normalized to zero mean and unit variance. We also normalize the temporal component to $[0, 1]$ interval.

C.4.5 Hyperparameters

In all experiments we use Adam optimizer [134]. Below, we list other hyperparameters.

Smoothing experiments: Batch size: 100, Learning rate: 1e-3 with decay 0.5 every 20 epochs, Hidden layers: 3. Models: ODE models: Solver: euler or dopri5. Flow models: ResNet or coupling flow, Flow layers: 1 or 2, $\varphi(t)$: tanh for ResNet and linear for coupling (used in all experiments). Datasets:

	Encoder-decoder hidden dim.	Latent state dim.	GRU dim.
MuJoCo	100-100	20	50
Activity	30-100	20	100
Physionet	40-50	20	50

Filtering experiment: Batch size: 100, Learning rate: 1e-3 with decay 0.33 every 20 epochs, Hidden dimension: 64, Datasets: MIMIC-III or MIMIC-IV. ODE models: Solver: euler or dopri5, Hidden layers: 3. Flow models: GRU flow or ResNet flow, Flow layers: 1 or 4, Hidden layers: 2.

TPP experiment: (With or without marks) Batch size: 50, Learning rate: 1e-3, Hidden dimension: 64, Datasets: Reddit or MOOC or Wiki. ODE models: Models: continuous or mixture (ODE-LSTM or GRU-ODE), Hidden layers: 3. Flow models: Models: continuous (ResNet or coupling flow) or mixture (ResNet or coupling or GRU flow), Flow layers: 1, Hidden layers: 2. RNN models: GRU.

Density estimation experiment: Batch size: 50, Learning rate: 1e-3, Hidden dimension: 64, Models: time-varying or attentive (for both CNFs and NFs). CNF: Hidden layers: 4. Coupling normalizing flows: Base density layers: 4 or 8, Time-dependent NF layers: 4 or 8.

D Appendix for Chapter 6

D.1 Derivations

D.1.1 Discrete diffusion posterior probability

We extend Ho et al. [102] by using full covariance $\boldsymbol{\Sigma}(\boldsymbol{t})$ to define the noise distribution across time $\boldsymbol{t}$. If $\boldsymbol{\Sigma} = \boldsymbol{L}\boldsymbol{L}^T$ and keeping the same definitions from Section 6.1.1 for β_n, α_n, and $\bar{\alpha}_n$, we can write:

$$\boldsymbol{X}_n = \sqrt{1-\beta_n}\boldsymbol{X}_{n-1} + \sqrt{\beta_n}\boldsymbol{L}\boldsymbol{\epsilon}, \tag{D.1}$$

$$\boldsymbol{X}_n = \sqrt{\bar{\alpha}_n}\boldsymbol{X}_0 + \sqrt{1-\bar{\alpha}_n}\boldsymbol{L}\boldsymbol{\epsilon}, \tag{D.2}$$

with $\boldsymbol{\epsilon} \in \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$. This corresponds to the following transition distributions:

$$q(\boldsymbol{X}_n|\boldsymbol{X}_{n-1}) = \mathcal{N}(\sqrt{1-\beta_n}\boldsymbol{X}_{n-1}, \beta_n\boldsymbol{\Sigma}), \tag{D.3}$$

$$q(\boldsymbol{X}_n|\boldsymbol{X}_0) = \mathcal{N}(\sqrt{\bar{\alpha}_n}\boldsymbol{X}_0, (1-\bar{\alpha}_n)\boldsymbol{\Sigma}). \tag{D.4}$$

We are interested in $q(\boldsymbol{X}_{n-1}|\boldsymbol{X}_n, \boldsymbol{X}_0) \propto q(\boldsymbol{X}_n|\boldsymbol{X}_{n-1})q(\boldsymbol{X}_{n-1}|\boldsymbol{X}_0)$. Since both distributions on the right-hand side are normal, the result will be normal as well. We can write the resulting distribution as $\mathcal{N}(\tilde{\boldsymbol{\mu}}, \tilde{\boldsymbol{\Sigma}})$, where:

