NEURAL PROBABILISTIC METHODS FOR EVENT SEQUENCE MODELING

by

Hongyuan Mei

A dissertation submitted to The Johns Hopkins University

in conformity with the requirements for the degree of

Doctor of Philosophy

Baltimore, Maryland

July, 2021

Abstract

This thesis focuses on modeling event sequences, namely, sequences of *discrete* events in *continuous* time. We build a family of generative probabilistic models that is able to reason about *what* events will happen in the future and *when*, given the history of previous events. Under our models, each event—as it happens—is allowed to update the future intensities of multiple event types, and the intensity of each event type—as nothing happens—is allowed to evolve with time along a trajectory.

We use neural networks to allow the "updates" and "trajectories" to be complex and realistic. In the purely neural version of our model, *all* future event intensities are conditioned on the hidden state of a continuous-time LSTM, which has consumed *every* past event as it happened. To exploit domain-specific knowledge of how an event might only affect *a few—but not all*—future event intensities, we propose to introduce domain-specific structure into the model. We design a modeling language, by which a domain expert can write down the rules of a temporal deductive database. The database tracks facts over time; the rules deduce facts from other facts and from past events. Each fact has a time-varying state, computed by a neural network whose topology is determined by the fact's provenance, including its experience of the past events that have contributed to deducing it. The possible event types at any time are given by special facts, whose intensities are neurally modeled alongside their states.

We develop efficient methods for training our models, and doing inference with them. Applying the general principle of noise-contrastive estimation, we work out a stochastic training objective that is less expensive to optimize than the log-likelihood, which people typically maximize for parameter estimation. As in the discrete-time case that inspired us, the parameters that maximize our objective will *provably* maximize the log-likelihood as well. For the scenarios where we are given incomplete sequences, we propose particle smoothing—a form of sequential importance sampling—to impute the missing events.

This thesis includes extensive experiments, demonstrating the effectiveness of our models and algorithms. On many synthetic and real-world datasets, on held-out sequences, we show empirically: (1) our purely neural model achieves competitive likelihood and predictive accuracy; (2) our neural-symbolic model improves prediction by encoding appropriate domain knowledge in the architecture; (3) for models to achieve the same level of log-likelihood, our noise-contrastive estimation needs considerably fewer function evaluations and less wall-clock time than maximum likelihood estimation; (4) our particle smoothing method is effective at inferring the ground-truth unobserved events.

In this thesis, I will also discuss a few future research directions, including embedding our models within a reinforcement learner to discover causal structure and learn an intervention policy.

Thesis Committee

Primary Readers

Jason Eisner (Primary Advisor)
Professor
Department of Computer Science
Johns Hopkins University

David Duvenaud
Assistant Professor
Department of Computer Science
University of Toronto

Minjie Xu
Research Scientist
Bloomberg L.P.

Yanxun Xu
Assistant Professor
Department of Applied Mathematics and Statistics
Johns Hopkins University

Previous Publications

Portions of this thesis have been published in slightly different forms in the following co-authored papers:

Chapter 3 Hongyuan Mei and Jason Eisner (2017), "The Neural Hawkes Process: A Neurally Self-Modulating Multivariate Point Process". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Chapter 4 Hongyuan Mei, Guanghui Qin, Minjie Xu, and Jason Eisner (2020). "Neural Datalog Through Time: Informed Temporal Modeling via Logical Specification". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Chapter 5 Hongyuan Mei, Chenghao Yang, and Jason Eisner (2021). "Continuous-Time Attention is All You Need". In: *Submission*.

Chapter 6 Hongyuan Mei, Tom Wan, and Jason Eisner (2020). "Noise-Contrastive Estimation for Multivariate Point Processes". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Chapter 7 Hongyuan Mei, Guanghui Qin, and Jason Eisner (2019), "Imputing Missing Events in Continuous-Time Event Streams". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Acknowledgments

First, I would like to thank my advisor, Jason Eisner, for shaping me into a researcher that I am proud of being. This is not a trivial task—it involves constantly raising standards as I improve, always making technical conversations engaging and insightful, often giving helpful advice on all the career-related matters, occasionally pushing[1] me to "think harder", knowing when to say—which happened rarely though—"you've done a great job",[2] and never refusing to support. He is my role model—I may never become as visionary or knowledgeable as he is, but if I can maintain a comparable level of enthusiasm or devotion over my career, then I'll count it as a success.[3]

My thesis committee members—David Duvenaud, Minjie Xu, Yanxun Xu, and, of course, Jason Eisner—have been supportive beyond the call of duty. They have provided valuable insight and held this work to a high standard; any remaining errors are of course my own. I am also grateful to my Graduate Board Oral (GBO) exam committee, which consists of—in addition to Jason Eisner and Yanxun Xu—Raman Arora, Mark Dredze, Carey Priebe, Sanjeev Khudanpur (alternate), and Benjamin Van Durme (alternate). Other professors have helped enrich my graduate experience by kindly sharing ideas and teaching great classes: Kevin Duh, Randal Burns, and

[1] This statement doesn't mean that Jason ever *physically* pushed me. Instead, it was always delivered *verbally*—in a refreshing but friendly voice.

[2] Of course, what happened rarely was saying, not knowing when to say.

[3] But he is not whom to blame for my abuse of footnotes.

Kyle Rawlins. I thank researchers from other places for their helpful discussions on various things at various times over my research career: Jacob Buckman, Shiyu Chang, Nan Du, Songyun Duan, Kevin Gimpel, John Lafferty, Jing Li, Lek-Heng Lim, Greg Shakhnarovich, Rakshit Trivedi, Karan Uppal, Hongteng Xu, Mo Yu, Yujie Zha, Qiang Zhang, Simiao Zuo; also anonymous reviewers on my papers. I am especially grateful to Mohit Bansal and Matthew Walter, who opened the door to machine learning research for me and are still offering continued mentoring.

My work during the academic years from 2018 through 2021 was generously supported by a Bloomberg Data Science Ph.D. Fellowship. It was also supported by research grants to Jason Eisner, including an Amazon Research Award, a Facebook Research Gift, and Grant No.1718846 from the U.S. National Science Foundation. Experiments included in this thesis were enabled by the Maryland Advanced Research Computing Center (MARCC), CLSP Grid, and two Titan X Pascal GPUs donated by NVIDIA Corporation.

As an Argonaut,[4] I am fortunate to have a group of talented labmates who are always generous at brainstorming ideas as well as giving feedback on my paper drafts and practice talks. They are: Ryan Cotterell, Nathaniel (Wes) Filardo, Matthew Francis-Landau, Xiang (Lisa) Li, Chu-Cheng Lin, Brian (Zhichu) Lu, Sabrina Mielke, Nanyun (Violet) Peng, Guanghui Qin, Pushpendre Rastogi, Adi Renduchintala, Tim Vieira, Tom Wan, Dingquan Wang, and Hao Zhu.

My student colleagues at Hopkins have fostered a fun and creative atmosphere. Thanks to: Nanxin Chen, Tongfei Chen, Kuan Cheng, Shuoyang Ding, Dongji Gao, Ruizhe Huang, Ruizhi Li, Xiaochen Li, Chenxi Liu, Zaoxing Liu, Hang Lv, Xutai Ma,

[4] https://www.cs.jhu.edu/~jason/Argo/

Arya McCarthy, Suzanna Sia, Shuo Sun, Zachary Wood-Doughty, Shijie Wu, Patrick Xia, Sheng Zhang, Xiaohui Zhang, Xuan Zhang, and Zhuotun Zhu.

My administrative colleagues have managed to save me from as much bureaucracy as they could. Thanks to: Zachary Burwell, Yamese Diggs, Kim Franklin, Ruth Scally, and the Bloomberg Fellowship program committee.

Finally, I thank my parents—Guozhen Zhang and Wanzhi Mei—for their unconditional love and support. They did their best to understand why I wanted to quit my job and "go back to school"; they never complained as I couldn't come back for six years in a row (from mid-2012 through early-2018); they never urged me to graduate and get a job; they always assure me that they are doing well and that I can focus on my career. It is their love and support that has sent me this far—I am the author of this thesis, but they are the authors of me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis is about modeling event sequences, namely, sequences of discrete events in continuous time. Such data is becoming ubiquitous in real-world applied domains:

Education An educational technology system tracks time-stamped activities of users, such as what lessons and tests they take, and how well they do.

Home automation The central hub of a home automation system records time-stamped interactions between its users and the devices connected to it, such as controlling light, heat, and security cameras.

Medicine Electronic health records (EHRs) store time-stamped information about patients' acute incidents, hospital visits, tests, diagnoses, and medications.

Online shopping An online shopping website logs customers' time-stamped actions, including browsing, purchasing, and leaving feedback for the merchandise.

Social network A social network platform tracks interactions among users such as their posts, likes, shares, comments, and messages.

Modeling such data is crucial to accurately predict the future of a sequence given its past, i.e., which events are likely to happen next and when they will happen. That

should benefit applications such as medical prognosis, consumer behavior analysis, and social media activity prediction.

The novelty of this thesis is a flexible new family of probabilistic models for event sequences, along with efficient training and inference algorithms. For our contributions to be appreciated, a certain amount of historical and technical background needs to be laid out. Therefore, this chapter aims to position this thesis in a broader context of sequential data modeling.

1.1 Modeling Sequences in the Era of Neural Networks

The current era of neural networks started in the 2010s:[1] AlexNet swept to victory in the ImageNet Large Scale Visual Recognition Challenge (Krizhevsky, Sutskever, and Hinton, 2012); deep neural networks achieved the state-of-the-art performance on a variety of speech recognition benchmarks (Hinton et al., 2012); AlphaGo defeated the human European Go champion by 5 games to 0 (Silver et al., 2016). Over this era, a large number of neural models have been created for various domains.

1.1.1 Abundance of Neural Models for *Regular* Sequences

Natural language processing (NLP) is a particular field that has been revolutionized by neural networks. In this field, researchers are interested in modeling sequences of linguistic words, and many problems involve reasoning about the future words (or suffix) of a sequence given its prefix (and possibly other contexts). Technically, this is handled by constructing and consulting a "language model", i.e., a locally normalized

[1] There are multiple eras of neural networks in the history: the first started in the 1950s, marked by the invention of perceptron (Rosenblatt, 1958); the second started in the 1980s, marked by the use of backpropagation in training neural networks (Rumelhart, Hinton, and Williams, 1986). It is fair to say that we are currently living in the third era of neural nets.

probability distribution over natural language sentences:

$$p(\text{sentence}) = \prod_{i=1}^{I} p(i\text{-th word} \mid \text{prefix before it})$$

A naive parametrization might define a distribution $p(\text{word} \mid \text{prefix})$ for each possible prefix, ending up with exponentially many parameters. Its generalization will suffer. For decades, NLP researchers compromised by using n-gram models: each word is only allowed to depend on a few most recent words, restricting the models to only have polynomially many parameters. Though performing strikingly well in many scenarios, n-gram models fail to capture long-range dependencies. For example, it will be difficult for a trigram model to make an appropriate continuation—obviously, "Jason" or "him"—for "Jason passed the ball to Hongyuan, who passed it back to".

But it's *not* hard for a *neural* language model! Generally speaking, a neural language model will use neural networks to summarize the *entire* prefix into a multidimensional vector, which is then passed through a non-linear function to define $p(\text{word} \mid \text{prefix})$. Such a vector is capable of memorizing all the information of the given prefix,[2] allowing the choice of the next word to be conditioned on words that are way behind. This increase of expressiveness comes at a reasonable cost—the neural networks may need many parameters to work well, but they don't have to grow with the length of the prefix we want to condition on.

What neural networks in particular do neural language models use? Nearly all of them use recurrent neural networks—specifically LSTM (Hochreiter and Schmidhuber, 1997)—or attention mechanism (Bahdanau, Cho, and Bengio, 2015) or both. The NLP field has been greatly advanced by these models (Mikolov et al., 2010; Sundermeyer,

[2]Because it lies in a continuous and infinite space.

Ney, and Schluter, 2012; Chelba et al., 2013), especially ELMo (Peters et al., 2018), BERT (Devlin et al., 2018), and GPTs (Radford et al., 2019; Brown et al., 2020).

It is straightforward to adapt these models to other kinds of sequences (e.g., genes), as long as they are *regular* sequences as well. We call natural language sentences as "regular sequences", because it always takes exactly *one* step to generate the *next* word. For example, the word right after the 4th word will always be the 5th word—it is meaningless to take 1.3 step and generate the 5.3-th word. Readers may laugh at the silliness of this example, and then question the author of this thesis "is there any *irregular* sequence at all such that it may take 1.3 step to generate an item in it?" The author would respond "yes, please read the next section."

1.1.2 Scarcity of Neural Models for *Irregular* Sequences

Event sequences are *irregular* in the sense that it may take any amount of time for an event to happen. In other words, time intervals between events are irregular. A principled way to handle such irregularity is to quantify the probability of what we observe at any time t, yielding a *continuous-time* "language model":

$$p(\text{event sequence}) = \exp\left(\int_{t=0}^{T} \log p(\text{observation at time } t \mid \text{history before } t)\right)$$

Note that the observation at any time t may be an event of a certain type, or "nothing". Such a model is called a multivariate point process, and it needs to define a bank of probability *trajectories*—one for each event type. Figure 1.1 shows the trajectories defined by a Poisson process (Palm, 1943), which assumes that events are independent of one another. The trajectory of "nothing" can be obtained by subtracting all these trajectories from the constant of 1.

A self-exciting process (Hawkes, 1971; Liniger, 2009) allows dependencies,

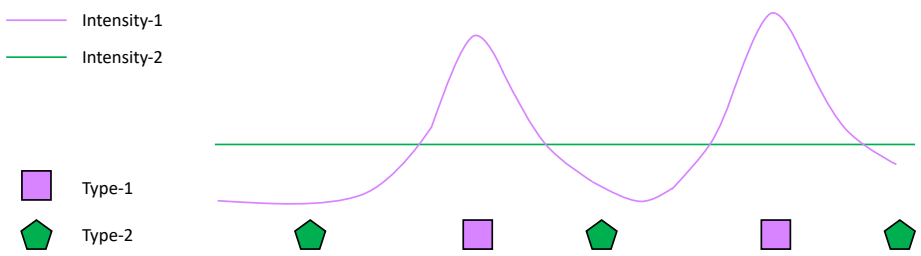


Figure 1.1: Typical probability trajectories of a Poisson process. Under this model, the probability of an event happening at time t is independent of other events, although it may vary with t.

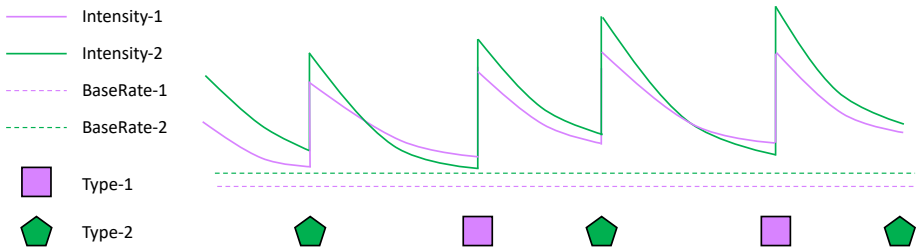


Figure 1.2: Typical probability trajectories defined by a Hawkes process (introduced shortly in section 2.1.2). Under this model, past events always raise the probabilities of future events, but the excitation effects will exponentially decay towards the base rates.

assuming that past events can temporarily *raise* the probability of future events. A self-correcting process (Isham and Westcott, 1979) supposes that past events can temporarily *suppress* the probability of future events. Typical probability trajectories defined by these models are shown in Figures 1.2 and 1.3.

Various *non-neural* versions of these models were developed,[3] but they all inherit some restrictive assumptions that real-world data often violates. For example, all variants of Hawkes process assume that past events always excite future events, then they'll all fail to capture context-dependent effects. Consider that Hongyuan eats cakes: when the cakes are delicious and he is hungry, eating a cake will raise the

[3]Examples will be given in Chapter 3

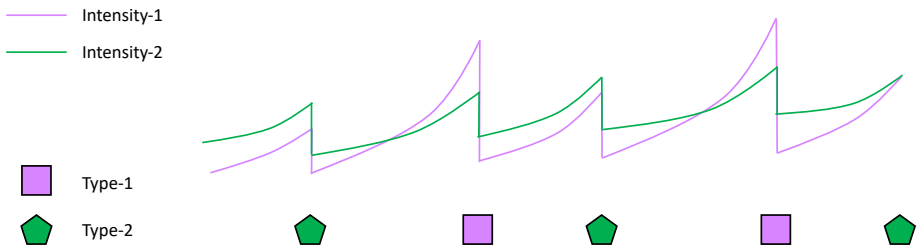


Figure 1.3: Typical probability trajectories defined by a self-correcting process. Under this model, probabilities increase with time t, but drop discontinuously as each event happens.

probability that Hongyuan eats another; but eating one more cake may suppress the probability that Hongyuan eats another, since he is already full.

In retrospect, using neural networks seems an extremely obvious way to achieve greater model capacity; however, till 2016, the field of event sequence modeling was still experiencing a surprising scarcity of neural models.

1.1.3 Call for Neural Networks: Opportunities and Challenges

Opportunities exist in designing neural models to capture effects that non-neural models miss. A natural idea is to extend neural language models to handling event sequences. After all, when applied to language modeling, neural networks already capture many kinds of complex sequential dependencies that are similar to those in event sequences. But challenges also arise, as I'll discuss in the following sections.

1.1.3.1 Flexibility

Obviously, the first and most important question is how to extend existing neural language models to irregular sequences. A naive application of a neural language model would work as follows: the model reads each event—with its timing—as it happens, and the updated state vector defines a new bank of probability trajectories; the trajectories are flat, since the state vector stays constant before the next update. This

naive model can capture many complicated and realistic update patterns, including the context-dependent effects in the aforementioned "Hongyuan eats cakes" example.

But such a naive model will fail to capture many real-world patterns. Consider that Hongyuan receives a shot of COVID vaccine: the probability that Hongyuan has allergic reaction will rise up after the shot—but *not* immediately—and this is why he'll be monitored at the vaccination site for fifteen minutes. But only fifteen minutes, because the probability of allergic reaction will eventually decrease. This bell-shaped trajectory is apparently beyond what the naive model can offer.

There are a couple of easy improvements, but at unpleasant costs. The first is to feed many special "clock tick" events into the model as nothing happens, approximating the bell-shaped trajectory with a piecewise constant function. This approach will waste a lot of computation when nothing happens at all. The other is to allow the probability trajectory to follow a parametric form (e.g., exp, lognormal), and use the state vector to define its parameters. This approach may increase approximation error: e.g., what if we use a lognormal function but the actual trajectory is a bathtub curve?

Therefore, it is really challenging to design a neural model that is able to capture all the complex and realistic patterns in event sequences.

1.1.3.2 Scalability

Suppose that we already have a flexible and purely neural model. Under such a model, each event updates the state vector, which then determines the distribution from which the next event is drawn. Alas, when the relationship between events and the neural state is *unrestricted*—when anything can potentially affect anything—fitting an accurate model is very difficult, particularly in a real-world domain that allows millions of event types including many rare types.

Consider a domain of many people: we track their locations, as well as what events can happen given their locations. For example, we might know that Eve is in New York City, and that her friend Adam is in Chicago. Given these facts, it is possible that Eve travels to Chicago. Once she actually does, her location and the set of possible events will be updated: Eve can't "travel to Chicago" anymore because she is already there; Eve and Adam may have a dinner together since they are in the same city.

Can we expect a purely neural model to *learn* how to encode every location fact in some multi-dimensional state vector? Or how to deduce the set of possible events from these location facts? Or how to update these location facts after each travel event? It might learn well for *a few* people and places—if the model has seen travel events involving them often enough in the training data. But it will be difficult to *generalize* across people and places.

1.1.3.3 Efficiency

It will also be a challenge to efficiently train a neural model and do inference with it.

Challenge for Training: Expensive Objective Maximum likelihood estimation is a popular training method for generative models. However, to obtain the likelihood of a generative model given the observed data, one must compute the probability of each observed sample, which often includes an expensive normalizing constant. For example, in a language model, each word is typically drawn from a softmax distribution over a large vocabulary, whose normalizing constant requires a summation over the vocabulary. There is a similar computational cost for multivariate point processes. Their likelihood is improved not only by raising the probability of the observed events, but by lowering the probabilities of the events that were observed *not* to occur. There are infinitely many times at which no event of any type occurred;

to predict these *non*-occurrences, the likelihood must integrate the event probability for each event type over the entire observed time interval. Therefore, the likelihood is expensive to compute, particularly when there are many possible event types. This problem will be worsen if each intensity is computed through a neural network.

Challenge for Inference: Intractable Posterior In many real-world applications, we are only given incomplete event sequences, and we need to impute the missing events. For example, in poker or StarCraft, a player lacks full information about what her opponents have acquired (cards) or done (build mines and train soldiers). Accurately imputing hidden actions from "what I did" and "what I observed others doing" can help the player make good decisions. Suppose we already have a trained model $p(\text{complete sequence})$, as well as the "missingness mechanism" $p(\text{missing events} \mid \text{complete sequence})$, which stochastically determines which of the events will not be observed. In principle, one can use Bayes' Theorem to define the posterior distribution $p(\text{complete events} \mid \text{observed events})$; in practice, this posterior is computationally difficult to reason about. Even for a simple $p(\text{complete sequence})$ like a Hawkes process, Markov chain Monte Carlo (MCMC) methods are needed, and these methods obtain an efficient transition kernel only by exploiting special properties of the process (Shelton, Qin, and Shetty, 2018). Unfortunately, such properties will no longer hold for a more flexible neural model.

1.2 Our Approach: Neural Probabilistic Methods

We aim to develop a flexible new family of neural probabilistic models, along with efficient training and inference algorithms, to resolve the aforementioned challenges.

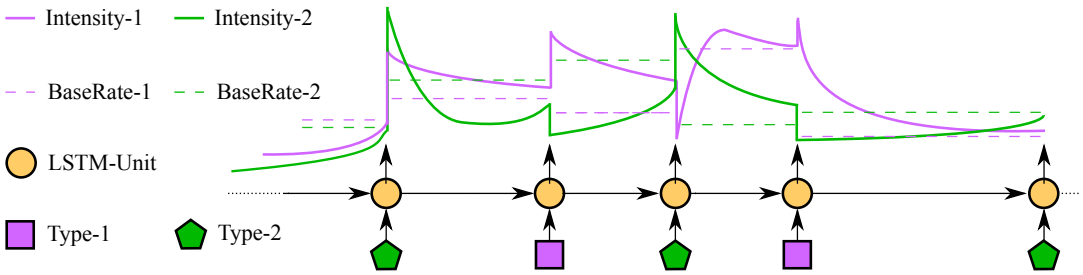


Figure 1.4: Drawing an event stream from a neural Hawkes process. An LSTM reads the sequence of past events (polygons) to arrive at a hidden state (orange). That state determines the future "intensities" of the two types of events—that is, their time-varying instantaneous probabilities. The intensity functions are continuous parametric curves (solid lines) determined by the most recent LSTM state, with dashed lines showing the steady-state asymptotes that they would eventually approach. In this example, events of type 1 excite type 1 but inhibit type 2. Type 2 excites itself, and excites or inhibits type 1 according to whether the count of type 2 events so far is odd or even. Those are immediate effects, shown by the sudden jumps in intensity. The events also have longer-timescale effects, shown by the shifts in the asymptotic dashed lines.

1.2.1 Neuralizing Intensity Functions for Flexibility

Our neural model relies on a novel continuous-time LSTM. The LSTM state is a deterministic function of the past history. It plays the same role as the state of a deterministic finite-state automaton. However, our network enjoys a continuous and infinite state space (a high-dimensional Euclidean space), as well as a learned transition function. In our network design, the state vector is not only updated *discontinuously* with each successive event occurrence, but also evolves *continuously* as time elapses between events. We determine the event probabilities at any time t from the state vector at that time, allowing past events to influence the future in complex and realistic ways. We call our model the *neural* Hawkes process since this idea was initially inspired by the Hawkes process. Figure 1.4 shows a bank of complicated probability trajectories that a neural Hawkes process is able to handle.

1.2.2 Integrating a Deductive Database for Scalability

We propose to address the scalability challenge by introducing domain-specific structure into the model. Specifically, for the travel domain in section 1.1.3.2, one might declare that the probability that Eve travels to Chicago is determined entirely by Eve's state, the states of Eve's friends such as Adam, and the state of affairs in Chicago. Given that modeling assumption, parameter estimation can no longer incorrectly overfit this probability using spurious features based on unrelated temporal patterns of (say) wheat sales and soccer goals. To improve generalization, one can reuse this "Eve travels to Chicago" model for any person `A` traveling to any place `C`.

Our main contribution is a modeling language that can concisely model all these `travel(A,C)` probabilities using a few rules over variables `A`, `B`, `C`. Here `B` ranges over `A`'s friends, where the `friend` relation is also governed by rules and can itself be affected by stochastic events. In our system, a domain expert simply writes down the rules of a temporal deductive database, which tracks the possible event types and other *boolean* facts over time. This logic program is then used to automatically construct a deep recurrent neural architecture, whose distributed state consists of vector-space *embeddings* of all present facts. Its output specifies the distribution of the next event.

The number of parameters of such a model grows only with the number of rules, not with the much larger number of event types or other facts. This is analogous to how a probabilistic relational model (Getoor and Taskar, 2007; Richardson and Domingos, 2006) derives a graphical model structure from a database, building random variables from database entities and repeating subgraphs with shared parameters.

Unlike graphical models, ours is a neural-symbolic hybrid. The system state includes both rule-governed discrete elements (the set of facts) and learned continuous

elements (the embeddings of those facts). It can learn a neural probabilistic model of people's movements while relying on a discrete symbolic deductive database to cheaply and accurately record who is where.

1.2.3 Approximate Algorithms for Efficiency

We propose efficient methods for estimating parameters of our models, and imputing missing events with a trained model.

Efficient Training As an alternative to maximum likelihood estimation, we propose to train the model by learning to discriminate the observed events from events sampled from a noise process. Our method is a version of noise-contrastive estimation—a general parameter estimation method with a less expensive stochastic objective. This method was originally developed for unnormalized (energy-based) distributions and then extended to conditional softmax distributions such as language models. We generalize it to multivariate point processes, along with its theoretical guarantees for optimality, consistency, and efficiency. Particularly, like in the instantiation for language models, the parameters that maximize our training objective will *provably* maximize the log-likelihood as well.

Efficient Inference of Missing Events Given a trained model of complete sequences, we propose to use sequential importance sampling to approximate the posterior distribution $p(\text{complete events} \mid \text{observed events})$.

The key is to construct a good proposal distribution. A naive version is to draw each event given the complete history of previous events, and add any observed event to the sequence as they happen. Alas, it is computationally inefficient. For a complete sequence to be actually probable under the posterior, this naive proposal distribution must have the good fortune to draw only events that happen to be consistent with future

observations. Such lucky particles would appropriately get a high weight relative to other particles. The problem is that we will rarely get such particles at all.

We develop a trainable family of proposal distributions based on a type of bidirectional continuous-time LSTM: bidirectionality lets the proposals condition on future observations, not just on the past as in the naive approach. We still have to reweight our particles to match the actual posterior under the model. But this reweighting is not as drastic as for the naive approach, because the new proposal distribution was constructed and trained to resemble the actual posterior. This idea is called particle smoothing (Doucet and Johansen, 2009).

1.3 The Roadmap

The thesis proceeds as follows. Chapter 2 introduces the technical background about probabilistic modeling of event sequences. Particularly, it gives an overview of multivariate temporal point processes, along with typical methods for estimating parameters (via maximum likelihood estimation) and sampling predictions of the future (via rejection sampling). This chapter aims to prepare readers for later chapters which focus on our specific methodological innovations. A reader who's familiar with this field may feel free to skip this chapter.

Chapter 3 presents the neural Hawkes process (Mei and Eisner, 2017)—our flexible purely neural model that can capture complex patterns in real-world sequences. This chapter describes its model architecture, and explains why it has desirable qualitative properties. On multiple synthetic and real-world datesets, the neural Hawkes process achieves superior likelihood and predictive accuracy on held-out sequences, including under missing-data conditions.

Chapter 4 presents the neural Datalog through time (Mei et al., 2020)—our neural-symbolic hybrid whose model architecture is determined by a temporal deductive database. This chapter introduces our modeling language, by which a user can write down domain-specific knowledge as rules. Then it shows how this logic program automatically derives a deep recurrent neural network. This neural-symbolic model achieves better prediction accuracy than strong competitors, including our neural Hawkes process.

Chapter 5 presents the *attentive* neural Hawkes process and the *attentive* neural Datalog through time (Mei, Yang, and Eisner, 2020). This chapter introduces our continuous-time Transformer architecture, which is able to embed an actual or possible event by looking at the previous actual events. Then it shows how to construct generative point process models based on this architecture, yielding Transformer-based variants of the models introduced in Chapter 3 and Chapter 4. The new attentive models match or surpass the previous versions on several synthetic and real-world datasets, evaluated by likelihood and prediction accuracy.

Chapter 6 presents our instantiation of noise-contrastive estimation for multivariate point processes (Mei, Wan, and Eisner, 2020). This chapter reviews the general principle of this parameter estimation method, and its application in training neural language models. Then it shows how to extend the method to our models, as well as analyzes its theoretical guarantees and runtime complexity. Empirically, this method indeed takes much less wall-clock time while still achieving competitive likelihood.

Chapter 7 presents our particle methods for imputing missing events (Mei, Qin, and Eisner, 2019). This chapter shows how to construct a smart proposal distribution that can be trained to condition proposals on future observations, not only on the

past. It also shows how to turn an ensemble of possible complete sequences into a single consensus prediction that has low Bayes risk under our chosen loss metric. Our particle smoothing method is effective at inferring the ground-truth unobserved events, consistently improving upon particle filtering.

Chapter 8 concludes this thesis, and plans out several exciting avenues for future work. Particularly, this chapter proposes several ways to deploy our model family—as an environment model—within a reinforcement learner to discover causal structure and learn an intervention policy. It also discusses the potential impact of such a reinforcement learner—improving the future course of events in a medical, economic, or social event sequence.

Chapter 2

Formalism

We have stated that we aim to develop novel machine learning methods for probabilistic modeling of event sequences. In this chapter, I introduce the formalism for constructing probability distributions over event sequences, followed by typical methods used for fitting a distribution to data and reasoning about a distribution.

This chapter aims to familiarize readers with some machine learning concepts and techniques in the context of event sequence modeling, including point process, intensity function, maximum likelihood estimation, and thinning algorithm. Readers may feel free to skip this chapter if they are already familiar with these terms.

2.1 Multivariate Point Process

Given a fixed time interval $[0, T)$, we may observe an **event sequence** $x_{[0,T)}$: at each continuous time t, the observation x_t is one of the discrete types $\{\varnothing, 1, \ldots, K\}$ where $\varnothing$ means *no event*. An non-$\varnothing$ observation is called an **event**. That is, there are K types of events, tokens of which are observed to occur in continuous time.

A generative probabilistic model of an event sequence is called a **multivariate point process**. It assumes that an event of type $k \in \{1, \ldots, K\}$ occurs at time

t—more precisely, in the infinitesimally wide interval $[t, t + dt)$—with probability $\lambda_k(t \mid x_{[0,t)})dt$. The value $\lambda_k(t \mid x_{[0,t)}) \geq 0$ depends on the history of events $x_{[0,t)}$ that were drawn at times $< t$. It can be regarded as a rate per unit time: $\lambda_k(t \mid x_{[0,t)})$ is the limit as $dt \to^+ 0$ of $\frac{1}{dt}$ times the expected number of events of type k on the interval $[t, t + dt)$, where the expectation is conditioned on the history. λ_k is known as the **intensity function**, and the total intensity of all event types is given by $\lambda(t \mid x_{[0,t)}) = \sum_{k=1}^{K} \lambda_k(t \mid x_{[0,t)})$. As the event probabilities are infinitesimal, we have $p(x_t = \varnothing) = 1$ at any time t, and the times of the events are almost surely distinct.

Specific point processes differ in the design of their intensity functions. In the following subsections, I will introduce a series of examples.

2.1.1 Poisson Process

A basic model is the Poisson process (Palm, 1943), which assumes that events occur independently of one another. That is, its intensity functions λ_k ignore the history. In a homogenous Poisson process, the intensities are constant:

$$\lambda_k(t \mid x_{[0,t)}) = \mu_k \tag{2.1}$$

where $\mu_k \geq 0$ is a learnable parameter.

In a non-homogenous Poisson process, the intensity of an event happening at time t may vary with t, but it is still independent of other events. Figure 1.1 displays a bank of typical intensity trajectories of a Poisson process.

2.1.2 Hawkes Process

A well-known generalization that captures interactions is the Hawkes process (Hawkes, 1971; Liniger, 2009), in which past events from the history conspire to *raise* the

intensity of each type of event. Such excitation is positive, additive over the past events, and exponentially decaying with time:

$$\lambda_k(t \mid x_{[0,t)}) = \mu_k + \sum_{r:r<t, x_r \neq \varnothing} \alpha_{x_r,k} \exp(-\delta_{x_r,k}(t-r)) \tag{2.2}$$

where $\mu_k \geq 0$ is the base intensity of event type k, $\alpha_{j,k} \geq 0$ is the degree to which an event of type j initially excites type k, and $\delta_{j,k} > 0$ is the decay rate of that excitation. When an event occurs, all intensities are elevated to various degrees, but then will decay toward their base rates μ_k. Figure 1.2 displays a bank of typical intensity trajectories of a Hawkes process.

This model is also called the self-exciting multivariate point process (SE-MPP).

2.1.3 Generalized Hawkes Process

The positivity constraints in the Hawkes process limit its expressivity. First, the positive interaction parameters $\alpha_{j,k}$ fail to capture inhibition effects, in which past events *reduce* the intensity of future events. Second, the positive base rates μ_k fail to capture the inherent inertia of some events, which are unlikely until their cumulative excitation by past events crosses some threshold. One may relax the positivity constraints on $\alpha_{j,k}$ and μ_k, allowing them to range over $\mathbb{R}$, which allows *inhibition* ($\alpha_{j,k} < 0$) and *inertia* ($\mu_k < 0$). However, the resulting total activation could now be negative. It therefore should be passed through a non-linear transfer function $f_k : \mathbb{R} \to \mathbb{R}_+$ to obtain a positive intensity function as required:

$$\lambda_k(t \mid x_{[0,t)}) = f_k(\mu_k + \sum_{r:r<t, x_r \neq \varnothing} \alpha_{x_r,k} \exp(-\delta_{x_r,k}(t-r))) \tag{2.3}$$

As t increases between events, the intensity $\lambda_k(t)$ may both rise and fall, but eventually approaches the base rate $f_k(\mu_k + 0)$, as the influence of each previous event

still decays toward 0 at a rate $\delta_{j,k} > 0$. This is the generalized Hawkes process we proposed in Mei and Eisner (2017). Our choice of f_k is the scaled "softplus" function $f(x) = s \log(1 + \exp(x/s))$, which approaches ReLU as $s \to 0$. A detailed discussion about this function can be found in section 2.A. We learn a separate scale parameter s_k for each event type k, which adapts to the rate of that type.

This model is also called the decomposable self-modulating multivariate point process (D-SM-MPP): it is more expressive than Hawkes process while still maintaining its decomposable structure.

2.2 Training a Point Process

Maximum likelihood estimation (MLE) is the most common method for estimating the parameters $\boldsymbol{\theta}$ of a generative probabilistic model. Suppose that $p_{\boldsymbol{\theta}}$ is the probability density defined by the model. The MLE estimate is given by:

$$\hat{\boldsymbol{\theta}}_{\text{MLE}} \stackrel{\text{def}}{=} \arg\max_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\text{data}) \tag{2.4}$$

In practice, for numerical stability, the *log-likelihood* is used in lieu of the likelihood:

$$\hat{\boldsymbol{\theta}}_{\text{MLE}} \stackrel{\text{def}}{=} \arg\max_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\text{data}) \tag{2.5}$$

For a point process, the log-likelihood turns out to be given by a simple formula—the sum of the log-intensities of the events that happened, at the times they happened, minus an integral of the total intensities over the observation interval $[0, T)$:

$$\log p_{\boldsymbol{\theta}}(x_{[0,T)}) = \sum_{t : x_t \neq \varnothing} \log \lambda_{x_t}(t \mid x_{[0,t)}) - \underbrace{\int_{t=0}^{T} \lambda(t \mid x_{[0,t)}) \mathrm{d}t}_{\text{call this } \Lambda} \tag{2.6}$$

The full derivation is given in section 2.B. Intuitively, the $-\Lambda$ term (which is ≤ 0)

sums the log-probabilities of infinitely many *non*-events. Why? The probability that there was *not* an event of any type in the infinitesimally wide interval $[t, t + dt)$ is $1 - \lambda(t)dt$, whose log is $-\lambda(t)dt$.

How should we find the $\hat{\boldsymbol{\theta}}_{\text{MLE}}$ then?

2.2.1 When a Closed-Form Solution Exists ...

Let's first look at an extremely simple case—the homogeneous Poisson process. With intensities being constant, its log-likelihood can be simplified to be:

$$\log p_{\boldsymbol{\theta}}(x_{[0,T)}) = \sum_{t:x_t \neq \varnothing} \log \mu_{x_t} - T \sum_{k=1}^{K} \mu_k \tag{2.7}$$

Solving the first order necessary conditions yields the closed-form solution as below:

$$\hat{\mu}_k = \sum_{t:x_t=k} \frac{1}{T} \tag{2.8}$$

That is, each rate μ_k is estimated to be the empirical rate of events of type k. Intuitively, if a process has a higher μ_k, it would expect to observe more events of type k; a process with a lower μ_k would expect to observe fewer events of that type.

Note that the closed-form solution we just derived is very particular to the Poisson process. In general, a closed-form solution may not exist. What should we do then?

2.2.2 When a Gradient Method Is Needed ...

When a closed-form solution doesn't exist, we can locally maximize the log-likelihood from equation (2.6) using any stochastic gradient method. For this, we need to be able to get an unbiased estimate of the gradient $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(x_{[0,T)})$ with respect to the model parameters $\boldsymbol{\theta}$. In principle, this is straightforward to obtain by back-propagation.

The tricky part is the integral Λ. In some models, it may be exactly computed. For

Algorithm 2.1 Integral Estimation (Monte Carlo)

Input: model parameters and an event sequence $x_{[0,T)}$
Output: estimated integral Λ and its gradient $\nabla\Lambda$

procedure ESTIMATEINTEGRAL(model, $x_{[0,T)}$)
  ▷ *estimate the integral Λ and its gradient $\nabla\Lambda$ by Monte Carlo approximation*
  $\Lambda \leftarrow 0$; $\nabla\Lambda \leftarrow \mathbf{0}$
  for N samples : ▷ *e.g., take $N > 0$ proportional to T*
    draw $t \sim \mathrm{Unif}(0, T)$
    for $j \leftarrow 1$ **to** K :
      Λ += $\lambda_k(t)$ ▷ *via current model parameters*
      $\nabla\Lambda$ += $\nabla\lambda_k(t)$ ▷ *via back-propagation*
  $\Lambda \leftarrow T\Lambda/N$; $\nabla\Lambda \leftarrow T\nabla\Lambda/N$ ▷ *weight the samples*
  return $(\Lambda, \nabla\Lambda)$

example, the integral Λ of the Hawkes process can be spelled out as:

$$T \sum_{k=1}^{K} \mu_k + \sum_{t:x_t \neq \varnothing} \sum_{k=1}^{K} \frac{\alpha_{x_t,k}}{\delta_{x_t,k}} \left(1 - \exp(-\delta_{x_t,k}(T - t))\right) \tag{2.9}$$

But can we expect to derive a simple formula like this for any given model? Unfortunately not. For example, the generalized Hawkes process in section 2.1.3 doesn't have such a simple formula since its softplus function complicates the intensity function.

The insight for handling the integral in the general case is that the single function evaluation $\lambda(t)$ at a random $t \sim \mathrm{Unif}(0, T)$ gives an unbiased estimate $T\lambda(t)$ of the entire integral—that is, the expected value is Λ. Its gradient via back-propagation is therefore a unbiased estimate of $\nabla\Lambda$.[1] The Monte Carlo algorithm in Algorithm 2.1 averages over several (N) samples to reduce the variance of this noisy estimator.

Each step of training computes the gradient on a training sequence. With P params, this takes time $O((I + N)P)$, if I is the number of observed events and N is the number of samples used to estimate the integral. We take $N = O(I)$ in practice,

[1] Since gradient commutes with expectation.

so we have runtime $O(IP)$ like Hawkes.

Note that our stochastic gradient is unbiased for any N; large N merely reduces its variance. The gradient for the Hawkes process has 0 variance, since it has analytical form and does not require sampling at all.

Note that uniform sampling (Algorithm 2.1) is not the lowest-variance way to compute the integral of a non-negative function; it would be better to take proportionately more samples where the function is higher, although this will only sometimes be worth the extra overhead. One option is to use adaptive quadrature (Baran, Demaine, and Katz, 2008), which is biased though.