$$\begin{aligned}
\tilde{\boldsymbol{\mu}} &= \boldsymbol{R}(\boldsymbol{X}_n - \boldsymbol{A}\boldsymbol{\mu}_1) + \boldsymbol{\mu}_1 \\
\tilde{\boldsymbol{\Sigma}} &= \boldsymbol{\Sigma}_1 - \boldsymbol{R}\boldsymbol{A}\boldsymbol{\Sigma}_1^T \\
\boldsymbol{R} &= \boldsymbol{\Sigma}_1\boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{\Sigma}_1\boldsymbol{A}^T + \boldsymbol{\Sigma}_2)^{-1},
\end{aligned}$$

with $\boldsymbol{A} = \sqrt{1-\beta_n}\boldsymbol{I}$, $\boldsymbol{\mu}_1 = \sqrt{\bar{\alpha}_{n-1}}\boldsymbol{X}_0$, $\boldsymbol{\Sigma}_1 = (1-\bar{\alpha}_{n-1})\boldsymbol{\Sigma}$, and $\boldsymbol{\Sigma}_2 = \beta_n\boldsymbol{\Sigma}$. We can now write:

$$\begin{aligned}
\boldsymbol{R} &= (1-\bar{\alpha}_{n-1})\boldsymbol{\Sigma}\sqrt{1-\beta_n}\left(\sqrt{1-\beta_n}(1-\bar{\alpha}_{n-1})\boldsymbol{\Sigma}\sqrt{1-\beta_n} + \beta_n\boldsymbol{\Sigma}\right)^{-1} \\
&= \frac{(1-\bar{\alpha}_{n-1})\sqrt{\alpha_n}}{\alpha_n(1-\bar{\alpha}_{n-1}) + 1 - \alpha_n}\boldsymbol{\Sigma}\boldsymbol{\Sigma}^{-1} \\
&= \frac{1-\bar{\alpha}_{n-1}}{1-\bar{\alpha}_n}\sqrt{\alpha_n},
\end{aligned}$$

and from there:

$$\begin{aligned}
\tilde{\boldsymbol{\mu}} &= \frac{1-\bar{\alpha}_{n-1}}{1-\bar{\alpha}_n}\sqrt{\alpha_n}\left(\boldsymbol{X}_n - \sqrt{1-\beta_n}\sqrt{\bar{\alpha}_{n-1}}\boldsymbol{X}_0\right) + \sqrt{\bar{\alpha}_{n-1}}\boldsymbol{X}_0 \\
&= \frac{1-\bar{\alpha}_{n-1}}{1-\bar{\alpha}_n}\sqrt{\alpha_n}\boldsymbol{X}_n + \sqrt{\bar{\alpha}_{n-1}}\left(1 - \frac{1-\bar{\alpha}_{n-1}}{1-\bar{\alpha}_n}\alpha_n\right)\boldsymbol{X}_0 \\
&= \frac{1-\bar{\alpha}_{n-1}}{1-\bar{\alpha}_n}\sqrt{\alpha_n}\boldsymbol{X}_n + \frac{\sqrt{\bar{\alpha}_{n-1}}}{1-\bar{\alpha}_n}\beta_n\boldsymbol{X}_0,
\end{aligned} \tag{D.5}$$

and using the fact that $\boldsymbol{\Sigma}$ is a symmetric matrix:

$$\begin{aligned}\tilde{\boldsymbol{\Sigma}} &= (1-\bar{\alpha}_{n-1})\boldsymbol{\Sigma} - \frac{1-\bar{\alpha}_{n-1}}{1-\bar{\alpha}_n}\sqrt{\alpha_n}\sqrt{1-\beta_n}(1-\bar{\alpha}_{n-1})\boldsymbol{\Sigma}^T \\ &= \left(1-\bar{\alpha}_{n-1} - \frac{1-\bar{\alpha}_{n-1}}{1-\bar{\alpha}_n}\alpha_n(1-\bar{\alpha}_{n-1})\right)\boldsymbol{\Sigma} \\ &= \frac{1-\bar{\alpha}_{n-1}}{1-\bar{\alpha}_n}\beta_n\boldsymbol{\Sigma}.\end{aligned} \tag{D.6}$$

Therefore, the only difference to the derivation in Ho et al. [102] is the $\boldsymbol{\Sigma}(\boldsymbol{t})$ instead of the identity matrix $\boldsymbol{I}$ in the covariance.