Another option falls out when we regard the integral $\int_{t=0}^{T} \sum_{k=1}^{K} \lambda_k(t \mid x_{[0,t)}) \mathrm{d}t$ as the expected total number of events yielded by the point process where the intensities $\lambda_k(t \mid x_{[0,t)})$ depend on the true history $x_{[0,t)}$. To justify it more clearly: for each possible pair of (t, k), we have an independent Bernoulli variable that is 1 with probability $\lambda_k(t \mid x_{[0,t)})\mathrm{d}t$—we'd like to find the expected sum of these Bernoulli variables, which is the same as the sum of the expectations. An easy solution is to estimate the expectation of each Bernoulli variable by sampling it: we draw samples from the true process and then add up all these samples—almost surely, only finitely many of the samples will be 1, and we just count them. This option turns Algorithm 2.1 around: instead of computing the expectation of the total intensity (hard to compute) under a uniform distribution of times (easy to sample), we can compute the expectation of 1 (easy to compute) under a point-process distribution of times (hard to sample); this means that the sampling distribution is indeed focused on (t, k) with larger intensities. But how should we sample those times from a point process? In the next section, I will present an algorithm that draws a *full* sequence from a point process; in Chapter 6,

Algorithm 2.2 Data Simulation (Thinning Algorithm)

Input: interval $[0, T)$; model parameters
Output: sample sequence $x_{[0,T)}$

procedure DRAWSEQUENCE(T)
  ▷ *draw an event sequence over interval* $[0, T)$
  $x_{[0,T)} \leftarrow$ a sequence of no events ▷ $x_t = \varnothing$ *for all* $t \in [0, T)$
  $t_{\text{last}} \leftarrow 0$
  while $t_{\text{last}} < T$: ▷ *draw next event, as it might fall in* (t_{last}, T)
    $t_{\text{next}}, k_{\text{next}} \leftarrow$ DRAWEVENT(t_{last})
    if $t_{\text{next}} < T$: $x_{t_{\text{next}}} \leftarrow k_{\text{next}}$
    $t_{\text{last}} \leftarrow t_{\text{next}}$
  return $x_{[0,T)}$
procedure DRAWEVENT(t_{last})
  ▷ *thinning algorithm: draw next event after time* t_{last}
  for $k = 1$ **to** K : ▷ *draw "next" event of each type*
    find upper bound $\lambda^* \geq \lambda_k(t \mid x_{[0,t)})$ for all $t \in (t_{\text{last}}, \infty)$
    $t \leftarrow t_{\text{last}}$
    repeat
      draw $\Delta \sim \text{Exp}(\lambda^*)$, $u \sim \text{Unif}(0, 1)$
      $t \mathrel{+}= \Delta$ ▷ *time of next proposed event*
    until $u\lambda^* \leq \lambda_k(t \mid x_{[0,t)})$ ▷ *accept proposal with prob* $\lambda_k(t \mid x_{[0,t)})/\lambda^*$
    $t_k \leftarrow t$
  return $\min_k t_k, \arg\min_k t_k$

I will give a slightly modified version that draws each Bernoulli variable conditioned on the true history $x_{[0,t)}$ (section 6.3.3), and then show how to use that algorithm to yield a better estimate of the integral $\int_{t=0}^{T} \lambda(t \mid x_{[0,t)}) \mathrm{d}t$ (section 6.6).

2.3 Sampling From a Point Process

If we wish to draw a sequence from a point process, we should draw events from left to right, as shown in the DRAWSEQUENCE procedure of Algorithm 2.2. Each event is drawn by calling the thinning algorithm (Lewis and Shedler, 1979; Liniger, 2009) as shown in the DRAWEVENT procedure of Algorithm 2.2.

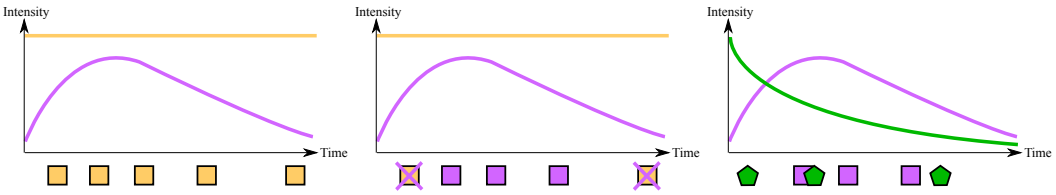


Figure 2.1: Sampling the next event, using the same visual notation as in Figure 1.4. The x axis shows a prefix of the infinite interval $(t_{\text{last}}, \infty)$. In the first graph, *gold* events are proposed from a homogeneous Poisson process with intensity λ^* (gold straight line). In the second graph, the purple curve $\lambda_1(t \mid x_{[0,t)})$ randomly accepts some of these gold events, with probability $\lambda_1(t \mid x_{[0,t)})/\lambda^*$ for the event at time t; here it accepts three of the ones shown and rejects the others. In the third graph, the surviving type-1 events (purple squares) are interleaved with the surviving type-2 events (green pentagons). The next event is the earliest one among these surviving candidates. In practice, these sequences are constructed lazily so that we find only the earliest surviving event of each type. This is possible because the inter-arrival times between gold proposed events are distributed as $\text{Exp}(\lambda^*)$, making it straightforward to enumerate any finite prefix of a random infinite gold sequence.

2.3.1 Thinning Algorithm for Drawing an Event

We explain the thinning algorithm here and illustrate its conception in Figure 2.1. Suppose we have already sampled the prefix over the interval $[0, t_{\text{last}}]$ where the last event happened at time t_{last}. The K event types are now in a race to see who generates the next event. Typically, the winning type will have relatively high intensity.

How do we conduct the race? For each event type k, $\lambda_k(t \mid x_{[0,t)})$ is the model intensity at time t provided that nothing has happened in the interval (t_{last}, t). For each k independently, we draw the time t_k of the next event from a non-homogeneous Poisson process whose intensity is defined as $\lambda_k(t \mid x_{[0,t)})$ for $t \in (t_{\text{last}}, \infty)$. We then take $t_{\text{next}} = \min_k t_k$ and $k_{\text{next}} = \arg\min_k t_k$. That is, we keep just the earliest of the K events. We cannot keep the rest because they are not correctly distributed according to the new intensities as updated by the earliest event.

But how do we draw the next event time from each univariate proces? A draw from such a process is actually a whole *set* of times in $(t_{\text{last}}, \infty)$: we will take

t_k to be the earliest of these. In theory, this set is drawn by *independently* choosing at each time $t \in (t_{\text{last}}, \infty)$, with infinitesimal probability proportional to $\lambda_k(t \mid x_{[0,t)})$, whether an event occurs. One could do this by *independently* applying rejection sampling at each time t: choose with larger probability λ^* whether a "proposed event" occurs at time t, and if it does, accept the proposed event with probability only $\lambda_k(t \mid x_{[0,t)})/\lambda^* \leq 1$. This is equivalent to simultanously drawing a set of proposed times from a *homogenous* Poisson process with constant rate λ^*, and then "thinning" that proposed set, as illustrated in Figure 2.1. This approach helps because it is easy to draw from the homogenous process: the intervals between successive proposed events are IID $\text{Exp}(\lambda^*)$, so it is easy to sample the events in sequence. The inner **repeat** loop (line 15) in Algorithm 2.2 lazily carries out just enough of this infinite homogenous draw from λ^* to determine the time t_k of the earliest *accepted* event, which is the earliest event in the non-homogeneous draw from λ_k, as desired.

Finally, how do we construct the upper bound λ^*? This is really specific to the chosen model. For the Hawkes process, since the excitation is always positive and decaying, the tightest upper bound turns out to be the model intensity immediately after the last event happened, i.e., $\lambda^* = \lim_{t \to t_{\text{last}}^+} \lambda_k(t \mid x_{[0,t)})$. For the generalized Hawkes process, the intensity in equation (2.3) is of the form $f_k(\mu + g_1(t) + \cdots + g_n(t))$ where f_k is monotonically non-decreasing and each g is a *bounded* function on $(t_{\text{last}}, \infty)$. We can therefore replace each g function by its upper bound to obtain $\lambda^* = f_k(\mu + \max_t g_1(t) + \cdots + \max_t g_n(t))$. Specifically, each summand $\alpha_{x_r,k} \exp(-\delta_{x_r,k}(t-r))$ is upper-bounded by $\max(\alpha_{x_r,k}, 0)$.

2.3.2 A More Efficient Version of Thinning Algorithm

While the DRAWEVENT procedure in Algorithm 2.2 is classical and intuitive, it is also beneficial to consider a more efficient variant. Instead of drawing the next event from each of K *different* non-homogeneous Poisson processes and keeping the earliest, we can construct a *single* non-homogenous Poisson process with aggregate intensity function $\lambda(t \mid x_{[0,t)}) = \sum_{k=1}^{K} \lambda_k(t \mid x_{[0,t)})$ over $(t_{\text{last}}, \infty)$. An upper bound λ^* on this aggregate function can be obtained by summing the upper bounds on the individual λ_k functions. We then use the thinning algorithm only to sample the next event time t_{next} from this aggregate process λ. Finally, we "disaggregate" by choosing k_{next} from the distribution $p(x_t = k \mid x_t \neq \varnothing) = \lambda_k(t \mid x_{[0,t)})/\lambda(t \mid x_{[0,t)})$.[2] This is equivalent to Algorithm 2.2. In terms of Figure 2.1, this more efficient version enumerates a gold sequence that is the union of the K gold sequences, and stops with the first accepted gold event. Thus, whereas Figure 2.1 had to propose two type-1 events in order to get the first accepted type-1 event (the leftmost purple event), the more efficient version would not have had to spend time proposing either of those, because an earlier proposed event (the leftmost green event) had already been accepted and determined to be of type 2.

2.4 Inference with a Point Process

We have seen how to train a point process and how to sample from it. They are not very useful on their own. We still need inference methods to make sense out of the trained model. Minimum Bayes-risk (MBR) decoding is a general principle for doing

[2] In practice, acceptance and disaggregation can be combined into a single step. That is, each successive event t proposed from the homogeneous Poisson(λ^*) process is either kept as type k, with probability $\lambda_k(t)/\lambda^*$, or rejected, with probability $1 - \lambda(t)/\lambda^*$. If it is accepted, we have found our next event $x_t = k$. If it is rejected, we increment t by $\Delta \sim \text{Exp}(\lambda^*)$ to get the next proposed event.

inference with a probabilistic model. For any specific query, MBR decoding aims to find the hypothesis h that has the *lowest* expected loss L under the model $p_{\boldsymbol{\theta}}$:

$$\hat{h}_{\text{MBR}} \stackrel{\text{def}}{=} \arg\min_h \sum_{h^*} L(h, h^*) p_{\boldsymbol{\theta}}(h^*) \tag{2.10}$$

In the following subsections, we will see how the MBR decoding principle is instantiated for different tasks.

2.4.1 Predicting the Next Event

The most common task to evaluate a point process is to predict the *time* and *type* of the single next event given an event sequence prefix. We write $p_{\text{next}}(t)$ as the probability density of the next event's time. For the commonly used L_2 loss $L(t, t^*) = (t - t^*)^2$, the MBR decode turns out to be the expected time of the next event under the model:

$$\hat{t}_{\text{MBR}} = \int_{t_{\text{last}}}^{\infty} t p_{\text{next}}(t) dt \tag{2.11}$$

where t_{last} is the time of the last event in the given prefix.

What is the MBR decode for the next event's type? For the commonly used 0-1 loss $L(k, k^*) = \mathbb{1}(k \neq k^*)$, the MBR decode would just be the most likely type. Given the next event time t_{next}, it would be

$$\hat{k}_{\text{MBR}} = \arg\max_k \frac{\lambda_k(t_{\text{next}} \mid x_{[0,t_{\text{next}})})}{\lambda(t_{\text{next}} \mid x_{[0,t_{\text{next}})})} \tag{2.12}$$

When t_{next} is not known, we would have to marginalize over it, ending up with

$$\hat{k}_{\text{MBR}} = \arg\max_k \int_{t_{\text{last}}}^{\infty} \frac{\lambda_k(t \mid x_{[0,t)})}{\lambda(t \mid x_{[0,t)})} p_{\text{next}}(t) dt \tag{2.13}$$

Now let's spell out the probability density of the next event's time:

$$p_{\text{next}}(t) = p_{\boldsymbol{\theta}}(t_{\text{next}} = t \mid x_{[0,t_{\text{next}}]}) = \lambda(t \mid x_{[0,t)}) \exp\left(-\int_{t_{\text{last}}}^{t} \lambda(s \mid x_{[0,s)}) ds\right) \quad (2.14)$$

With $p_{\text{next}}(t)$ being complicated in the general case, the integrals in equations (2.11) and (2.13) have to be estimated by Monte Carlo sampling. Suppose that we have $\{\hat{t}_m\}_{m=1}^{M}$—M IID draws of the next event's time. They were drawn by using the thinning algorithm in Algorithm 2.2. Then we will have

$$\hat{t}_{\text{MBR}} = \frac{1}{M} \sum_{m=1}^{M} \hat{t}_m \qquad \text{in lieu of equation (2.11)}$$

$$\hat{k}_{\text{MBR}} = \arg\max_k \frac{1}{M} \sum_{m=1}^{M} \frac{\lambda_k(\hat{t}_m \mid x_{[0,\hat{t}_m)})}{\lambda(\hat{t}_m \mid x_{[0,\hat{t}_m)})} \qquad \text{in lieu of equation (2.13)}$$

2.4.2 Predicting a Sequence of Future Events

Given an event sequence prefix $x_{[0,t_{\text{last}}]}$, it is also useful to predict its continuation over (t_{last}, T). In principle, the MBR decode is given by:

$$\arg\min_c \sum_{c^*} L(c, c^*) p_{\boldsymbol{\theta}}(c^* \mid x_{[0,t_{\text{last}}]}) \quad (2.16)$$

where we use c as shorthand for the continuation $x_{(t_{\text{last}},T)}$. Similar to the "next event" case, we have to make an estimate by Monte Carlo sampling:

$$\arg\min_{c \in \mathcal{C}} \sum_{c^* \in \mathcal{C}} \frac{1}{M} L(c, c^*) \quad (2.17)$$

where $\mathcal{C} = \{\hat{x}_{(t_{\text{last}},T)}\}_{m=1}^{M}$ is a collection of M IID draws of the continuation. That is, we approximate the distribution by a collection of samples, and then take the MBR decode to be the sample that has the lowest averaged loss.

A keen reader may have noticed that we made two steps of approximation—approximating $p_{\boldsymbol{\theta}}$ with IID draws (i.e., $\sum_{c^* \in \mathcal{C}}$) and only choosing the MBR decode from those draws (i.e., $\arg\min_{c \in \mathcal{C}}$). The first is inevitable, but could we do better than the second? With certain loss function L, it is possible to expand the search space beyond $\mathcal{C}$. In Chapter 7, we will introduce a specific loss function L, and show how to compose multiple samples in $\mathcal{C}$ into a single decode that has lower averaged loss than any of those samples.

2.4.3 Imputing Past Missing Events

The tasks in sections 2.4.1 and 2.4.2 both assume that the given prefix $x_{[0,t_{\text{last}}]}$ is *fully* observed. But there are many situations where the prefix is only *partially* observed. That is, we ought to impute missing events over that interval:

$$\arg\min_m \sum_{m^*} L(m, m^*) p_{\boldsymbol{\theta}}(m^* \mid \text{observed events over } [0, t_{\text{last}}]) \tag{2.18}$$

where m stands for the *missing* events over $[0, t_{\text{last}}]$. This task is much more challenging than those in sections 2.4.1 and 2.4.2, since it is intractable to sample *exactly* from $p_{\boldsymbol{\theta}}(\text{missing events} \mid \text{observed events})$.

To the best of our knowledge, the first general Monte Carlo method for this task is our particle smoothing method (Mei, Qin, and Eisner, 2019) that we will discuss in Chapter 7. So we leave all the discussion on this topic to that chapter.

2.5 Notation Summary

Now we end this chapter by summarizing our notation system, which we will follow throughout the thesis. We denote vectors by bold lowercase letters such as $\boldsymbol{\theta}$, and matrices by bold capital Roman letters such as $\mathbf{U}$ (Chapter 3). Subscripted letters

denote distinct scalars, vectors, or matrices (e.g., $\mathbf{w}_k$ in Chapter 3). Scalar quantities, including vector and matrix elements such as μ_k and $\alpha_{j,k}$, are written without bold. Capitalized scalars represent upper limits on lowercase scalars, e.g., $1 \leq k \leq K$. Function symbols are notated like their return type. All $\mathbb{R} \to \mathbb{R}$ functions are extended to apply elementwise to vectors and matrices.

The notation x for a sequence is special. Arguably, it should be boldfaced—at least when subscripted by an interval such as $[0, T)$—since it is more like a vector but not a scalar. However, that would be unnecessarily eye-catching in many equations, distracting readers from other—and often more important—elements in those equations. Therefore, we choose to not boldface it.

Appendices

2.A Discussion of the Transfer Function

As explained in section 2.1.3, when we allow *inhibition* and *inertia*, we need to pass the total activation through a non-linear **transfer function** $f : \mathbb{R} \to \mathbb{R}_+$ to obtain a positive intensity function. This was our equation (2.3).

What non-linear function f should we use? The ReLU function $f(x) = \max(x, 0)$ seems at first a natural choice. However, it returns 0 for negative x; we need to keep our intensities strictly positive at all times when an event could possibly occur, to avoid infinitely bad log-likelihood at training time or infinite log-loss at test time.

A better choice would be the "softplus" function $f(x) = \log(1 + \exp(x))$, which is strictly positive and approaches ReLU when x is far from 0. Unfortunately, "far from 0" is defined in units of x, so this choice would make our model sensitive to the units used to measure time. For example, if we switch the units of t from seconds to milliseconds, then the base intensity $f(\mu_k)$ must become 1000 times lower, forcing μ_k to be very negative and thus creating a much stronger inertial effect.

To avoid this problem, we introduce a temporal scaling parameter $s > 0$ and define $f(x) = s \log(1 + \exp(x/s))$. The scale parameter s controls the curvature of $f(x)$, which approaches ReLU as $s \to 0$, as shown in Figure 2.2. We can regard $f(x)$, x, and s as rates, with units of inverse time, so that $f(x)/s$ and x/s are unitless quantities

related by softplus. We actually learn a separate scale parameter s_k for each event type k, which will adapt to the rate of events of that type.

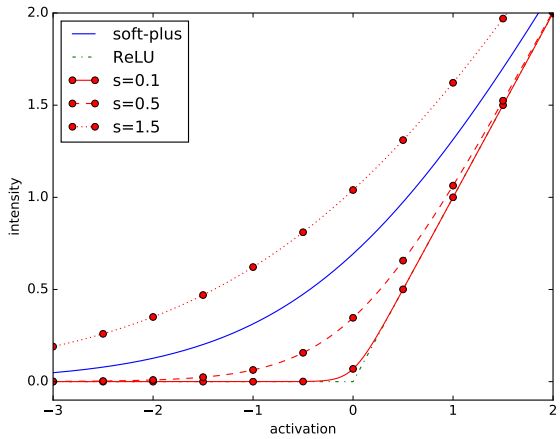


Figure 2.2: The softplus function is a soft approximation to a rectified linear unit (ReLU), approaching it as x moves away from 0. We use it to ensure a strictly positive intensity function. We incorporate a scale parameter s that controls the curvature.

2.B Likelihood Function

For the proposed models, given complete observations of an event sequence over the time interval $[0, T)$, the log-likelihood of the parameters turns out to be given by the simple formula shown in section 2.2. We start by giving the full derivation of that formula, repeated here:

$$\log p(x_{[0,T)}) = \sum_{t:x_t \neq \varnothing} \log \lambda_{x_t}(t \mid x_{[0,t)}) - \underbrace{\int_{t=0}^{T} \lambda(t \mid x_{[0,t)}) dt}_{\text{call this } \Lambda} \tag{2.6}$$

Suppose that the event sequence $x_{[0,T)}$ has I events at times $0 = t_0 < t_1 < \ldots <$

$t_I < t_{I+1} = T$. Then we can rewrite the log-likelihood as:

$$\log p(x_{[0,T)}) = \sum_{i=1}^{I} \log \lambda_{k_i}(t_i) - \int_{t=0}^{T} \lambda(t) dt$$

For presentation simplicity, we denote as k_i the type of each event x_{t_i}, and omit the history $x_{[0,t)}$ in the intensities.