D.1.2 Discrete diffusion loss

We use the evidence lower bound from Equation 6.3. The distribution $q(\boldsymbol{X}_{n-1}|\boldsymbol{X}_n, \boldsymbol{X}_0)$ is defined as $\mathcal{N}(\tilde{\boldsymbol{\mu}}, C_1\boldsymbol{\Sigma})$, where C_1 is some constant (Equations D.5 and D.6). Similar to Ho et al. [102], we start with the parameterization for the reverse process $p(\boldsymbol{X}_{n-1}|\boldsymbol{X}_n) = \mathcal{N}(\boldsymbol{\mu}_\theta(\boldsymbol{X}_n, \boldsymbol{t}, n), \beta_n\boldsymbol{\Sigma})$, where:

$$\boldsymbol{\mu}_\theta(\boldsymbol{X}_n, \boldsymbol{t}, n) = \frac{1}{\sqrt{\alpha_n}}\left(\boldsymbol{X}_n - \frac{\beta_n}{\sqrt{1-\bar{\alpha}_n}}\boldsymbol{\epsilon}_\theta(\boldsymbol{X}_n, \boldsymbol{t}, n)\right).$$

Then the KL-divergence is between two normal distributions so we can write the following, where C_2 is a term that does not depend on the parameters θ:

$$\begin{aligned}D_{\mathrm{KL}}[q(\boldsymbol{X}_{n-1}|\boldsymbol{X}_n, \boldsymbol{X}_0) \parallel p(\boldsymbol{X}_{n-1}|\boldsymbol{X}_n)] &= D_{\mathrm{KL}}[\mathcal{N}(\tilde{\boldsymbol{\mu}}, C_1\boldsymbol{\Sigma}) \parallel \mathcal{N}(\boldsymbol{\mu}_\theta(\boldsymbol{X}_n, \boldsymbol{t}, n), \beta_n\boldsymbol{\Sigma})] \\ &= \frac{1}{2}(\tilde{\boldsymbol{\mu}} - \boldsymbol{\mu}_\theta)^T\boldsymbol{\Sigma}^{-1}(\tilde{\boldsymbol{\mu}} - \boldsymbol{\mu}_\theta) + C_2.\end{aligned}$$

Ho et al. [102] show that their loss can be simplified to Equation 6.4 given their particular parameterization. Recall that we obtain noise by computing $\boldsymbol{L}\tilde{\boldsymbol{\epsilon}}$, where $\tilde{\boldsymbol{\epsilon}}$ is unit normal and $\boldsymbol{L}$ is the lower triangular matrix from the Cholesky decomposition of the covariance $\boldsymbol{\Sigma} = \boldsymbol{L}\boldsymbol{L}^T$.

Therefore, we can factorize $\boldsymbol{L}$ from the bracket containing the difference of two means to get:

$$D_{\mathrm{KL}}[q(\boldsymbol{X}_{n-1}|\boldsymbol{X}_n, \boldsymbol{X}_0) \parallel p(\boldsymbol{X}_{n-1}|\boldsymbol{X}_n)] = (\boldsymbol{L}\boldsymbol{a})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{L}\boldsymbol{a}) = \boldsymbol{a}^T\boldsymbol{L}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{L}\boldsymbol{a},$$

where we write $\boldsymbol{a}$ as a shorthand for the term depending on $\boldsymbol{X}_0$ and unit normal noise $\tilde{\boldsymbol{\epsilon}}$. The term $\boldsymbol{L}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{L}$ evaluates to identity and we are again left with the same loss as in Ho et al. [102]. That is, we can use the same trick to simplify the loss to be the mean squared error between the true noise and the predicted noise, which leads to faster evaluation and better results. Note that in the above notation, we have a set of observations $\boldsymbol{X}$ for times $\boldsymbol{t}$ that we feed into the model $\boldsymbol{\epsilon}_\theta$ to predict a set of noise values $\boldsymbol{\epsilon}(t)$, $t \in \boldsymbol{t}$, whereas, previous works predicted the noise for each data point independently.

D.1.3 Continuous diffusion transition probability

Given an SDE in Equation 6.11 we want to compute the change in the variance $\tilde{\boldsymbol{\Sigma}}_s$, where s denotes the diffusion time. The derivation is similar to that in Song et al. [252]. We start with the Equation 5.51 from Särkkä and Solin [234]:

$$\frac{\mathrm{d}\tilde{\boldsymbol{\Sigma}}_s}{\mathrm{d}s} = \mathbb{E}[f(\boldsymbol{X}_s, s)(\boldsymbol{X}_s - \boldsymbol{\mu})^T] + \mathbb{E}[(\boldsymbol{X}_s - \boldsymbol{\mu})f(\boldsymbol{X}_s, s)^T] + \mathbb{E}[\boldsymbol{L}(\boldsymbol{X}_s, s)\boldsymbol{Q}\boldsymbol{L}(\boldsymbol{X}_s, s)^T],$$

where f is the drift, $\boldsymbol{L}$ is the SDE diffusion term and $\boldsymbol{Q}$ is the diffusion matrix. From here, the only difference to Song et al. [252] is in the last term; they obtain $\beta(s)\boldsymbol{I}$ while we have a full covariance matrix from the stochastic process: $\beta(s)\boldsymbol{\Sigma}$. Therefore, we only need to slightly modify the result:

$$\frac{\mathrm{d}\boldsymbol{\Sigma}_s}{\mathrm{d}s} = \beta(s)(\boldsymbol{\Sigma} - \tilde{\boldsymbol{\Sigma}}_s),$$

which will give us the covariance of the transition probability as in Equation 6.12. The derivation for the mean is unchanged as our drift term is the same as in Song et al. [252].

D.1.4 Sampling from an Ornstein-Uhlenbeck process

In the following, we discuss three different approaches to sampling noise $\boldsymbol{\epsilon}(\cdot)$ from an OU process defined by γ at time points $t_0, \ldots, t_{M-1}$.

1. **Modified Wiener.** As we already mentioned in Section 6.2.1, we can use a time-changed and scaled Wiener process: $e^{-\gamma t}W_{e^{2\gamma t}}$. Sampling from a Wiener process is straightforward: given a set of time increments $\Delta t_0, \ldots, \Delta t_{M-1}$, we sample M points independently from $\mathcal{N}(0, \Delta t_i)$ and cumulatively sum all the samples. The time changed process first needs to reparameterize the time values. The issue arises when applying the exponential for large t which leads to numerical instability. This can be mitigated by re-scaling t.

2. **Discretized SDE.** A numerically stable approach involves *solving* the OU SDE in fixed steps. The point at $t = 0$, $\boldsymbol{\epsilon}(0)$ is sampled from unit Gaussian. After that, each point is obtained based on the previous, that is, i-th point $\boldsymbol{\epsilon}(t_i)$ is calculated as $\boldsymbol{\epsilon}(t_i) = c\boldsymbol{\epsilon}(t_{i-1}) + \sqrt{1 - c^2}z$, where $c = \exp(-\gamma(t_i - t_{i-1}))$ and $z \sim \mathcal{N}(0, 1)$. This is an iterative procedure but is quite fast and stable.

3. **Multivariate normal.** Finally, we can treat the process as a multivariate normal distribution with mean zero and covariance $\boldsymbol{\Sigma}_{ij}(t_i, t_j) = \exp(-\gamma|t_i - t_j|)$. Given a set of time points $\boldsymbol{t}$ it is easy to obtain the covariance matrix $\boldsymbol{\Sigma}$ and its factorization $\boldsymbol{L}^T\boldsymbol{L}$. To sample, we first draw $\tilde{\boldsymbol{\epsilon}} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ and then $\boldsymbol{\epsilon} = \boldsymbol{L}\tilde{\boldsymbol{\epsilon}}$. Since our model performs best if it predicts $\tilde{\boldsymbol{\epsilon}}$, we opted for this particular sampling approach. If $\boldsymbol{t}$ is not changing, $\boldsymbol{L}$ can be computed once and the performance impact will be minimal. Also when sampling new realizations, $\boldsymbol{L}$ has to be computed only once, before the sampling loop (see Algorithm 2).

D.2 Experimental details

D.2.1 Probabilistic modeling

D.2.1.1 Datasets

The properties of the open datasets used in the forecasting experiment are detailed in Table D.1. Additionally, we generate 6 synthetic datasets, each with 10000 samples, that involve stochastic processes, dynamical and chaotic systems.

1. CIR (Cox-Ingersoll-Ross SDE) is the stochastic differential equations defined by:

$$\mathrm{d}x = a(b - x)\,\mathrm{d}t + \sigma\sqrt{x}\,\mathrm{d}W_t,$$

   where we set $a = 1$, $b = 1.2$, $\sigma = 0.2$ and sample $x_0 \sim \mathcal{N}(0, 1)$ but only take the positive values, otherwise the $\sqrt{x}$ term is undefined. We solve for $t \in \{1, \ldots, 64\}$.