First, we define $N(t) = |\{h : t_h \leq t\}|$ to be the count of events (of any type) preceding time t. So given the past history $\mathcal{H}_i \stackrel{\text{def}}{=} x_{[0,t_{i-1}]}$, the number of events in $(t_{i-1}, t]$ is denoted as $\Delta N(t_{i-1}, t) \stackrel{\text{def}}{=} N(t) - N(t_{i-1})$. Let $T_i > t_{i-1}$ be the random variable of the next event time and let K_{i+1} be the random variable of the next event type. The cumulative distribution function and probability density function of T_i (conditioned on $\mathcal{H}_i$) are given by:

$$F(t) = P(T_i \leq t) = 1 - P(T_i > t) \tag{2.19a}$$

$$= 1 - P(\Delta N(t_{i-1}, t) = 0) \tag{2.19b}$$

$$= 1 - \exp\left(-\int_{t_{i-1}}^{t} \lambda(s) ds\right) \tag{2.19c}$$

$$= 1 - \exp\left(\Lambda(t_{i-1}) - \Lambda(t)\right) \tag{2.19d}$$

$$f(t) = \exp\left(\Lambda(t_{i-1}) - \Lambda(t)\right) \lambda(t) \tag{2.19e}$$

where $\Lambda(t) = \int_0^t \lambda(s) ds$ and $\lambda(t) = \sum_{k=1}^{K} \lambda_k(t)$.

Moreover, given the past history $\mathcal{H}_i$ and the next event time t_i, the distribution of k_i is given by:

$$P(K_i = k_i \mid t_i) = \frac{\lambda_{k_i}(t_i)}{\lambda(t_i)} \tag{2.20}$$

Therefore, we can derive the likelihood function as follows:

$$\mathcal{L} = \prod_{i:t_i \leq T} \mathcal{L}_i = \prod_{t_i \leq T} \{f(t_i) P(K_i = k_i \mid t_i)\} \tag{2.21a}$$

$$= \prod_{i:t_i \leq T} \{\exp\left(\Lambda(t_{i-1}) - \Lambda(t_i)\right) \lambda_{k_i}(t_i)\} \tag{2.21b}$$

and the log-likelihood is given by

$$\log \mathcal{L} \tag{2.22a}$$

$$= \sum_{i:t_i \leq T} \log \lambda_{k_i}(t_i) - \sum_{i:t_i \leq T} \left(\Lambda(t_i) - \Lambda(t_{i-1})\right) \tag{2.22b}$$

$$= \sum_{i:t_i \leq T} \log \lambda_{k_i}(t_i) - \Lambda(T) \tag{2.22c}$$

$$= \sum_{i:t_i \leq T} \log \lambda_{k_i}(t_i) - \int_{t=0}^{T} \lambda(t) dt \tag{2.22d}$$

Chapter 3

A Purely Neural Model: Neural Hawkes Process

In the last chapter, we introduced the formalism for building probabilistic models of event sequences, and presented a few examples. In this chapter, we formally introduce the neural Hawkes process—our novel probabilistic model that can capture effects that the previous models all miss. Our model allows past events to influence the future in complex and realistic ways, by conditioning future event intensities on the hidden state of a novel continuous-time LSTM that has consumed the stream of past events. Our model has desirable qualitative properties. It achieves superior likelihood and predictive accuracy on real and synthetic datasets, including under missing-data conditions.

3.1 The Model

Recall from section 2.1 that building a multivariate point process is to design its intensity functions λ_k. We use a recurrent neural network, which allows learning a complex dependence of the intensities on the number, order, and timing of past events. We refer to our model as the **neurally self-modulating multivariate point process**

(N-SM-MPP), or **neural Hawkes process (NHP)** for short.[1]

3.1.1 Conditioning Intensities on a Continuous-Time LSTM

Just as introduced in section 2.1, each event type k has an time-varying intensity $\lambda_k(t)$, which jumps discontinuously at each new event, and then drifts continuously toward a baseline intensity. In the new process, however, these dynamics are controlled by a hidden state vector $\mathbf{h}(t) \in (-1, 1)^D$, which in turn depends on a vector $\mathbf{c}(t) \in \mathbb{R}^D$ of memory cells in a **continuous-time LSTM**. This novel recurrent neural network architecture is inspired by the familiar discrete-time LSTM (Hochreiter and Schmidhuber, 1997; Graves, 2012). The difference is that in the continuous interval following an event, each memory cell c *exponentially decays* at some rate δ toward some steady-state value $\bar{c}$.

Suppose that the event sequence $x_{[0,T)}$ has I events at times $0 = t_0 < t_1 < \ldots < t_I < t_{I+1} = T$. We denote as k_i the type of each event x_{t_i}. At each time $t > 0$, we obtain the intensity $\lambda_k(t)$ by (3.1a), where (3.1b) defines how the hidden states $\mathbf{h}(t)$ are continually obtained from the memory cells $\mathbf{c}(t)$ as the cells decay:

$$\lambda_k(t) = f_k(\mathbf{w}_k^\top \mathbf{h}(t)) \tag{3.1a}$$

$$\mathbf{h}(t) = \mathbf{o}_i \odot (2\sigma(2\mathbf{c}(t)) - 1) \text{ for } t \in (t_{i-1}, t_i] \tag{3.1b}$$

This says that on the interval $(t_{i-1}, t_i]$—in other words, after event $i-1$ up until event i occurs at some time t_i—the $\mathbf{h}(t)$ defined by equation (3.1b) determines the intensity functions via equation (3.1a). So for t in this interval, according to the model, $\mathbf{h}(t)$ is a sufficient statistic of the history $x_{[0,T)}$ with respect to future events.

[1] Our model is significantly more expressive than Hawkes process; we'd still like to keep "Hawkes" in our model name since we were inspired by that model.

$\mathbf{h}(t)$ is analogous to $\mathbf{h}_i$ in an LSTM language model (Mikolov et al., 2010), which summarizes the past event sequence $x_1, \ldots, x_{i-1}$. But in our decay architecture, it will also reflect the interarrival times $t_1 - 0, t_2 - t_1, \ldots, t_{i-1} - t_{i-2}, t - t_{i-1}$.

This interval $(t_{i-1}, t_i]$ ends when the next event, which has type k_i, stochastically occurs at some time t_i. At this point, the continuous-time LSTM reads (t_i, k_i) and updates the current (decayed) hidden cells $\mathbf{c}(t)$ to new initial values $\mathbf{c}_{i+1}$, based on the current (decayed) hidden state $\mathbf{h}(t_i)$.

3.1.2 Updates of the Continuous-Time LSTM

How does the continuous-time LSTM make those updates? Other than depending on decayed values $\mathbf{h}(t_i)$, the update formulas resemble the discrete-time case:[2]

$$\mathbf{i}_{i+1} \leftarrow \sigma\left(\mathbf{W}_{\mathrm{i}}\mathbf{k}_i + \mathbf{U}_{\mathrm{i}}\mathbf{h}(t_i) + \mathbf{d}_{\mathrm{i}}\right) \tag{3.2a}$$

$$\mathbf{f}_{i+1} \leftarrow \sigma\left(\mathbf{W}_{\mathrm{f}}\mathbf{k}_i + \mathbf{U}_{\mathrm{f}}\mathbf{h}(t_i) + \mathbf{d}_{\mathrm{f}}\right) \tag{3.2b}$$

$$\mathbf{z}_{i+1} \leftarrow 2\sigma\left(\mathbf{W}_{\mathrm{z}}\mathbf{k}_i + \mathbf{U}_{\mathrm{z}}\mathbf{h}(t_i) + \mathbf{d}_{\mathrm{z}}\right) - 1 \tag{3.2c}$$

$$\mathbf{o}_{i+1} \leftarrow \sigma\left(\mathbf{W}_{\mathrm{o}}\mathbf{k}_i + \mathbf{U}_{\mathrm{o}}\mathbf{h}(t_i) + \mathbf{d}_{\mathrm{o}}\right) \tag{3.2d}$$

$$\mathbf{c}_{i+1} \leftarrow \mathbf{f}_{i+1} \odot \mathbf{c}(t_i) + \mathbf{i}_{i+1} \odot \mathbf{z}_{i+1} \tag{3.2e}$$

$$\bar{\mathbf{c}}_{i+1} \leftarrow \bar{\mathbf{f}}_{i+1} \odot \bar{\mathbf{c}}_i + \bar{\mathbf{i}}_{i+1} \odot \mathbf{z}_{i+1} \tag{3.2f}$$

$$\boldsymbol{\delta}_{i+1} \leftarrow f\left(\mathbf{W}_{\mathrm{d}}\mathbf{k}_i + \mathbf{U}_{\mathrm{d}}\mathbf{h}(t_i) + \mathbf{d}_{\mathrm{d}}\right) \tag{3.2g}$$

[2]The upright-font subscripts i, f, z and o are not variables, but constant labels that distinguish different $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{d}$ tensors. The $\bar{\mathbf{f}}$ and $\bar{\mathbf{i}}$ in equation (3.2f) are defined analogously to $\mathbf{f}$ and $\mathbf{i}$ but with different weights.

The vector $\mathbf{k}_i \in \{0,1\}^K$ is the i^{th} input: a one-hot encoding of the new event's type k_i, with non-zero value only at the entry indexed by k_i. The above formulas will make a discrete update to the LSTM state. They resemble the discrete-time LSTM, but there are two differences. First, the updates do not depend on the "previous" hidden state from just after time t_{i-1}, but rather its value $\mathbf{h}(t_i)$ at time t_i, after it has decayed for a period of $t_i - t_{i-1}$. Second, equations (3.2f)–(3.2g) are new. They define how in future, as $t > t_i$ increases, the elements of $\mathbf{c}(t)$ will continue to deterministically decay (at different rates) from $\mathbf{c}_{i+1}$ toward targets $\bar{\mathbf{c}}_{i+1}$. Specifically, $\mathbf{c}(t)$ is given by (3.3), which continues to control $\mathbf{h}(t)$ and thus $\lambda_k(t)$ (via (3.1), except that i has now increased by 1).

$$\mathbf{c}(t) \stackrel{\text{def}}{=} \bar{\mathbf{c}}_{i+1} + (\mathbf{c}_{i+1} - \bar{\mathbf{c}}_{i+1}) \exp\left(-\boldsymbol{\delta}_{i+1}\,(t - t_i)\right) \text{ for } t \in (t_i, t_{i+1}] \tag{3.3}$$

In short, not only does (3.2e) define the usual cell values $\mathbf{c}_{i+1}$, but equation (3.3) defines $\mathbf{c}(t)$ on $\mathbb{R}_{>0}$. On the interval $(t_i, t_{i+1}]$, $\mathbf{c}(t)$ follows an exponential curve that begins at $\mathbf{c}_{i+1}$ (in the sense that $\lim_{t \to t_i^+} \mathbf{c}(t) = \mathbf{c}_{i+1}$) and decays toward $\bar{\mathbf{c}}_{i+1}$ (which it would approach as $t \to \infty$, if extrapolated).

3.1.3 Qualitative Properties

A schematic example is shown in Figure 3.1. This is the same figure as Figure 1.4; we display it here again for convenient reference. As in the Hawkes and generalized Hawkes processes (see section 2.1), $\lambda_k(t)$ drifts deterministically between events toward some base rate. But the neural version is different in three ways: ① The base rate is not a constant μ_k, but shifts upon each event.[3] ② The drift can be non-monotonic, because the excitatory and inhibitory influences on $\lambda_k(t)$ from different

[3] Equations (3.1b) and (3.3) imply that after event $i-1$, the base rate jumps to $f_k(\mathbf{w}^\top(\mathbf{o}_i \odot (2\sigma(2\bar{\mathbf{c}}_i) - 1)))$.

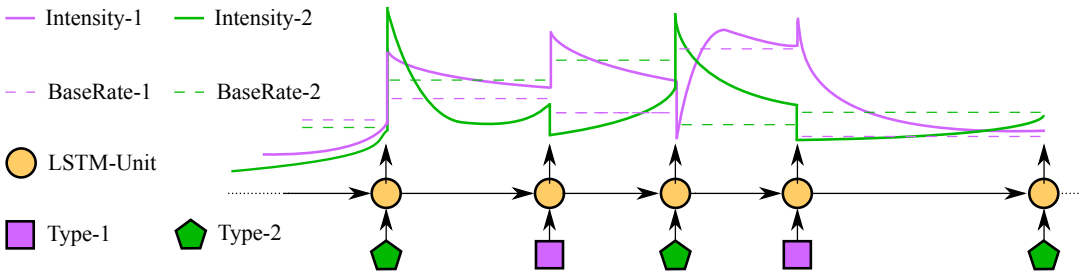


Figure 3.1: Drawing an event stream from a neural Hawkes process. An LSTM reads the sequence of past events (polygons) to arrive at a hidden state (orange). That state determines the future "intensities" of the two types of events—that is, their time-varying instantaneous probabilities. The intensity functions are continuous parametric curves (solid lines) determined by the most recent LSTM state, with dashed lines showing the steady-state asymptotes that they would eventually approach. In this example, events of type 1 excite type 1 but inhibit type 2. Type 2 excites itself, and excites or inhibits type 1 according to whether the count of type 2 events so far is odd or even. Those are immediate effects, shown by the sudden jumps in intensity. The events also have longer-timescale effects, shown by the shifts in the asymptotic dashed lines.

elements of $\mathbf{h}(t)$ may decay at different rates. ③ The sigmoidal transfer function means that the behavior of $\mathbf{h}(t)$ itself is a little more interesting than exponential decay. Suppose that $\mathbf{c}_i$ is very negative but increases toward a target $\bar{\mathbf{c}}_i > 0$. Then $\mathbf{h}(t)$ will stay close to -1 for a while and then will rapidly rise past 0. This usefully lets us model a delayed response (e.g. the last green segment in Figure 3.1).

We point out two behaviors that are naturally captured by our LSTM's "forget" and "input" gates:

- if $\mathbf{f}_{i+1} \approx \mathbf{1}$ and $\mathbf{i}_{i+1} \approx \mathbf{0}$, then $\mathbf{c}_{i+1} \approx \mathbf{c}(t_i)$. So $\mathbf{c}(t)$ and $\mathbf{h}(t)$ will be *continuous* at t_i. There is no jump due to event i, though the steady-state target may change.
- if $\bar{\mathbf{f}}_{i+1} \approx \mathbf{1}$ and $\bar{\mathbf{i}}_{i+1} \approx \mathbf{0}$, then $\bar{\mathbf{c}}_{i+1} \approx \bar{\mathbf{c}}_i$. So although there may be a jump in activation, it is temporary. The memory cells will decay toward the same steady states as before.

Among other benefits, this lets us fit datasets in which (as is common) some pairs of event types do *not* influence one another. Section 3.A.2 explains why both Hawkes and our model have this ability.

The drift of $\mathbf{c}(t)$ between events controls how the system's expectations about future events change as more time elapses with no event having yet occured. Equation (3.3) chooses a moderately flexible parametric form for this drift function (see section 3.C for some alternatives). Equation (3.2e) was designed so that $\mathbf{c}$ in an LSTM could learn to count past events with discrete-time exponential discounting; and (3.3) can be viewed as extending that to continuous-time exponential discounting.

Our memory cell vector $\mathbf{c}(t)$ is a *deterministic* function of the past history $x_{[0,t)}$.[4] Thus, the event intensities at any time are also deterministic via equation (3.1). The stochastic part of the model is the random choice—based on these intensities—of *which* event happens next and *when* it happens. The events are in competition: an event with high intensity is likely to happen sooner than an event with low intensity, and whichever one happens first is fed back into the LSTM. If no event type has high intensity, it may take a long time for the next event to occur.

3.1.4 Training and Inference

We can train the model and do inference with it by following the general recipes in sections 2.2 and 2.4. Training the model means learning the LSTM parameters in equation (3.2) along with the other parameters mentioned in this section, namely $s_k \in \mathbb{R}$ in f_k and $\mathbf{w}_k \in \mathbb{R}^D$ for $k \in \{1, 2, \ldots, K\}$.

To sample predictions of future events by the thinning algorithm, we need to construct the upper bound λ^* on λ_k after each event (t_i, k_i). Note that, just like the

[4] Section 3.A.1 explains how our LSTM handles the start and end of the sequence.

generalized Hawkes process (equation (2.3)), our neural model (equation (3.1)) also has the form $f_k(g_0(t) + g_1(t) + \cdots + g_n(t))$ where f_k is monotonically non-decreasing and each g is a *bounded* function on (t_i, ∞). We can therefore replace each g function by its upper bound to obtain $\lambda^* = f_k(\max g_0(t) + \max_t g_1(t) + \cdots + \max_t g_n(t))$. Specifically, each summand $w_{kd} h_d(t) = w_{kd} \cdot o_{id} \cdot (2\sigma(2c_d(t)) - 1)$ is upper-bounded by $\max_{c \in \{c_{id}, \bar{c}_{id}\}} w_{kd} \cdot o_{id} \cdot (2\sigma(2c) - 1)$.

3.2 Related Work

The Hawkes process has been widely used to model event streams, including for topic modeling and clustering of text document streams (He et al., 2015; Du et al., 2015a), constructing and inferring network structure (Yang and Zha, 2013; Choi et al., 2015; Etesami et al., 2016), personalized recommendations based on users' temporal behavior (Du et al., 2015b), discovering patterns in social interaction (Guo et al., 2015; Lukasik et al., 2016), learning causality (Xu, Farajtabar, and Zha, 2016), and so on.

By the time we built the neural Hawkes process, there had been research in expanding the expressivity of Hawkes processes. Zhou, Zha, and Song (2013) describe a self-exciting process that removes the assumption of exponentially decaying influence (as we do). They replace the scaled-exponential summands in equation (2.2) with learned positive functions of time (the choice of function again depends on k_i, k). Lee, Lim, and Ong (2016) generalize the constant excitation parameters $\alpha_{j,k}$ to be stochastic, which increases expressivity. Our model also allows non-constant interactions between event types, but arranges these via deterministic, instead of stochastic, functions of continuous-time LSTM hidden states. Wang et al. (2016) consider non-linear effects of past history on the future, by passing the intensity functions of the Hawkes process through a non-parametric isotonic link function g, which is in the same place

as our non-linear function f_k. In contrast, our f_k has a fixed parametric form (learning only the scale parameter), and is approximately linear when x is large. This is because we model non-linearity (and other complications) with a continuous-time LSTM, and use f_k only to ensure positivity of the intensity functions.

Du et al. (2016) independently combined Hawkes processes with recurrent neural networks (and Xiao et al., 2017a propose an advanced way of estimating the parameters of that model). However, Du et al.'s architecture is different in several respects. They use standard discrete-time LSTMs without our decay innovation, so they must encode the intervals between past events as explicit numerical inputs to the LSTM. They have only a single intensity function $\lambda(t)$, and it simply decays exponentially toward 0 between events, whereas our more modular model creates separate (potentially transferrable) functions $\lambda_k(t)$, each of which allows complex and non-monotonic dynamics en route to a non-zero steady state intensity. Some structural limitations of their design are that t_i and k_i are conditionally independent given $\mathbf{h}$ (they are determined by separate distributions), and that their model cannot avoid a positive probability of extinction at all times. Finally, since they take $f = \exp$, the effect of their hidden units on intensity is effectively multiplicative, whereas we take $f =$ softplus to get an approximately additive effect inspired by the classical Hawkes process. Our rationale is that additivity is useful to capture independent (disjunctive) causes; at the same time, the hidden units that our model adds up can each capture a complex joint (conjunctive) cause.

We expect the exponential drift in equation (3.3) to be expressive enough in most settings. However, one might want to capture fancier patterns, e.g., periodic fluctuation of the intensity between events; see section 3.C for more discussion about fancier

drifts. A more expressive way is to directly model the time derivative of the state $\mathrm{d}\mathbf{h}(t)/\mathrm{d}t$ by a neural network, just as in Rubanova, Chen, and Duvenaud (2019); training such a model requires back-propagation through an ODE solver, a technique that was proposed in Chen et al. (2018). Unlike our neural Hawkes process whose intensities are all bounded, the neural ODE design allows the intensities to increase without bound; this freedom may be desirable in some applications but unreasonable in others. Also due to the unbounded intensities, the thinning algorithm (section 2.3) is not applicable to neural ODE models; Chen, Amos, and Nickel (2021) proposed an inversion-sampling approach to draw events from such models.

3.3 Experiments and Analysis

We fit our various models on several simulated and real-world datasets, and evaluated them in each case by the *log-probability* that they assigned to held-out data. We also compared our approach with that of Du et al. (2016) on their *prediction* task. The datasets that we use in this chapter range from one extreme with only $K = 2$ event types but mean sequence length > 2000, to the other extreme with $K = 5000$ event types but mean sequence length 3. Dataset details can be found in Table 3.2 in section 3.B.1. Training details (e.g., hyperparameter selection) can be found in section 3.B.2. Our code and data are available at `https://github.com/HMEIatJHU/neurawkes`.