2. Lorenz is a chaotic system in three dimensions. It is governed by the following equations:

$$\begin{aligned}\dot{x} &= \sigma(y - x),\\ \dot{y} &= \rho x - y - xz,\\ \dot{z} &= xy - \beta z,\end{aligned}$$

   where $\rho = 28$, $\sigma = 10$, $\beta = 2.667$, and t is sampled 100 times, uniformly on $[0, 2]$, and $x, y, z \sim \mathcal{N}(\mathbf{0}, 100\boldsymbol{I})$.

3. Ornstein-Uhlenbeck is defined as:

$$\mathrm{d}x = (\mu t - \theta x)\,\mathrm{d}t + \sigma\,\mathrm{d}W_t,$$

   with $\mu = 0.02$, $\theta = 0.1$ and $\sigma = 0.4$. We sample time the same way as for CIR.

4. Predator-prey is a 2D dynamical system defined with an ODE:

$$\begin{aligned}\dot{x} &= 2/3x - 2/3xy,\\ \dot{y} &= xy - y.\end{aligned}$$

5. Sine dataset is generated as a mixture of 5 random sine waves $a\sin(bx + c)$, where $a \sim \mathcal{N}(3, 1)$, $b \sim \mathcal{N}(0, 0.25)$, and $c \sim \mathcal{N}(0, 1)$.

6. Sink is again a dynamical system, governed by:

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \begin{bmatrix}-4 & 10\\ -3 & 2\end{bmatrix}\boldsymbol{x},$$

   with $\boldsymbol{x}_0 \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$.

Dataset	Dim. d	Dom.	Freq.	Time steps	Pred. steps
Exchange	8	$\mathbb{R}^+$	day	$6,071$	30
Solar	137	$\mathbb{R}^+$	hour	$7,009$	24
Electricity	370	$\mathbb{R}^+$	hour	$5,833$	24

Table D.1: Multivariate dimension, domain, frequency, total training time steps, and prediction length properties of the training datasets used in the forecasting experiments.

D.2.1.2 CTFP

We implement continuous-time flow process [56] which is a normalizing flow model for stochastic processes. That is, there is a predefined base distribution $p(\boldsymbol{z})$ and a series of invertible transformations f such that we can generate samples $\boldsymbol{x} = f(\boldsymbol{z})$, and evaluate the density in closed-form by computing $\boldsymbol{z} = f^{-1}(\boldsymbol{x})$ and using the change of variables formula. For more details on normalizing flows, see Kobyzev et al. [138]. The novel idea in CTFP is to change the base density to a stochastic process, i.e., a Wiener process, to obtain the distribution over the functions, similar to our work. In our case, we do not use invertible functions but learn to inverse the noising process, and additionally, we add noise at multiple levels instead only in the beginning. In the experiments, we define a CTFP model as a 12-layer real NVP architecture [59] with 2 hidden layers in each layer's MLP.

D.2.1.3 Latent ODE

Latent ODE is a variational autoencoder architecture, with an encoder that represents the complete time series as a single vector following $q(\boldsymbol{z})$, and a decoder that produces the samples at observation times t_i, $\boldsymbol{z}(t_i) = f(\boldsymbol{z}), \boldsymbol{z} \sim q(\boldsymbol{z})$. The final step is projection to a data space $\boldsymbol{q}(t_i) \mapsto \boldsymbol{x}(t_i)$. The key idea is to use the neural ordinary differential equation [35] to define the evolution of the latent variable $\boldsymbol{z}(\cdot)$, thus, have a probabilistic model of the function. This is different from our approach as it models the function in a latent space, with a single source of randomness at the beginning of the time series. That is, the random value is sampled at $t = 0$ and the time series is determined from there onward, whereas our method samples random values on the whole interval $[0, T]$ and does so multiple times (for N diffusion steps) until we get the new realization. In the experiments, we use a two layer neural network for the neural ODE, and another two layer network for projection to the data space.