3.3.1 Synthetic Datasets

Our hope is that the neural Hawkes process is a flexible tool that can be used to fit naturally occurring data. We first checked that we could successfully fit data generated from *known* distributions. That is, when the generating distribution actually fell

within our model family, could our training procedure recover the distribution in practice? When the data came from a Hawkes or generalized Hawkes process, could we nonetheless train our neural process to fit the distribution well?

We used the thinning algorithm (section 2.3) to sample event sequences from different processes with randomly generated parameters:

- (a) a standard Hawkes process (SE-MPP, see section 2.1.2);
- (b) our generalized Hawkes process (D-SM-MPP, see section 2.1.3);
- (c) our neural Hawkes process (N-SM-MPP, see section 3.1).

We then tried to fit each dataset with all these models.

The results are shown in Figure 3.2, including log-likelihood (reported in nats per event) on the sequences and the breakdown of time interval and event types. We found that all models were able to fit the (a) and (b) datasets well with no statistically significant difference among them, but that the (c) models were substantially and significantly better at fitting the (c) datasets. In all cases, the (c) models were able to obtain a low KL divergence from the true generating model (the difference from the oracle column). This result suggests that the neural Hawkes process may be a wise choice: it introduces extra expressive power that is sometimes necessary and does not appear (at least in these experiments) to be harmful when it is not necessary.

In this experiment, we were not limited to measuring the likelihood of the models on the stochastic event sequences. We also knew the true latent intensities of the generating process, so we were able to directly measure whether the trained models predicted these intensities accurately. The pattern of results was similar.

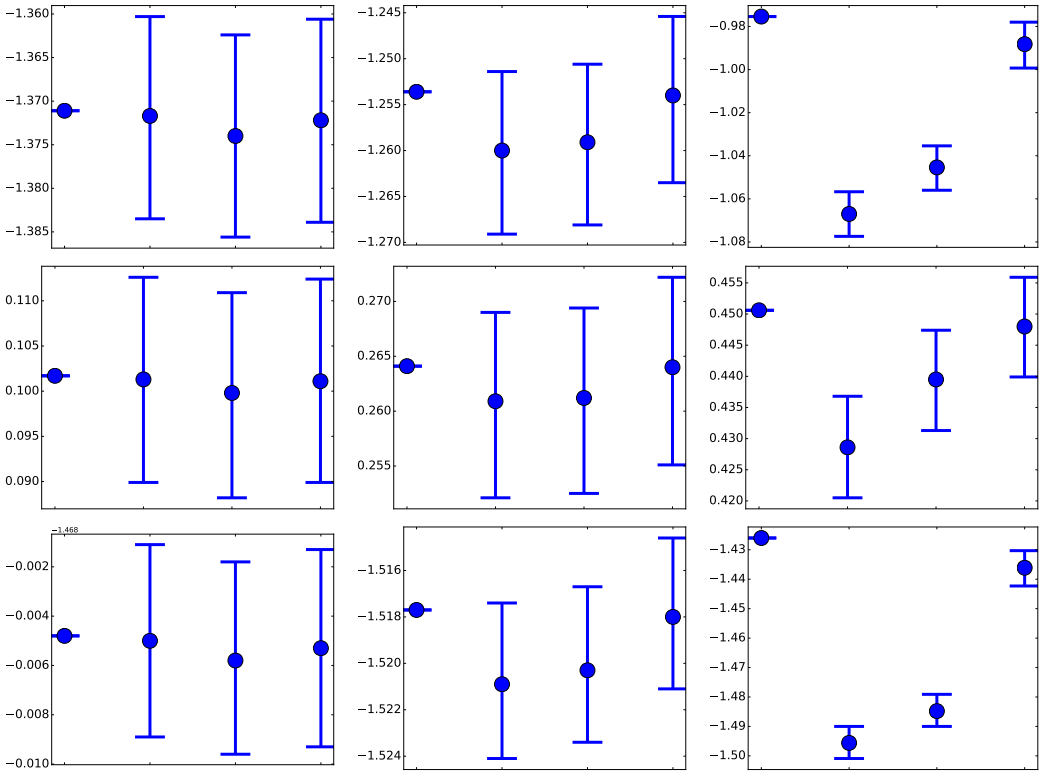
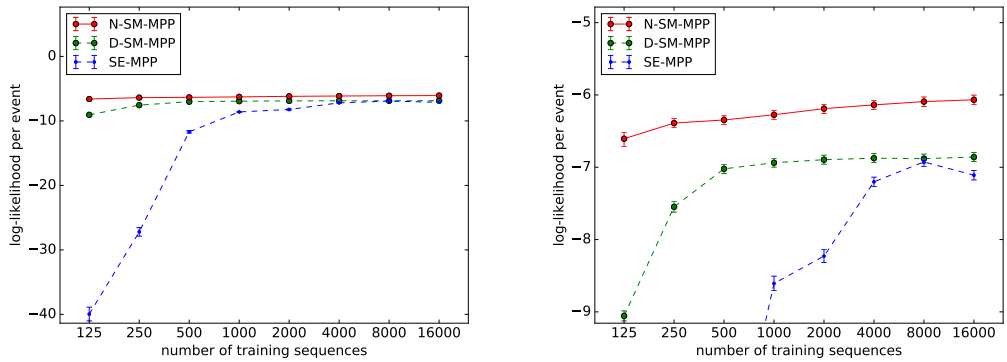


Figure 3.2: Log-likelihood (reported in nats per event) of each model on held-out synthetic data. Rows (top-down) are log-likelihood on the entire sequence, time interval, and event type. On each row, the figures (from left to right) are datasets generated by the Hawkes process, generalized Hawkes process, and neural Hawkes process. In each figure, the models (from left to right) are Oracle, Hawkes process, generalized Hawkes process, and neural Hawkes process. Larger values are better. Note that log-likelihood for continuous variables can be positive, since it uses the log of a probability density that may be > 1.

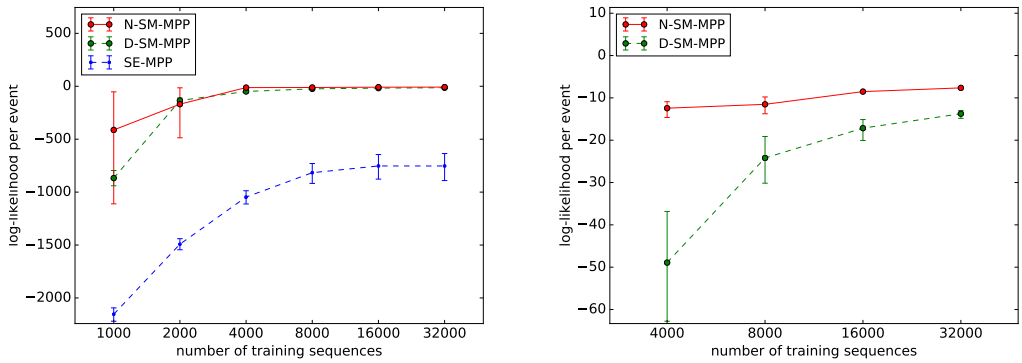
3.3.2 Real-World Media Datasets

Retweets Dataset (Zhao et al., 2015)**.** On Twitter, *novel* tweets are generated from some distribution, which we do not model here. Each novel tweet serves as the beginning-of-sequence event (see section 3.A.1) for a subsequent sequence of *retweet* events. We model the dynamics of these sequences: how retweets by various types of users ($K = 3$) predict later retweets by various types of users. The dataset is interesting for its temporal pattern. People like to retweet an interesting post soon after it is created and retweeted by others, but may gradually lose interest, so the intervals between retweets become longer over time. In other words, the stream begins in a *self-exciting state*, in which previous retweets increase the intensities of future retweets, but eventually interest dies down and events are less able to excite one another. The Hawke and generalized Hawkes processes are essentially incapable of modeling such a phase transition, but our neural model should have the capacity to do so.

We generated learning curves (Figure 3.3a) by training our models on increasingly long prefixes of the training set. As we can see, both neural Hawkes process and generalized Hawkes process *significantly* outperform the Hawkes process at *all* training sizes. There is no obvious *a priori* reason to expect inhibition or even inertia in this application domain, which explains why the generalized Hawkes process makes only a small improvement over the Hawkes process when the latter is well-trained. But the generalized Hawkes process requires much less data, and also has more stable behavior (smaller error bars) on small datasets. Our neural model is even better. Not only does it do better on the average stream, but its *consistent* superiority over the other two models is shown by the per-sequence scatterplots in Figure 3.4, demonstrating



(a) Learning curve (with 95% error bars) of all three models on the Retweets dataset. The left graph is the right graph zoomed out. Our neural model (N-SM-MPP) significantly outperforms our generalized Hawkes process (D-SM-MPP), and both significantly outperform the Hawkes process (SE-MPP).



(b) Learning curves (with 95% error bars) of all three models on the MemeTrack dataset. Similar to Figure 3.3a, our neural model (N-SM-MPP) significantly outperforms our generalized Hawkes process (D-SM-MPP), and both significantly outperform the Hawkes process (SE-MPP).

Figure 3.3: Learning curves (with 95% error bars) of all three models on the Retweets and MemeTrack datasets.

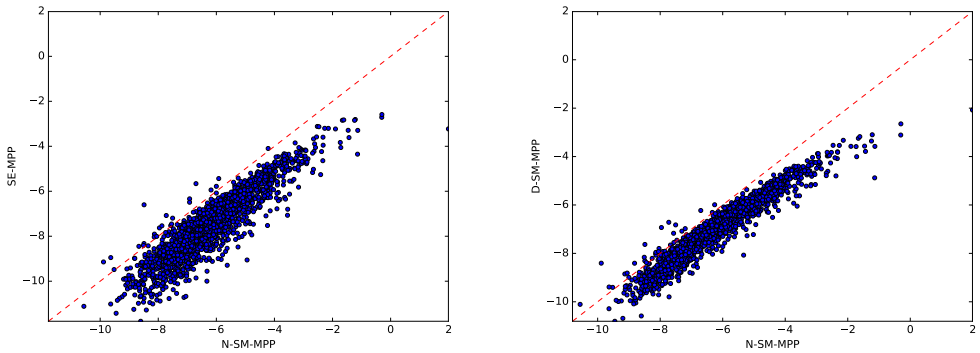


Figure 3.4: Scatterplots on the Retweets dataset, comparing the held-out log-likelihood of the models (when trained on our full Retweets training set) with respect to *each* of the 2000 test sequences. The left graph is neural Hawkes (N-SM-MPP) vs. Hawkes (SE-MPP); the right graph is neural Hawkes (N-SM-MPP) vs. generalized Hawkes (D-SM-MPP). Nearly all points fall to the right of $y = x$, since the neural Hawkes process (N-SM-MPP) is consistently more predictive than the other two models.

the importance of our model's neural component even with large datasets.

MemeTrack Dataset (Leskovec and Krevl, 2014)**.** This dataset is similar in conception to Retweets, but with many more event types ($K = 5000$). It considers the reuse of fixed phrases, or "memes," in online media. It contains time-stamped instances of meme use in articles and posts from 1.5 million different blogs and news sites. We model how the future occurrence of a meme is affected by its past trajectory across different websites—that is, given one meme's past trajectory across websites, when and where it will be mentioned again.

On this dataset, the advantage of our full neural models was dramatic, yielding cross-entropy per event of around -8 relative to the -15 of the generalized Hawkes process—which in turn is *far* above the -800 of the Hawkes process. Figure 3.3b illustrates the persistent gaps among the models. A scatterplot similar to Figure 3.4 is given in Figure 3.10. We attribute the poor performance of the Hawkes process to

its failure to capture the latent properties of memes, such as their interestingness or political stance. This is a form of missing data, as we now discuss.

Missing Interestingness. Most memes in MemeTrack are uninteresting and give rise to only a short sequence of mentions. Thus the base mention probability is low. An ideal analysis would recognize that if a specific meme has been mentioned several times already, it is *a posteriori* interesting and will probably be mentioned in future as well. The Hawkes process cannot distinguish the interesting memes from the others, except insofar as they appear on more influential websites. By contrast, our generalized Hawkes can partly capture this inferential pattern by using *negative* base rates $\boldsymbol{\mu}$ to create "inertia" (section 2.1.3). Indeed, all 5000 of its learned μ_k parameters were negative, with values ranging from -10 to -30, which numerically yields 0 intensity and is hard to excite.

Missing Political Stance. An ideal analysis would also recognize that if a specific meme has appeared mainly on conservative websites, it is *a posteriori* conservative and unlikely to appear on liberal websites in the future. The generalized Hawkes, unlike the Hawkes process, can again partly capture this, by having conservative websites *inhibit* liberal ones. Indeed, 24% of its learned α parameters were negative. (We re-emphasize that this inhibition is merely a predictive effect—probably not a direct causal mechanism.)

How does our neural model handle such missingness? Our neural model is even more powerful than the generalized Hawkes process. The LSTM state aims to learn sufficient statistics for predicting the future, so it can learn hidden dimensions (which fall in $(-1, 1)$) that encode useful posterior beliefs in boolean properties of the meme such as interestingness, conservativeness, timeliness, etc. The LSTM's "long

short-term memory" architecture explicitly allows these beliefs to persist indefinitely through time in the absence of new evidence, without having to be refreshed by redundant new events as in the decomposable models. Also, the LSTM's hidden dimensions are computed by sigmoidal activation rather than softplus activation, and so can be used implicitly to perform logistic regression. The flat left side of the sigmoid resembles softplus and can model *inertia* as we saw above: it takes several mentions to establish interestingness. Symmetrically, the flat right side can model *saturation*: once the posterior probability of interestingness is at 80%, it cannot climb much farther no matter how many more mentions are observed.

A final potential advantage of the LSTM is that in this large-K setting, it has fewer parameters than the other models, sharing statistical strength across event types (websites) to generalize better. The learning curves in Figure 3.3b suggest that on small data, the other models may overfit their $O(K^2)$ interaction parameters $\alpha_{j,k}$. Our neural model only has to learn $O(D^2)$ pairwise interactions among its D hidden nodes (where $D \ll K$), as well as $O(KD)$ interactions between the hidden nodes and the K event types. In this case, $K = 5000$ but $D = 64$. This reduction by using latent hidden nodes is analogous to nonlinear latent factor analysis.

3.3.3 Sensitivity to Number of Parameters

Does our method do well because of its flexible nonlinearities or just because it has more parameters? The answer is both. We experimented on the Retweets data with reducing the number of hidden units D. As shown in Table 3.1, our neural model substantially outperformed the Hawkes process on held-out data even with very few parameters, although more parameters does even better.

More information about model sizes is given in section 3.B.3. Note that the neural

	HAWKES	NEURAL HAWKES WITH DIFFERENT D						
		1	2	4	8	16	32	256
# OF PARAMETERS	21	31	87	283	1011	3811	14787	921091
LOG-LIKELIHOOD	-7.19	-6.51	-6.41	-6.36	-6.24	-6.18	-6.16	-6.10

Table 3.1: Sensitivity to number of parameters on the Retweets dataset.

Hawkes process does not *always* have more parameters. When K is large, we can greatly reduce the number of parameters below that of a Hawkes process, by choosing $D \ll K$, as for the MemeTrack dataset.

3.3.4 Modeling Sequences With Missing Data

Our neural Hawkes process is able to cope with missing data. Even in a domain where Hawkes might be appropriate, it is hard to apply Hawkes when sequences are only partially observed. Real datasets may *systematically* omit some types of events (e.g., illegal drug use, or offline purchases) which, in the true generative model, would have a strong influence on the future. They may also have *stochastically* missing data, where the missingness mechanism—the probability that an event is not recorded—can be complex and data-dependent (MNAR). In this setting, we can fit our model directly to the observation sequences, and use it to predict observation sequences that were generated in the same way (using the same complete-data distribution and the same missingness mechanism). Note that if one knew the true complete-data distribution—perhaps Hawkes—and the true missingness mechanism, one would optimally predict the incomplete future from the incomplete past in Bayesian fashion, by integrating over possible completions (imputing the missing events and considering their influence on the future). Our hope is that the neural model is expressive enough that it can learn to approximate this true predictive distribution. Its hidden state after

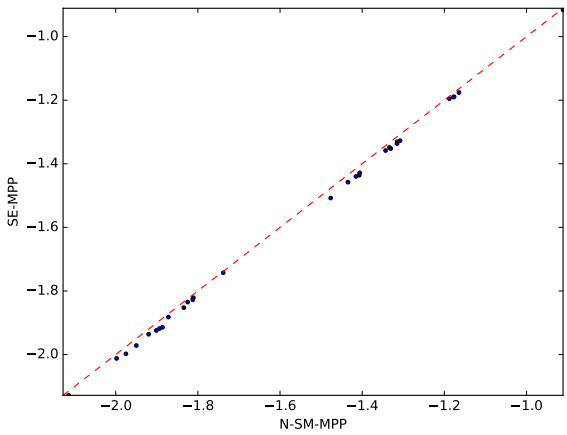


Figure 3.5: Scatterplot of neural Hawkes (N-SM-MPP) vs. Hawkes (SE-MPP), comparing their log-likelihoods with respect to *each* of the 31 test sets of incomplete sequences. All 31 points fall to the right of $y = x$.

observing the past should implicitly encode the Bayesian posterior, and its update rule for this hidden state should emulate the "observable operator" that updates the posterior upon each new observation.

Experimental Results. We set up an artificial experiment to more directly investigate the missing-data setting, where we do not observe *all* events during $[0, T)$, but train and test our model just as if we had. We sampled synthetic event sequences from an ordinary Hawkes process removed all the events of selected types, and then compared the neural Hawkes process with the Hawkes process as models of these censored sequences. Since we took $K = 5$, there were $2^5 - 1 = 31$ ways to construct a dataset of censored sequences. As shown in Figure 3.5, for *each* of the 31 resulting datasets, training a neural Hawkes model achieves better generalization.

Analysis. Now we discuss why the kind of behavior in Figure 3.5 is to be expected. Supppose the true complete-data distribution p^* is itself an unknown neural

Hawkes process. Then a sufficient statistic for prediction from the incompletely observed past would be the posterior distribution over the true hidden neural state $\mathbf{t}$ of the unknown process, which was reached by reading the *complete* past. We would ideally obtain our predictions by correctly modeling the missing observations and integrating over them. However, inference would be computationally quite expensive even if p^* were known, to say nothing of the case where p^* is unknown and we must integrate over its parameters as well.[5]

We instead train a neural model that attempts to bypass these problems. The hope is that our model's hidden state, after it reads only the observed *incomplete* past, will be nearly as predictive as the posterior distribution above. In our missing-data experiments, the true complete-data distribution p^* happened to be a classical Hawkes process, but we censored some event types. We then modeled the observed incomplete sequence as if it were a complete sequence. In this setting, a Hawkes process will in general be unable to fit the data well, which is why the neural Hawkes process has an advantage in all 31 experiments.

What goes wrong with using the Hawkes model? Suppose that in the true Hawkes model p^*, type 1 is rare but strongly excites type 2 and type 3, which do not excite themselves or each other. Type 1 events are missing in the observed sequence. What is the correct predictive distribution in this situation (with knowledge of p^*)? Seeing lots of type 2 events in a row suggests that they were preceded by a (single) missing type 1 event, which predicts a higher intensity for type 3 in future. The more type 2 events we see, the surer we are that there was a type 1 event, but we doubt that there were multiple type 1 events, so the predicted intensity of type 3 is expected to

[5]Chapter 7 presents our novel method for efficiently imputing missing events with a known p^*.

increase sublinearly as $P(\text{type} = 1)$ approaches 1. As neural networks are universal function approximators, a neural Hawkes model may be able to recognize and fit this sublinear behavior in the incomplete training data. However, if we fit only a Hawkes model to the incomplete training data, it would have to posit that type 2 excites type 3 directly, so the predicted intensity of type 3 would incorrectly increase linearly with the number of type 2 events.

3.3.5 Prediction Tasks—Medical, Social and Financial

To compare with Du et al. (2016), we evaluate our model on the *prediction* tasks and datasets that they proposed. The Financial Transaction dataset contains long sequences of high frequency stock transactions for a single stock, with the two event types "buy" and "sell." The electrical medical records (MIMIC-II) dataset is a collection of de-identified clinical visit records of Intensive Care Unit patients for 7 years. Each patient has a sequence of hospital visit events, and each event records its time stamp and disease diagnosis. The Stack Overflow dataset represents two years of user awards on a question-answering website: each user received a sequence of badges (of 22 different types).