D.2.1.4 Our models

We use two models, one is a simple feedforward network, and the second is an RNN-based model. We also use a simple transformer-based model [273] that achieves similar results to an RNN. The model takes in the time series $\boldsymbol{X}$, times of the observations $\boldsymbol{t}$ and the diffusion step n or diffusion time s. The output is the same size as $\boldsymbol{X}$. The feedforward

		CIR	Lorenz	OU	Predator-prey	Sine	Sink
RNN-based model							
DSPD	Gauss	0.5245±0.0252	0.512±0.0212	0.568±0.051	0.5275±0.0383	0.5565±0.0353	0.526±0.0085
	GP	0.5115±0.0282	0.5135±0.0288	0.5055±0.0458	0.5855±0.0219	0.5255±0.009	0.513±0.0103
	OU	0.514±0.0737	0.6095±0.0964	0.5605±0.0581	0.5865±0.053	0.507±0.11	0.6255±0.1672
CSPD	Gauss	0.644±0.0373	0.5015±0.0243	0.6105±0.0153	0.548±0.0751	0.611±0.0516	0.5495±0.0313
	GP	0.5795±0.0541	0.674±0.0739	0.5025±0.0622	0.607±0.0538	0.5575±0.0376	0.5345±0.0201
	OU	0.4535±0.165	0.715±0.0884	0.5255±0.011	0.5835±0.0723	0.556±0.118	0.5795±0.0173
Feedforward model							
DSPD	Gauss	0.624±0.0438	0.713±0.1798	0.5275±0.0371	1.0±0.0	0.7875±0.0585	0.9695±0.0302
	GP	0.558±0.0611	0.894±0.212	0.5535±0.1152	0.7565±0.1362	0.735±0.2146	0.784±0.2281
	OU	1.0±0.0	1.0±0.0	1.0±0.0	1.0±0.0	1.0±0.0	1.0±0.0
CSPD	Gauss	0.537±0.0458	0.959±0.0808	0.5155±0.0165	0.9995±0.001	0.6335±0.0765	0.9095±0.1306
	GP	0.645±0.1034	1.0±0.0	0.507±0.0264	0.894±0.212	0.894±0.212	0.88±0.088
	OU	0.984±0.032	1.0±0.0	0.9905±0.019	1.0±0.0	1.0±0.0	1.0±0.0

Table D.2: Accuracy of the discriminator trained on samples from a diffusion model. Values around 0.5 indicate the discriminative model cannot distinguish the model samples and real data. Values closer to 1 indicate the generative model is not capturing the data distribution.

model embeds the time and the diffusion step with a positional encoding [273] and passes it together with $\boldsymbol{X}$ through the multilayer neural network. Here, there is no interaction between the points along the time dimension. The model, however, has the capacity to learn transformation based on time of observation. The second model is RNN based, that is, we pass the same concatenated input as before to a 2-layer bidirectional GRU [40] and use a single linear layer to project to the output dimension. Table D.2 shows that it is important to have interactions in the time dimension, regardless of the noise source, because otherwise we only learn the marginal distribution and the quality of the samples suffers.

D.2.2 Neural process

D.2.2.1 Dataset

We sample points from a Gaussian process to obtain a single time series. In the end, we have 8000 time series and 2000 test time series. We sample the number of time points from a Poisson distribution with $\lambda = 10$ but restrict the values to always be above 5 and below 50. The time points are sampled uniformly on $[0, 1]$. The observations are sampled from a multivariate normal distribution with mean zero and covariance obtained from an RBF kernel. The σ value in the kernel is uniformly sampled in $[0.01, 0.05]$ for each time series independently. Half of the sampled points are treated as unobserved while the rest are used as a context in the model.

D.2.2.2 Additional results

We test the hypothesis that using a stochastic process with similar properties to the data will lead to better performance. The difference to the neural process setup in Section 6.4 is that we fix the synthetic GP to always have $\sigma = 0.05$. As can be seen from Figure D.1, the marginal distribution will be equal regardless of which process and which kernel parameter we use. On the other hand, when we look at path probability $p(\boldsymbol{X})$, we notice better results when the noise process matches data properties (as was also shown in Table 6.1 and D.2). That means, while our model can reverse the process well, the qualitative properties of the sampled curves will be different. In particular, the curves will be *rougher* with increasing γ in OU and *smoother* with increasing σ in GP.

D.2.3 CSDI imputation

The imputation experiment presented in Sections 6.3.3 and 6.4 uses the original CSDI model [262] and only changes the noise to include the stochastic process source. In this case, the time points at which we evaluate the stochastic process are regular which does not reflect the true nature of the Physionet dataset. Here, we change the setup such that the measurements keep the actual time that has passed instead of rounding to the nearest hour. This is still in favour of the original paper as it only takes one measurement per hour and discards others if they are present. The model from Tashiro et al. [262] remains the same and we replace the independent normal noise with the GP noise with $\sigma \in \{0.005, 0.01, 0.02\}$.

We run each experimental setup 10 times with different data maskings (see Tashiro et al. [262] for more details) and report the results in Table D.3. We perform the Wilcoxon one-sided signed-rank test [45] and reject the null hypothesis that the expected RMSE values are the same when $p < 0.05$. As we can see, higher values of σ produce better results which makes sense since $\sigma = 0.005$ is, informally, closer to independent Gaussian sampling than $\sigma = 0.02$, which has stronger temporal dependency between the samples. We suspect 10%-missing case does not produce significant results due to noise. Using higher σ does not further improve the results.

Missingness:		10%		50%		90%	
Metrics:		RMSE	p-value	RMSE	p-value	RMSE	p-value
CSDI (baseline)		0.603±0.274	–	0.658±0.060	–	0.839±0.043	–
$\sigma =$	0.005	0.541±0.085	0.125	0.647±0.049	0.116	0.824±0.032	0.188
	0.01	0.575±0.195	0.125	0.640±0.050	**0.001**	0.823±0.028	**0.032**
	0.02	0.515±0.039	0.326	0.636±0.050	**0.001**	0.811±0.032	**0.001**

Table D.3: Imputation results averaged over 10 runs and p-value of Wilcoxon one-sided test.

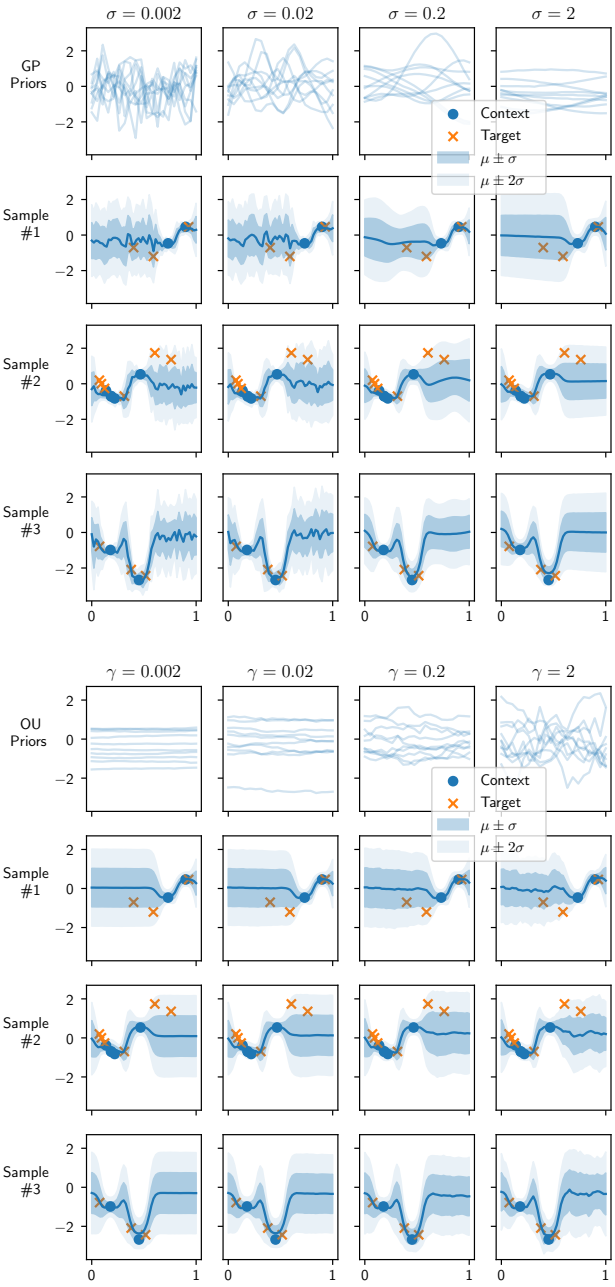


Figure D.1: (Top) Neural process with Gaussian process diffusion, fitted on GP synthetic data. Columns correspond to different values of the kernel parameter σ. The first row shows samples from the GP prior. As we can see, the higher the value of σ the smoother the process is. This is also reflected in the samples from the model. (Bottom) Same but for the Ornstein-Uhlenbeck process, however, increasing the kernel parameter γ now decreases the smoothness. All of the models perfectly capture the marginal distribution.