We follow Du et al. (2016) and attempt to predict every held-out event (t_i, k_i) from the sequence prefix $x_{[0,t_{i-1}]}$ over the interval $[0, t_{i-1}]$. We evaluate the prediction $\hat{k}_i$ with 0-1 loss—yielding an error rate, or ER—and evaluate the prediction $\hat{t}_i$ with L_2 loss—yielding a root-mean-squared error, or RMSE. We make minimum Bayes risk predictions as explained in section 2.4. Figure 3.6 shows that our model consistently outperforms that of Du et al. (2016) on event type prediction on all the datasets, although for time prediction neither model is consistently better.

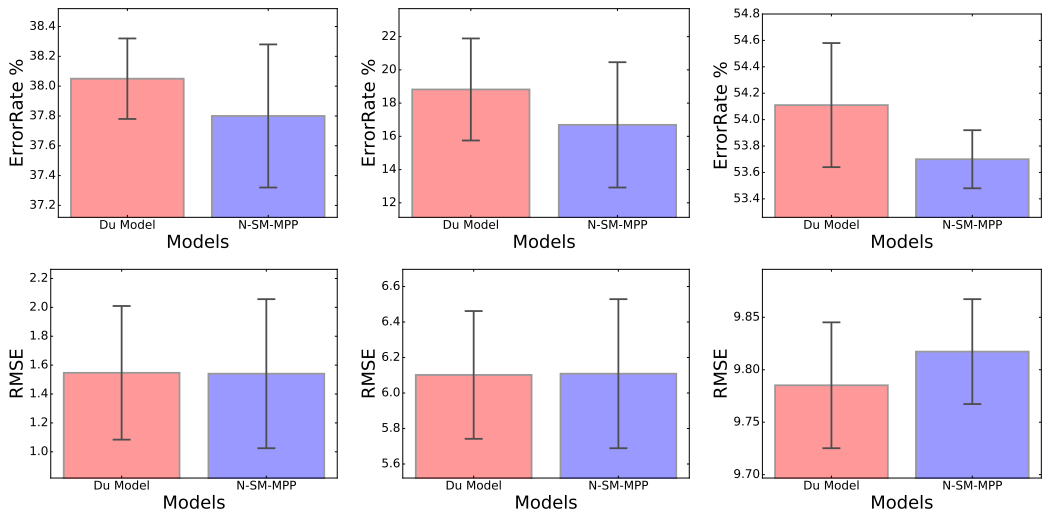


Figure 3.6: Prediction results on Financial Transactions, MIMIC-II, and Stack Overflow datasets (from left to right). Error bars show standard deviation over 5 experiments with different train-dev-test splits. For prediction of the types k_i (top row), our method (N-SM-MPP) achieved lower error in 4/5, 5/5, and 5/5 of the experiments. For prediction of the times t_i (bottom row), our method achieved lower error in 5/5, 2/5, and 0/5 of the experiments.

3.4 Conclusion

In this chapter, we presented our neural Hawkes process, an extension to the multivariate Hawkes process, a popular generative model of sequences of typed, time-stamped events. Past events may now either excite *or* inhibit future events. They do so by *sequentially* updating the state of a novel *continuous-time* recurrent neural network (LSTM). Whereas Hawkes sums the time-decaying influences of past events, we instead sum the time-decaying influences of the LSTM nodes. Our novel model aims to address real-world phenomena. Empirically, we have shown that it yields a significantly improved ability to predict the course of future events. There are several exciting avenues for further improvements (discussed in section 3.C), including embedding our model within a reinforcement learner to discover causal structure and

learn an intervention policy.

Appendices

3.A Model Details

In this appendix, we discuss some qualitative properties of our models and give details about how we handle boundary conditions.

3.A.1 Boundary Conditions for the LSTM

We initialize the continuous-time LSTM's hidden state to $\mathbf{h}(0) = \mathbf{0}$, and then have it read a special beginning-of-sequence (BOS) event (k_0, t_0), where k_0 is a special event type (i.e., expanding the LSTM's input dimensionality by one) and t_0 is set to be 0. Then equation (3.2) defines $\mathbf{c}_1$ (from $\mathbf{c}_0 \stackrel{\text{def}}{=} \mathbf{0}$), $\bar{\mathbf{c}}_1$, $\boldsymbol{\delta}_1$, and $\mathbf{o}_1$. This is the initial configuration of the system as it waits for the first event to happen: this initial configuration determines the hidden state $\mathbf{h}(t)$ and the intensity functions $\lambda_k(t)$ over $t \in (0, t_1]$

We do not generate the BOS event but only condition on it, so the log-likelihood formula only sums over $i = 1, 2, \ldots$. This design is well-suited to various settings. In some settings, time 0 is special. For example, if we release children into a carnival and observe the sequence of their actions there, then BOS is the release event and no other events can possibly precede it. In other settings, data before time 0 are simply missing, e.g., the observation of a patient starts in midlife; nonetheless, BOS

in this case usefully indicates the beginning of the *observed* sequence. In both kinds of settings, the initial configuration just after reading BOS characterizes the model's belief about the unknown state of the true system just after time 0, as it waits for event 1. Computing the initial configuration by explicitly transitioning on BOS ensures that the initial hidden state $\mathbf{h}(0^+) \stackrel{\text{def}}{=} \lim_{t \to 0^+} \mathbf{h}(t)$ falls in the space of hidden states achievable by LSTM transitions. More important, in future work, we will be able to attach metadata about the sequence as a "mark" to the BOS event (see footnote 9), and the LSTM can learn how these metadata affect the initial configuration.

To allow finite sequences, we could optionally choose to identify one of the observable types in $\{1, 2, \ldots, K\}$ as a special end-of-sequence (EOS) event after which the sequence cannot possibly continue. If the model generates EOS, all intensities are permanently forced to 0—the LSTM is no longer consulted, so it is not necessary for the model parameters to explain why no further events are observed on the interval $[0, T]$: that is, the integral in equation (2.6) should be taken from $t = 0$ to the time of the EOS event or T, whichever is smaller.

3.A.2 Closure Under Superposition

Decomposable models have the nice property that they are closed under superposition of event sequences. Let $\mathcal{E}$ and $\mathcal{E}'$ be random event sequences, on a common time interval $[0, T]$ but over disjoint sets of event types. If each sequence is distributed according to a Hawkes process, then their superposition—that is, $\mathcal{E} \cup \mathcal{E}'$ sorted into temporally increasing order—is also distributed according to a Hawkes process. It is easy to exhibit parameters for such a process, using a block-diagonal matrix of $\alpha_{j,k}$ so that the two sets of event types do not influence each other. The closure property also holds for our decomposable self-modulating process, and for the same simple reason.

This is important since in various real settings, some event types tend not to interact. For example, the activities of two people Jay and Kay rarely influence each other,[6] although they are simultaneously monitored and thus form a single observed sequence of events. We want our model to handle such situations naturally, rather than insisting that Kay always reacts to what Jay does.

Thus, as section 3.1.1 noted, we have designed our neurally self-modulating process to preserve this ability to insulate event k from event j. By setting specific elements of $\mathbf{w}_k$ to 0, one could ensure that the intensity function $\lambda_k(t)$ depends on only a subset S of the LSTM hidden nodes. Then by setting specific LSTM parameters, one would make the nodes in S insensitive to events of type j: events of type j should open these nodes' forget gates ($\mathbf{f} = \mathbf{1}$) and close their input gates ($\mathbf{i} = \mathbf{0}$)—as section 3.1.1 suggested—so that their cell memories $\mathbf{c}(t)$ and hidden states $\mathbf{h}(t)$ do not change at all but continue decaying toward their previous steady-state values.[7] Now events of type j cannot affect the intensity $\lambda_k(t)$.

For example, the hidden states in S are affected in the same way when the LSTM reads $(k, 1), (j, 3), (j, 8), (k, 12)$ as when it reads $(k, 1), (k, 12)$, even though the intervals Δt between successive events are different. In other words, the architecture "knows" that $2 + 5 + 4 = 11$. The simplicity of this solution is a consequence of how our design does not encode the time intervals numerically, but only reacts to these intervals indirectly, through the interaction between the timing of events and

[6] Their surnames might be Box and Cox, after the 19th-century farce about a day worker and a night worker unknowingly renting the same room. But any pair of strangers would do.

[7] To be precise, we can achieve this arbitrarily closely, but not exactly, because a standard LSTM gate cannot be fully opened or closed. The openness is traditionally given by a sigmoid function and so falls in $(0, 1)$, never achieving 1 or 0 exactly unless we are willing to set parameters to $\pm\infty$. In practice this should not be an issue because relatively small weights can drive the sigmoid function extremely close to 1 and 0—in fact, $\sigma(37) = 1$ in 64-bit floating-point arithmetic.

the spontaneous decay of the hidden states. The memory cells of S decay for a total duration of 11 between the two k events, even if that interval has been divided into subintervals $2 + 5 + 4$.

With this method, we can explicitly construct a superposition process with LSTM state space $\mathbb{R}^{d+d'}$—the cross product of the state spaces $\mathbb{R}^d$ and $\mathbb{R}^{d'}$ of the original processes—in which Kay's events are not influenced at all by Jay's.

If we know *a priori* that particular event types interact only weakly, we can impose an appropriate prior on the neural Hawkes parameters. And in future work with large K, we plan to investigate the use of sparsity-inducing regularizers during parameter estimation, to create an inductive bias toward models that have limited interactions, without specifying which particular interactions are present.

Superposition is a formally natural operation on event sequences. It barely arises for ordinary sequence models, such as language models, since the superposition of two sentences is not well-defined unless all of the words carry distinct real-valued timestamps. However, there is an analogue from formal language theory. The "shuffle" of two sentences is defined to be the set of *possible* interleavings of their words—i.e., the set of superpositions that could result from assigning increasing timestamps to the words of each sentence, without duplicates. It is a standard exercise to show that regular languages are closed under shuffle. This is akin to our remark that neural-Hawkes-distributed random variables are closed under superposition, and indeed uses a similar cross-product construction on the finite-state automata. An important difference is that the shuffle construction does not require disjoint alphabets in the way that ours requires disjoint sets of event types. This is because finite-state automata allow nondeterministic state transitions and our processes do not.

Dataset	K	# of Event Tokens			Sequence Length		
		Train	Dev	Test	Min	Mean	Max
Synthetic	5	480449	60217	60139	20	60	100
Retweets	3	1739547	215521	218465	50	109	264
MemeTrack	5000	93267	14932	15440	1	3	31
MIMIC-II	75	1946	228	245	2	4	33
StackOverflow	22	343998	39247	97168	41	72	736
Financial	2	298710	33190	82900	829	2074	3319

Table 3.2: Statistics of each dataset.

3.B Experimental Details

In this appendix, we elaborate on the details of data generation, processing, and experimental results.

3.B.1 Dataset Statistics

Table 3.2 shows statistics about each dataset that we use in this chapter.

3.B.2 Training Details

We used a single-layer LSTM (Graves, 2012) in section 3.1.1, selecting the number of hidden nodes from a small set $\{64, 128, 256, 512, 1024\}$ based on the performance on the dev set of each dataset. We empirically found that the model performance is robust to these hyperparameters.

When estimating integrals with Monte Carlo sampling, N is the number of sampled negative observations in Algorithm 2.1, while I is the number of positive observations. In practice, setting $N = I$ was large enough for stable behavior, and we used this setting during training. For evaluation on dev and test data, we took $N = 10\,I$ for extra accuracy, or $N = I$ when I was very large.

DATASET	K	D	# OF MODEL PARAMETERS		
			HAWKES	GENERALIZED HAWKES	NEURAL HAWKES
SYNTHETIC	5	256	55	60	922117
RETWEETS	3	256	21	24	921091
MEMETRACK	5000	64	50005000	50010000	702856

Table 3.3: Size of each trained model on each dataset. The number of hidden nodes (D) is chosen automatically on dev data. We also tried halving D across several datasets, which had negligible effect, always decreasing held-out log-likelihood by $< 0.2\%$ relative.

For learning, we used the Adam algorithm with its default settings (Kingma and Ba, 2015). Adam is a stochastic gradient optimization algorithm that continually adjusts the learning rate in each dimension based on adaptive estimates of low-order moments. Our training objective was unregularized log-likelihood.[8] We initialized the Hawkes process parameters and s_k scale factors to 1, and all other non-LSTM parameters (section 3.1.1) to small random values from $\mathcal{N}(0, 0.01)$. We performed early stopping based on log-likelihood on the held-out dev set.

3.B.3 Model Sizes

The size of each trained model on each dataset is shown in Table 3.3. Our neural model has many parameters for expressivity, but it actually has considerably fewer parameters than the other models in the large-K setting (MemeTrack).

3.B.4 Pilot Experiments on Simulated Data

We used the thinning algorithm (section 2.3) to sample event sequences from three different processes with randomly generated parameters: (a) a standard Hawkes process (SE-MPP), (b) our decomposable self-modulating process (D-SM-MPP), (c)

[8] L_2 regularization did not appear helpful in pilot experiments, at least for our dataset size and when sharing a single regularization coefficient among all parameters.

our neural self-modulating processes (N-SM-MPP). We then tried to fit each dataset with all these models.

For each dataset, we took $K = 5$ as the number of event types. To generate each event sequence, we first chose the sequence length I (number of event tokens) uniformly from $\{20, 21, 22, \ldots, 100\}$ and then used the thinning algorithm to sample the first I events over the interval $[0, \infty)$. For subsequent training or testing, we treated this sequence (appropriately) as the complete set of events observed on the interval $[0, T]$ where $T = t_I$, the time of the last generated event. For each dataset, we generate 8000, 1000 and 1000 sequences for the training, dev, and test sets respectively.

For SE-MPP, we sampled the parameters from uniform distributions as $\mu_k \sim \mathrm{Unif}[0.0, 1.0]$, $\alpha_{j,k} \sim \mathrm{Unif}[0.0, 1.0]$, and $\delta_{j,k} \sim \mathrm{Unif}[10.0, 20.0]$. The large decay rates $\delta_{j,k}$ were needed to prevent the intensities from blowing up as the sequence accumulated more events. For D-SM-MPP, we sampled the parameters as $\mu_k \sim \mathrm{Unif}[-1.0, 1.0]$, $\alpha_{j,k} \sim \mathrm{Unif}[-1.0, 1.0]$, and $\delta_{j,k} \sim \mathrm{Unif}[10.0, 20.0]$. For N-SM-MPP, we sampled parameters from $\mathrm{Unif}[-1.0, 1.0]$.

The results are shown in Figure 3.2, including log-likelihood (reported in nats per event) on the sequences and the breakdown of time interval and event types.

Another interesting question is whether the trained neural Hawkes model accurately predicts the real-valued *intensities*, since for the synthetic data we actually know the intensities. This is a more direct evaluation of whether the model is accurately recovering the dynamics of the underlying generative process. Here we compared only SE-MPP and N-SM-MPP.

All types behaved similarly, so we report only averages over the K types. For both processes (a) and (c), the true intensity's variance was about 30% of the squared

mean intensity. Thus, the intensity changes enough over time that predicting it at particular times is not a trivial challenge. To determine how well a model predicted the true intensity function, we measured the mean squared error (MSE) of predicted intensity at a large sample of times in the held-out test seqs, and report the MSE here as a percentage of the *variance* of the true intensity. By this construction, a simple baseline of predicting each event type's mean intensity at all times would get 100% MSE.

Both the Hawkes and neural-Hawkes models predict the Hawkes intensities (a) accurately, at 1% MSE. This is similar to the leftmost column of Figure 3.2, where both models essentially achieved oracle performance. By contrast, for the complex neural Hawkes intensities (c), the neural Hawkes model achieves 9% MSE (still quite good) whereas Hawkes does far worse at 70% MSE. This is similar to the rightmost column of Figure 3.2, where the neural Hawkes model approached oracle performance but the Hawkes model did much worse.

3.B.5 Retweet Dataset Details

The Retweets dataset (section 3.3.2) includes 166076 retweet sequences, each corresponding to some original tweet. Each retweet event is labeled with the retweet time relative to the original tweet creation, so that the time of the original tweet is 0. (The original tweet serves as the beginning-of-sequence (BOS) marker as explained in section 3.A.1.) Each retweet event is also marked with the number of followers of the retweeter. As usual, we assume that these 166076 sequences are drawn independently from the same process, so that retweets in different sequences do not affect one another.

Unfortunately, the dataset does not specify the identity of each retweeter, only his

or her popularity. To distinguish different kinds of events that might have different rates and different influences on the future, we divide the events into $K = 3$ types: retweets by "small," "medium" and "large" users. Small users have fewer than 120 followers (50% of events), medium users have fewer than 1363 (45% of events), and the rest are large users (5% events). Given the past retweet history, our model must learn to predict how soon it will be retweeted again and how popular the retweeter is (i.e., which of the three categories).

We randomly sampled disjoint train, dev and test sets with 16000, 2000 and 2000 sequences respectively. We truncated sequences to a maximum length of 264, which affected 20% of them. For computing training and test likelihoods, we treated each sequence as the complete set of events observed on the interval $[0, T]$, where 0 denotes the time of the original tweet (which is not included in the sequence) and T denotes the time of the last tweet in the (truncated) sequence.

Figure 3.7 shows the learning curves of all the models, broken down by the log-probabilities of the event types and the time intervals separately. Figure 3.8 breaks down the log-likelihood by event type and time interval.

3.B.6 MemeTrack Dataset Details

The MemeTrack dataset (section 3.3.2) contains time-stamped instances of meme use in articles and posts from 1.5 million different blogs and news sites, spanning 10 months from August 2008 till May 2009, with several hundred million documents.

As in Retweets, we decline to model the appearance of novel memes. Each novel meme serves as the BOS event for a sequence of mentions on other websites, which we do model. The K event types correspond to the different websites. Given one meme's past trajectory across websites, our model must learn to predict how soon it

will be mentioned again and where.

We used the version of the dataset processed by Gomez Rodriguez, Leskovec, and Schölkopf (2013), which selected the top 5000 websites in terms of the number of memes they mentioned. We truncated sequences to a maximum length of 32, which affected only 1% of them. We randomly sampled disjoint train, dev and test sets with 32000, 5000 and 5000 sequences respectively, treating them as before.

Because our current implementation does not allow for a marked BOS event (see section 3.A.1), we currently ignore where the novel meme was originally posted, making the unfortunate assumption that the sequence of websites is independent of the originating website. Even worse, we must assume that the sequence of websites is independent of the actual text of the meme. However, as we see, our novel models have some ability to recover from these forms of missing data.

Figure 3.9 shows the learning curves of the breakdown of log-likelihood with the same format as Figure 3.7. Figures 3.10 and 3.11 show the scatterplots in the same format as Figures 3.4 and 3.8.

3.B.7 Prediction Task Details

Finally, we give further details of the prediction experiments from section 3.3.5. To avoid tuning on the test data, we split the original training set into a new training set and a held-out dev set. We train our neural model and that of Du et al. (2016) on the new training set, and choose hyper-parameters on the held-out dev set. Following Du et al. (2016), we consider three datasets, and use five different train-dev-test splits of each dataset to generate the experimental results in Figure 3.6. (None of the test sets' examples were used during manual development of our system.)

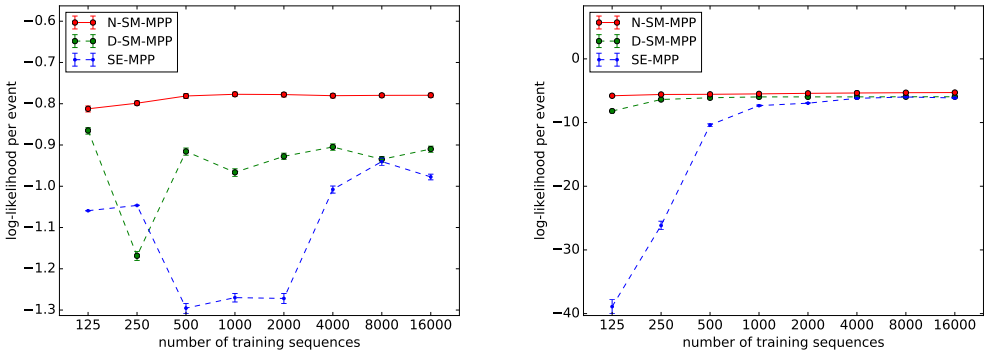


Figure 3.7: Learning curves (with 95% error bars) of all these models on the Retweets dataset, broken down by the log-probabilities of just the event types (left graph) and just the time intervals (right graph).

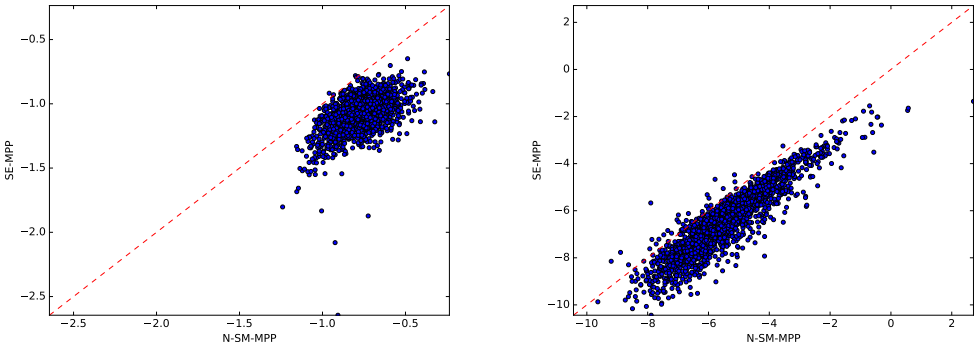


Figure 3.8: Scatterplots of neural Hawkes (N-SM-MPP) vs. Hawkes (SE-MPP) on Retweets. Same comparison as the left graph in Figure 3.4, but broken down by the log-probabilities of the event types (left graph) and the time intervals (right graph).

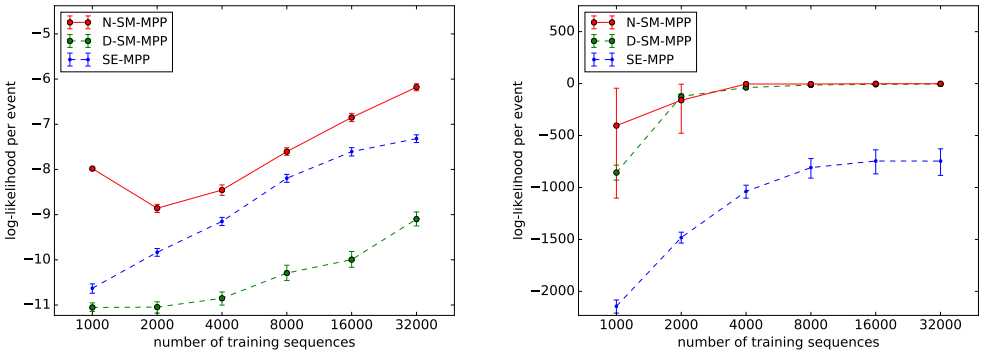


Figure 3.9: Learning curve (with 95% error bars) of all three models on the MemeTrack dataset, broken down by the log-probabilities of the event types (left graph) and the time intervals (right graph).

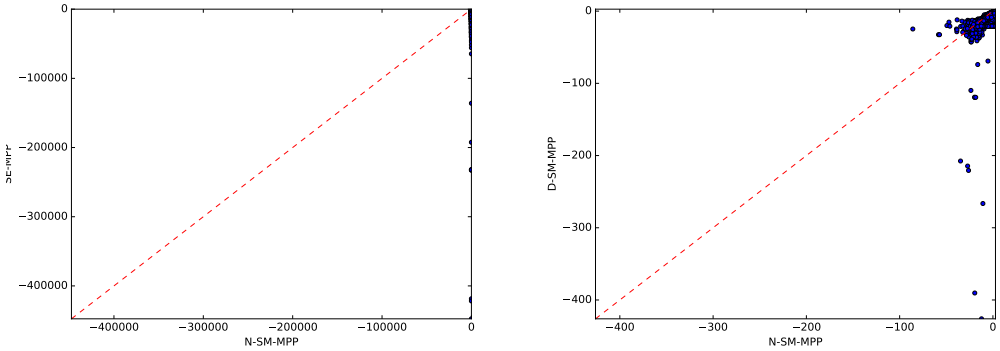


Figure 3.10: Scatterplot of on the MemeTrack dataset. The left is neural Hawkes (N-SM-MPP) vs. Hawkes (SE-MPP); the right is neural Hawkes (N-SM-MPP) vs. generalized Hawkes (D-SM-MPP). The neural model outperforms the generalized Hawkes process on 93.02% of the test sequences. This is not obvious from the plot, because almost all of the 5000 points are crowded near the upper right corner. Most of the visible points are outliers where the neural model performs unusually badly—and the generalized Hawkes process typically does even worse.

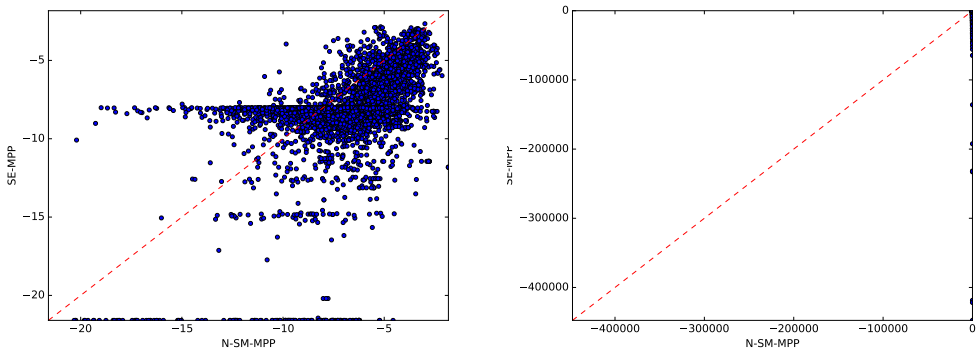


Figure 3.11: Scatterplots of neural Hawkes (N-SM-MPP) vs. Hawkes (SE-MPP) on MemeTrack. Same comparison as the left graph of Figure 3.10, but broken down by the log-probabilities of the event types (left graph) and the time intervals (right graph).

3.C Ongoing and Future Work

We are currently exploring several extensions to deal with more complex datasets. Based on our survey of existing datasets, we are particularly interested in handling:

- "baskets" of events (several events that are recorded as occuring simultaneously but without a specified order, e.g., the purchase of an entire shopping cart)
- hard constraints on the event type sequence $k_1, k_2, \ldots$
- marked events[9] and annotated events[10]
- causation by external events (artificial clock ticks, periodic holidays, weather)
- richer drift functions[11]
- hybrid of D-SM-MPP and N-SM-MPP, allowing direct influence from past events
- multiple agents each with their own state, who observe one another's actions

[9] A "mark" is some structured data attached to an event: for example, the textual content associated with a tweet, or the medical records associated with a doctor visit. The model should predict the marks from each event and its underlying hidden state, and they should be fed back into the LSTM as additional input.

[10] Humans may be asked to classify the events in an event sequence or the relationships among its events. Unlike marks, these annotations are not involved in the process that generates the event sequence, and so are not fed into the LSTM as input. Rather, they are assumed to be generated *post hoc* by the human from the entire observed sequence—and may depend on the human's implicit reconstruction of the hidden states. We can use any available annotations to help reconstruct the hidden states (F. Zaidan and Eisner, 2008), if we model them as stochastic functions of the hidden states. In particular, annotations on the training data serve as side information to improve training of the model. As a simple example, an annotation of the training event (k_i, t_i) could be assumed to depend also on the subsequent LSTM state $\mathbf{h}(t_i^+) \stackrel{\text{def}}{=} \lim_{t \to t_i^+} \mathbf{h}(t)$.

[11] We expect the exponential drift in equation (3.3) to be expressive enough in most settings. In principle, however, one might want to allow periodic fluctuation of the intensity between events, say by using a *complex* exponential in (3.3). Another way to increase expressivity would be to compute drift using the LSTM itself, by injecting special "clock tick" events into the input sequence at regular intervals (compare Xiao et al., 2017b). Each clock tick event (k_i, t_i) causes a rich nonlinear update of the LSTM state via equations (3.2)–(3.2), except that it should always set $\mathbf{c}_{i+1} = \mathbf{c}(t_i)$ for continuity. In this design, the interval between ordinary events is modeled piecewise—it is divided up into short pieces by the clock ticks, with $\mathbf{c}(t)$ on each piece modeled using our current function family.

(events)

Several of the above have been dealt with or further discussed in Chapter 4.

More important, we are interested in modeling causality. The current model might pick up that a hospital visit elevates the instantaneous probability of death, but this does not imply that a hospital visit *causes* death. (In fact, the severity of an earlier illness is usually the cause of both.)

A model that can predict the result of interventions is called a causal model. Our model family can naturally be used here: any choice of parameters defines a generative story that follows the arrow of time, which can be interpreted as a causal model in which patterns of earlier events *cause* later events to be more likely. Such a causal model predicts how the distribution over futures would change if we intervened in the sequence of events.

In general, one cannot determine the parameters of a causal model based on purely observational data (Pearl, 2009). Thus, in future, we plan to determine such parameters through randomized experiments by deploying our model family as an environment model within reinforcement learning. A reinforcement learning agent *tests* the effect of random interventions to discover their effect (exploration) and thus orchestrate more rewarding futures (exploitation).

In our setting, the agent is able to stochastically insert or suppress certain event types and observe the effect on subsequent events. Then our LSTM-based model will discover the causal effects of such actions, and the reinforcement learner will discover what actions it can take to affect future reward. Ultimately this could be a vehicle for personalized medical decision-making. Beyond the medical domain, a quantified-self smartphone app may intervene by displaying fine-grained advice on eating, sleeping,

exercise, and travel; a charitable agency may intervene by sending a social worker to provide timely counseling or material support; a social media website may increase positive engagement by intelligently distributing posts; or a marketer may stimulate consumption by sending more targeted advertisements.

Chapter 4

A Neural-Symbolic Hybrid: Neural Datalog Through Time

Chapter 3 presented the neural Hawkes process, an unrestricted neural model. Training such an unrestricted neural model might overfit to spurious patterns, particularly when the set of possible event types is large. In this chapter, we present the neural Datalog through time, our novel neural-symbolic hybrid model, which is able to exploit domain-specific knowledge of how past events might affect an event's present probability. We propose a novel modeling language, by which a user can write knowledge as rules. And we propose using a *temporal deductive database* to track structured facts over time. Rules serve to prove facts from other facts and from past events. Each fact has a time-varying state—a vector computed by a neural net whose topology is determined by the fact's *provenance*, including its experience of past events. The possible event types at any time are given by special facts, whose *probabilities* are neurally modeled alongside their states. In both synthetic and real-world domains, we show that neural probabilistic models derived from concise logic programs improve prediction by encoding appropriate domain knowledge in their architecture.

4.1 Motivation

As discussed in Chapter 1, event sequences are abundant in applied machine learning. A common task is to predict the future from the past. Often this is done by fitting a generative probability model.

Under an unrestricted neural model like our neural Hawkes process, each event e_i updates the state of the system from $\mathbf{s}_i$ to $\mathbf{s}_{i+1}$, which then determines the distribution from which the next event e_{i+1} is drawn.Alas, when the relationship between events and the system state is unrestricted—when anything can potentially affect anything—fitting an accurate model is very difficult, particularly in a real-world domain that allows millions of event types including many rare types. Thus, one would like to introduce domain-specific structure into the model.

For example, one might declare that the probability that Alice travels to Chicago is determined entirely by Alice's state, the states of Alice's coworkers such as Bob, and the state of affairs in Chicago. Given that modeling assumption, parameter estimation can no longer incorrectly overfit this probability using spurious features based on unrelated temporal patterns of (say) wheat sales and soccer goals.

To improve extrapolation, one can reuse this "Alice travels to Chicago" model for any person `A` traveling to any place `C`. Our main contribution is a modeling language that can concisely model all these `travel(A,C)` probabilities using a few rules over variables `A`, `B`, `C`. Here `B` ranges over `A`'s coworkers, where the `coworker` relation is also governed by rules and can itself be affected by stochastic events.

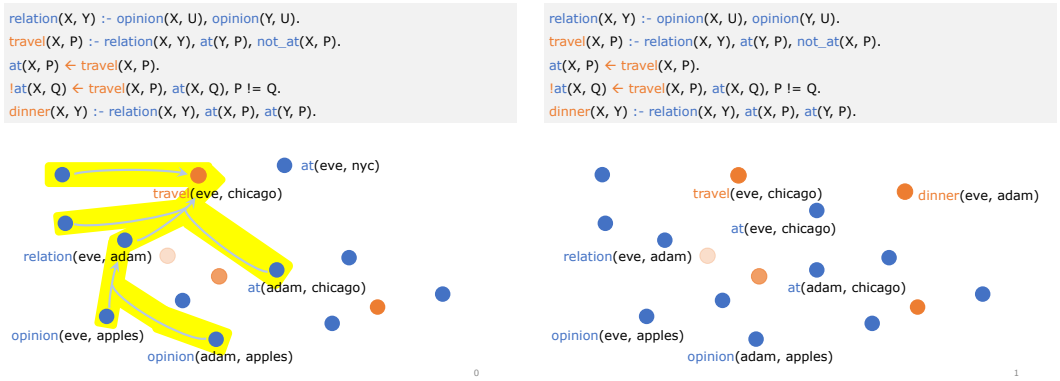
4.2 An Overview of Our Neural-Symbolic Paradigm

In our paradigm, a domain expert simply writes down the rules of a **temporal deductive database**, which tracks the possible event types and other *boolean* facts over time. This logic program is then used to automatically construct a deep recurrent neural architecture, whose distributed state consists of vector-space *embeddings* of all present facts. Its output specifies the distribution of the next event.

Logic Rules. What sort of rules? An event has a structured description with zero or more participating entities. When an event happens, pattern-matching against its description triggers **update rules**, which modify the database facts to reflect the new properties and relationships of these entities. Updates may have a cascading effect if the database contains **deductive rules** that derive further facts from existing ones at any time. (For example, `coworker(A,B)` is jointly implied by `boss(U,A)` and `boss(U,B)`). In particular, deductive rules can state that entities combine into a possible event type whenever they have the appropriate properties and relationships. (For example, `travel(A,C)` is possible if `C` is a place and `A` is a person who is not already at `C`.)

Fact Embeddings and Event Probabilities Since the database defines possible events and is updated by the event that happens, it already resembles the system state $\mathbf{s}_i$ of a temporal model. We enrich this logical state by associating an embedding with each fact currently in the database. This time-varying vector represents the state of *that fact*; recall that the set of facts may also change over time. When a fact is added by events or derived from other facts, its embedding is derived from their embeddings in a standard way, using parameters associated with the rules that established the fact. In this way, the model's rules together with the past events and the initial facts

Figure 4.1: Two snapshots of a deductive database of fact embeddings and event probabilities. Each dot is the two-dimensional projection of a multi-dimensional fact embedding; orange dots denote facts that declare possible events, and their brightness denotes the probabilities. These snapshots have the same set of temporal Datalog rules, but different facts, fact embeddings, and event probabilities.



(a) The database of fact embeddings and event probabilities when `eve` is still at `nyc`. The yellow-shaded hyperedges illustrate how the embedding of `travel(eve,chicago)` is *recursively* computed from the embeddings of other facts.

(b) When `travel(eve,chicago)` actually happened, the database was *updated*: the new fact `at(eve,chicago)` was deduced; it then contributed to deducing `dinner(eve,adam)` as well as computing its embedding and probability.

define the topology of a deep recurrent neural architecture, which can be trained via back-propagation through time (Williams and Zipser, 1989). For the facts that state that specific event types are possible, the architecture computes not only embeddings but also the probabilities of these event types. Figure 4.1 illustrates this paradigm with an example domain where people travel across different places.

Scalability and Generalizability The number of parameters of such a model grows only with the number of rules, not with the much larger number of event types or other facts. This is analogous to how a probabilistic relational model (Getoor and Taskar, 2007; Richardson and Domingos, 2006) derives a graphical model structure from a database, building random variables from database entities and repeating subgraphs with shared parameters. Unlike graphical models, ours is a neural-symbolic

hybrid. The system state $\mathbf{s}_i$ includes both rule-governed discrete elements (the set of facts) and learned continuous elements (the embeddings of those facts). It can learn a neural probabilistic model of people's movements while relying on a discrete symbolic deductive database to cheaply and accurately record who is where. A purely neural model such as our neural Hawkes process would have to *learn* how to encode every location fact in some very high-dimensional state vector, and retain and update it, with no generalization across people and places.

In our experiments, we show how to write down some logic programs that derive domain-specific models for event sequences, and demonstrate that their structure improves their ability to predict held-out data.

4.3 Our Modeling Language

We gradually introduce our specification language by developing a fragment of a human activity model. Similar examples could be developed in many other domains—epidemiology, medicine, education, organizational behavior, consumer behavior, economic supply chains, etc. Such specifications can be trained and evaluated using our implementation, which can be found at `https://github.com/HMEIatJHU/neural-datalog-through-time`.

For pedagogical reasons, section 4.3 will focus on our high-level scheme (see also the animated drawings in our ICML 2020 talk video). We defer the actual neural formulas until section 4.4.

4.3.1 Datalog

We adapt our notation from Datalog (Ceri, Gottlob, and Tanca, 1989), where one can write **deductive rules** of the form

$$head \texttt{ :- } condit_1\texttt{, ..., } condit_N\texttt{.} \tag{4.1}$$

Such a rule states that the head is true provided that the conditions are all true. In a simple case, the head and conditions are **atoms**, i.e., structured terms that represent boolean propositions. For example,

```
1 compatible(eve,adam) :- likes(eve,apples), likes(adam,apples).
```

If $N = 0$, the rule simply states that the head is true. This case is useful to assert basic facts:

```
2 likes(eve,apples).
```

Notice that in this case, the :- symbol is omitted.

A rule that contains **variables** (capitalized identifiers) represents the infinite collection of **ground** rules obtained by instantiating (grounding) those variables. For example,

```
3 compatible(X,Y) :- likes(X,U), likes(Y,U).
```

says that *any* two entities `X` and `Y` are compatible provided that there exists *any* `U` that they both like.

A Datalog **program** is an unordered set of rules. The atoms that can be proved from these rules are called **facts**. Given a program, one would use $[\![h]\!] \in \{\mathsf{true}, \mathsf{null}\}$ to denote the semantic value of atom h, where $[\![h]\!] = \mathsf{true}$ iff h is a fact.

4.3.2 Neural Datalog

In our formalism, a fact has an **embedding** in a vector space, so the semantic value of atom `likes(eve,apples)` describes more than just *whether* `eve` likes `apples`. To indicate this, let us rename and colorize the functors in line 3:

```
4  rel(X,Y) :- opinion(X,U), opinion(Y,U).
```

Now $[\![$`opinion(eve,apples)`$]\!]$ is a vector describing `eve`'s complex opinion about apples (or null if she has no opinion). $[\![$`rel(eve,adam)`$]\!]$ is a vector describing `eve` and `adam`'s relationship (or null if they have none).

With this extension, $[\![h]\!] \in \mathbb{R}^{D_h} \cup \{\mathsf{null}\}$, where the embedding dimension D_h depends on the atom h. The declaration

```
5  :- embed(opinion,8).
```

says that if h has the form `opinion(...)` then $D_h = 8$.[1]

When an atom is proved via a rule, its embedding is affected by the conditions of that rule, in a way that depends on trainable parameters associated with that rule. For example, according to line 4, $[\![$`rel(eve,adam)`$]\!]$ is a parametric function of the opinion vectors that `eve` and `adam` have about various topics `U`. The influences from all their shared topics are pooled together as detailed in section 4.4.1 below.

A model might say that *each* person has an opinion about *each* food, which is a function of the embeddings of the person and the food, using parameters associated with line 6:

```
6  opinion(X,U) :- person(X), food(U).
```

If the foods are simply declared as basic facts, as follows, then each food's embedding

[1] In the absence of such a declaration, $D_h = 0$. Then $[\![h]\!]$ has only two possible values, just as in Datalog; we do not color h.

is independently specified by the parameters associated with the rule that declares it:

```
7  food(apples).
8  food(manna).
     ⋮
```

Given all the rules above, whenever `person(X)` and `person(Y)` are facts, it follows that `rel(X,Y)` is a fact, and ⟦`rel(X,Y)`⟧ is defined by a multi-layer feed-forward neural network whose topology is given by the proof DAG for `rel(X,Y)`. The network details will be given in section 4.4.1.

Recursive Datalog rules can lead to arbitrarily deep networks that recursively build up a compositional embedding, just as in sequence encoders (Elman, 1990), tree encoders (Socher et al., 2012; Tai, Socher, and Manning, 2015), and DAG encoders (Goller and Kuchler, 1996; Le and Zuidema, 2015)—all of which could be implemented in our formalism. E.g.:

```
9  cursed(cain).
10 cursed(Y) :- cursed(X), parent(X,Y).
```

In Datalog, this system simply states that all descendants of `cain` are cursed. In neural Datalog, however, a child has a *specific* curse: a vector ⟦`cursed(Y)`⟧ that is computed from the parent's curse ⟦`cursed(X)`⟧ in a way that also depends on their relationship, as encoded by the vector ⟦`parent(X,Y)`⟧. Line 10's parameters model how the curse evolves (and hopefully attenuates) as each generation is re-cursed. Notice that ⟦`cursed(Y)`⟧ is essentially computed by a recurrent neural network that encodes the sequence of `parent` edges that connect `cain` to `Y`.[2]

We currently consider it to be a model specification error if any atom h participates

[2] Assuming that this path is unique. More generally, `Y` might descend from `cain` by multiple paths. The computation actually encodes the DAG of *all* paths, by pooling over all of `Y`'s cursed parents at each step, just as line 4 pooled over multiple topics.

in its own proof, leading to a circular definition of $[\![h]\!]$. This would happen in lines 9–10 only if `parent` were bizarrely defined to make some cursed person their own ancestor. section 4.A.1 discusses extensions that would define $[\![h]\!]$ even in these cyclic cases.

4.3.3 Datalog Through Time

For temporal modeling, we use atoms such as `help(X,Y)` as the structured names for events. We underline their functors. As usual, we colorize them if they have vector-space embeddings (see footnote 1), but as orange rather than blue.

We extend Datalog with **update rules** so that whenever a `help(X,Y)` event occurs under appropriate conditions, it can add to the database by proving new atoms:

```
11 grateful(Y,X) <- help(X,Y), person(Y).
```

An event can also cancel out such additions, which may make atoms false again.[3] The `!` symbol means "not":

```
12 !grateful(Y,X) <- harm(X,Y).
```

The general form of these **update rules** is

$$head \leftarrow event,\ condit_1,\ \ldots,\ condit_N. \tag{4.2a}$$

$$!head \leftarrow event,\ condit_1,\ \ldots,\ condit_N. \tag{4.2b}$$

which state that *event* makes *head* true or false, respectively, provided that the conditions are all true. An event occurring at time s affects the set of facts at times $t > s$, both directly through <- rules, and also indirectly, since the facts added or removed by <- rules may affect the set of additional facts that can be derived by

[3] The atom will remain true if it remains provable by a :- rule, or is proved by another <- rule at the same time.

:- rules at time t. Our approach can be used for either discrete time ($s, t \in \mathbb{N}$) or continuous time ($s, t \in \mathbb{R}_{\geq 0}$), where the latter supports irregularly spaced events, i.e., the focus of this thesis.

4.3.4 Neural Datalog Through Time

In section 4.3.2, we derived each fact's embedding from its proof DAG, representing its set of Datalog proofs. For Datalog through time, we must also consider how to embed facts that were proved by an earlier update. Furthermore, once an atom is proved, an update rule can prove it again. This will update its embedding, in keeping with our principle that a fact's embedding is influenced by *all* of its proofs.

As an example, when `X` helps `Y` and `grateful(Y,X)` first becomes true via line 11, the new embedding $[\![$`grateful(Y,X)`$]\!]$ is computed—using parameters associated with line 11—from the embeddings of `help(X,Y)` and `person(Y)`. Those embeddings model the nature of the help and the state of person `Y`. (This was the main reason for line 11 to include `person(Y)` as a condition.) Each time `X` helps `Y` again, $[\![$`grateful(Y,X)`$]\!]$ is further updated by line 11, so this gratitude vector records the *history* of help. The updates are LSTM-like (see section 4.4.3 for details).

In general, an atom's semantics can now vary over time and so should be denoted as $[\![h]\!](t)$: the **state** of atom h at time t, which is part of the overall database state. A :- rule as in equation (4.1) says that $[\![head]\!](t)$ depends parametrically on $\{[\![condit_i]\!](t) : 1 \leq i \leq N\}$. A <- rule as in equation (4.2a) says that if $event$ occurred at time $s < t$ and no events updating $head$ occurred on the time interval (s, t), then $[\![head]\!](t)$ depends parametrically on its previous value[4] $[\![head]\!](s)$ along

[4]More precisely, it depends on the LSTM cells that contributed to that previous value, as we will see in section 4.4.3.

with $[\![\texttt{event}]\!](s)$, $\{[\![\texttt{condit}_i]\!](s) : 1 \leq i \leq N\}$, and the elapsed time $t - s$. We will detail the parametric formulas in section 4.4.3.

Thus, $[\![\texttt{head}]\!](t)$ depends via :- rules on *head*'s *provenance* in the database at time t, and depends via <- rules on its *experience* of events at strictly earlier times.[5] This yields a neural architecture similar to a stacked LSTM: the :- rules make the neural network deep at a single time step, while the <- rules make it temporally recurrent across time steps. The network's irregular topology is defined by the :- and <- rules plus the events that have occurred.

4.3.5 Probabilistic Modeling of Event Sequences

Because events can **occur**, atoms that represent event types are special. They can be declared as follows:

```
13  :- event(help, 8).
```

Because the declaration is `event` rather than `embed`, at times when help(X,Y) is a fact, it will have a positive probability along with its embedding $[\![\texttt{help(X,Y)}]\!] \in \mathbb{R}^8$. This is what the underlined functor really indicates.

At times s when help(X,Y) is not a fact, the semantic value $[\![\texttt{help(X,Y)}]\!](s)$ will be null, and it will have neither an embedding nor a probability. At these times, it is simply not a possible event; its probability is effectively 0.

Thus, the model must include rules that establish the set of possible events as facts. For example, the rule

```
14  help(X,Y) :- rel(X,Y).
```

says if X and Y have a relationship, then help(X,Y) is true, meaning that events of the type help(X,Y) have positive probability (i.e., X can help Y). The embedding

[5] See section 4.4.3 for the precise interaction of :- and <- rules.

and probability are computed deterministically from ⟦rel(X,Y)⟧ using parameters associated with line 14, as detailed in section 4.4.2.

Now a neural-Datalog-through-time program specifies a probabilistic model over event sequences. Each stochastic event can update some database facts or their embeddings, as well as the probability distribution over possible next events. As section 4.2 outlined, each *stochastic draw* from the next-event distribution results in a *deterministic update* to that distribution—just as in a recurrent neural network language model Mikolov et al., 2010; Sundermeyer, Ney, and Schluter, 2012.

Our approach also allows the possibility of **exogenous** events that are not generated by the model, but are given externally. Our probabilistic model is then *conditioned* on these exogenous events. The model itself might have probability 0 of generating these event types at those times. Indeed, if an event type is to occur *only* exogenously, then the model should not predict any probability for it, so it should not be declared using `event`. We use a dashed underline for undeclared events since they have no probability.

For example, we might wish to use rules of the form *head* `<-` earthquake(C),... to model how an earthquake in city `C` tends to affect subsequent events, even if we do not care to model the *probabilities* of earthquakes. The *embeddings* of possible earthquake events can still be determined by parametric rules, e.g., `earthquake(C) :- city(C)`, if we request them by declaring `embed(earthquake,5)`.

4.3.6 Continuing the Example

In our example, the following rules are also plausible. They say that when `X` helps `Y`, this event updates the states of the helper `X` and the helpee `Y` and also the state of their relationship:

```
15  person(X) <- help(X,Y).
16  person(Y) <- help(X,Y)
17  rel(X,Y)  <- help(X,Y).
```

To enrich the model further, we could add (e.g.) `rel(X,Y)` as a condition to these rules. Then the update when `X` helps `Y` depends quantitatively on the state of their relationship.

There may be many other kinds of events observed in a human activity dataset, such as `sleep(X)`, `eat(X)`, `email(X,Y)`, `invite(X,Y)`, `hire(X,Y)`, etc. These can be treated similarly to `help(X,Y)`.

Our modeling architecture is intended to limit dependencies to those that are explicitly specified, just as in graphical models. However, the resulting independence assumptions may be too strong. To allow unanticipated influences back into the model, it can be useful to include a low-dimensional global state, which is updated by all events:

```
18  world <- help(X,Y).
      ⋮
```

`world` records a "public history" in its state, and it can be a condition for any rule. E.g., we can replace line 14 with

```
19  help(X,Y) :- rel(X,Y), world.
```

so that `eve`'s probability of helping `adam` might be affected by the history of other individuals' interactions.

Eventually `eve` and `adam` may die, which means that they are no longer available to help or be helped:

```
20  die(X) :- person(X).
```

If we want `person(eve)` to then become false, the model cannot place that atom in the database with a `:-` rule like

```
21 person(eve).
```

which would ensure that `person(eve)` can *always* be proved. Instead, we use a `<-` rule that initially adds `person(eve)` to the database via a special event, `init`, that always occurs exogenously at time $t = 0$:

```
22 person(eve) <- init.
```

With this treatment, the following rule can remove `person(eve)` again when she dies:

```
23 !person(X) <- die(X).
```

The reader may enjoy extending this model to handle possessions, movement, tribal membership/organization, etc.

4.3.7 Finiteness

Under our formalism, any given model allows only a finite set of possible events. This is because a Datalog program's facts are constructed by using functors mentioned in the program, with arguments mentioned in the program,[6] and nesting is disallowed. Thus, the set of facts is finite (though perhaps much larger than the length of the program).

It is this property that will ensure in section 4.4.2 that our probability model—which sums over all possible events—is well-defined. Yet this is also a limitation. In some domains, a model should not really place any *a priori* bound on the number of event types, since an infinite sequence may contain infinitely many distinct types—the

[6] A rule such as `likes(adam,Y) :- likes(adam,eve)` might be able to prove that `adam` likes everyone, including infinitely many unmentioned entities. To preserve finiteness, such rules are illegal in Datalog. A Datalog rule must be **range-restricted**: any variable in the head must also appear in the body.

number of types represented in the length-n prefix grows unboundedly with n. Even our running example should really support the addition of new entities: the event `procreate(eve,adam)` should result in a fact such as `person(cain)`, where `cain` is a newly allocated entity. Similarly, new species are allocated in the course of drawing a sequence from Fisher's (1943) species-sampling model or from a Chinese restaurant process; new words are allocated as a document is drawn from an infinite-vocabulary language model; and new real numbers are constantly encountered in a sequence of sensor readings. In these domains, no model can *prespecify* all the entities that can appear in a dataset. section 4.A.4 discusses potential extensions to handle these cases.

4.4 Formulas Associated With Rules

4.4.1 Neural Datalog

Recall from section 4.3.1 that if h is a fact, it is provable by at least one `:-` rule in at least one way. For neural Datalog (section 4.3.2), we then choose to define the embedding $[\![h]\!] \neq \mathsf{null}$ as

$$[\![h]\!] \stackrel{\text{def}}{=} \tanh\left(\sum_r [h]_r^{:-}\right) \in (-1,1)^{D_h} \tag{4.3}$$

where $[h]_r^{:-}$ represents the **contribution** of the r^{th} rule of the Datalog program. For example, $[\![$`opinion(eve,apples)`$]\!]$ receives non-zero contributions from *both* line 2 and line 6.[7] For a given `Y`, $[\![$`cursed(Y)`$]\!]$ may receive a non-zero contribution from line 9, line 10, or neither, according to whether `Y` is `cain` himself, a descendant of `cain`, or neither.

The contribution $[h]_r^{:-}$ has been pooled over all the ways (if any) that the r^{th} rule proves h. For example, for any entity `Y`, $[$`cursed(Y)`$]_{10}^{:-}$ needs to compute the

[7] Recall that we renamed `likes` in line 2 to `opinion`.

aggregate effect of the curses that `Y` inherits through *all* of `Y`'s cursed parents `X` in line 10. Similarly, $[\texttt{rel(X,Y)}]^{:-}_4$ computes the aggregate effect on the relationship from *all* of `X` and `Y`'s shared interests `U` in line 4. Recall from section 4.3.1 that a rule with variables represents a collection of ground rules obtained by instantiating those variables. We define its contribution by

$$[\![h]\!]^{:-}_r \stackrel{\text{def}}{=} \bigoplus^{\beta_r}_{g_1,\ldots,g_N} \mathbf{W}_r \underbrace{[1; [\![g_1]\!]; \ldots; [\![g_N]\!]]}_{\text{concatenation of column vectors}} \in \mathbb{R}^{D_h} \tag{4.4}$$

where for the summation, we allow h `:-` g_1`,` $\ldots$`,` g_N to range over all instantiations of the r^{th} rule such that the head equals h and $g_1, \ldots, g_N$ are all facts. There are only finitely many such instantiations (see section 4.3.7). $\mathbf{W}_r$ is a conformable parameter matrix associated with the r^{th} rule. (section 4.B offers extensions that allow more control over how parameters are shared among and within rules.)

The pooling operator $\bigoplus^{\beta}$ that we used above is defined to aggregate a set of vectors $\{\mathbf{x}_1, \ldots, \mathbf{x}_M\}$:

$$\bigoplus^{\beta}_{m} \mathbf{x}_m \stackrel{\text{def}}{=} v^{-1}(\sum_m v(\mathbf{x}_m)) \tag{4.5}$$

Remarks: For any definition of function v with inverse v^{-1}, $\bigoplus^{\beta}$ has a unique identity element, $v^{-1}(\mathbf{0})$, which is also the result of pooling no vectors ($M = 0$). Pooling a single vector ($M = 1$) returns that vector—so when rule r proves h in only one way, the contribution of the $[\![g_i]\!]$ to $[\![h]\!]$ does not have to involve an "extra" nonlinear pooling step in equation (4.4), but only the nonlinear $\tanh$ in equation (4.3).

Given $\beta \neq 0$, we take v to be the differentiable function

$$v(\mathbf{x}) \stackrel{\text{def}}{=} \text{sign}(\mathbf{x}) \, |\mathbf{x}|^{\beta} \tag{4.6a}$$

$$v^{-1}(\mathbf{y}) = \text{sign}(\mathbf{y}) \, |\mathbf{y}|^{1/\beta} \tag{4.6b}$$

where all operations are applied elementwise. Now the result of aggregating no vectors is $\mathbf{0}$, so rules that achieve no proofs of h contribute nothing to equation (4.3). If $\beta = 1$, then $v = \text{identity}$ and $\bigoplus^\beta$ is just summation. As $\beta \to \infty$, $\bigoplus^\beta$ emphasizes more extreme values, approaching a signed variant of max-pooling that chooses (elementwise) the argument with the largest absolute value. As a generalization, one could replace the scalar β with a vector $\boldsymbol{\beta}$, so that different dimensions are pooled differently. Pooling is scale-invariant: $\bigoplus_m^\beta \alpha \mathbf{x}_m = \alpha \bigoplus_m^\beta \mathbf{x}_m$ for $\alpha \in \mathbb{R}$.

For each rule r, we learn a scalar β_r,[8] and use $\bigoplus^{\beta_r}$ in equation (4.4).

4.4.2 Probabilities and Intensities

When a fact h has been declared by `event` to represent an event type, we need it to have not only an embedding but also a positive probability. We extend our setup by appending an extra row to the matrix $\mathbf{W}_r$ in equation (4.4), leading to an extra element in the column vectors $[h]_r^{:-}$. We then pass only the first D_h elements of $\sum_r [h]_r^{:-}$ through $\tanh$, obtaining the same $[\![h]\!]$ as equation (4.3) gave before. We pass the one remaining element through an $\exp$ function to obtain $\lambda_h > 0$.

Recall that for neural Datalog through time (section 4.3.4), all these quantities, including λ_h, vary with the time t. To model a discrete-time event sequence, define the **probability** of an event of type h at time step t to be proportional to $\lambda_e(t)$, normalizing over all event types that are possible then. This imitates the softmax distributions in other neural sequence models (Mikolov et al., 2010; Sundermeyer, Ney, and Schluter,

[8] It can be parameterized as $\beta = \exp b > 0$ (ensuring that aggregating positive numbers exceeds their max), or as $\beta = 1 + b^2 \geq 1$ (ensuring that the aggregate of positive numbers also does not exceed their sum). Our present experiments do the latter.

2012).

When time is continuous, as in our experiments (section 4.7), we need instantaneous probabilities. We take $\lambda_{\boldsymbol{h}}(t)$ to be the (Poisson) **intensity** of $\boldsymbol{h}$ at time t: that is, it models the limit as $dt \to 0^+$ of the expected *rate* of $\boldsymbol{h}$ on the interval $[t, t+dt)$ (i.e., the expected number of occurrences of $\boldsymbol{h}$ divided by dt). This follows the setup of our neural Hawkes process (Mei and Eisner, 2017). Also following that paper, we replace $\exp(x) > 0$ in the above definition of $\lambda_{\boldsymbol{h}}$ with the function $\text{softplus}_\tau(x) = \tau \log(1 + \exp(x/\tau)) > 0$. We learn a separate temporal scale parameter τ for each functor and use the one associated with the functor of $\boldsymbol{h}$.

4.4.3 Updates Through Time

We now add an LSTM-like component so that each atom will track the sequence of events that it has "seen"—that is, the sequence of events that updated it via <- rules (section 4.3.3). Recall that an LSTM is constructed from **memory cells** that can be increased or decreased as successive inputs arrive.

Every atom $\boldsymbol{h}$ has a **cell block** $\boxed{\boldsymbol{h}} \in \mathbb{R}^{D_{\boldsymbol{h}}} \cup \{\mathsf{null}\}$. When $\boxed{\boldsymbol{h}} \neq \mathsf{null}$, we augment $\boldsymbol{h}$'s embedding formula equation (4.3) to[9]

$$[\![\boldsymbol{h}]\!] \stackrel{\text{def}}{=} \tanh\left(\boxed{\boldsymbol{h}} + \sum_r [\boldsymbol{h}]_r^{:-}\right) \in (-1, 1)^{D_{\boldsymbol{h}}} \tag{4.7}$$

Properly speaking, $[\![\boldsymbol{h}]\!]$, $\boxed{\boldsymbol{h}}$, and $[\boldsymbol{h}]_r^{:-}$ are all functions of t.

At times when $\boxed{\boldsymbol{h}} = \mathsf{null}$, we like to say that $\boldsymbol{h}$ is **docked**. Every atom $\boldsymbol{h}$ is docked initially (at $t = 0$), but may be **launched** through an update of type equation (4.2a), which ensures that $\boxed{\boldsymbol{h}} \neq \mathsf{null}$ and thus $[\![\boldsymbol{h}]\!] \neq \mathsf{null}$ by equation (4.7).

[9] Recall from section 4.4.2 that if $\boldsymbol{h}$ is an event, we extend $[\![\boldsymbol{h}]\!]$ with an extra dimension to carry the probability. For equation (4.7) to work, we must likewise extend $\boxed{\boldsymbol{h}}$ with an extra cell (when $\boxed{\boldsymbol{h}} \neq \mathsf{null}$).

h is subsequently **adrift** (and remains a fact) until it is docked again through an update of type equation (4.2b), which sets $\boxed{h} = \mathsf{null}$.

How is $\boxed{h}$ updated by an event (or events[10]) occurring at time s? Suppose the r^{th} rule is an update rule of type equation (4.2a). Consider its instantiations $h \leftarrow e, g_1, \ldots, g_N$ (if any) with head h, such that e occurred at time s and $g_1, \ldots, g_N$ are all facts at time s. For the m^{th} instantiation, define

$$[h]^{\leftarrow}_{rm} \stackrel{\text{def}}{=} \mathbf{W}_r \underbrace{[1; [\![e]\!]; [\![g_1]\!]; \ldots; [\![g_N]\!]]}_{\text{concatenation of column vectors}} \tag{4.8}$$

where all embeddings are evaluated at time s, and $\mathbf{W}_r$ is again a conformable matrix associated with the r^{th} rule. We now explain how to convert $[h]^{\leftarrow}_{rm}$ to an **update vector** $[h]^{\Delta}_{rm}$, and how all update vectors combine to modify $\boxed{h}$.

How is $[h]^{\Delta}_{rm}$ obtained? In an ordinary LSTM (Hochreiter and Schmidhuber, 1997), a cell block $\boxed{h}$ is updated by

$$\boxed{h}_{\text{new}} = \mathbf{f} \cdot \boxed{h}_{\text{old}} + \mathbf{i} \cdot (2\mathbf{z} - 1) \tag{4.9}$$

corresponding to an increment

$$\boxed{h} \;\mathrel{+}=\; (\mathbf{f} - 1) \cdot \boxed{h} + \mathbf{i} \cdot (2\mathbf{z} - 1) \tag{4.10}$$

where the forget gates $\mathbf{f}$, input gates $\mathbf{i}$, and inputs $\mathbf{z}$ are all in $(0, 1)^{D_h}$. Thus, we define $[h]^{\Delta}_{rm}$ as the right side of equation (4.10) when $(\mathbf{f}; \mathbf{i}; \mathbf{z}) \stackrel{\text{def}}{=} \sigma([h]^{\leftarrow}_{rm})$, with $[h]^{\leftarrow}_{rm} \in \mathbb{R}^{3D_h}$ from equation (4.8).

Discrete-Time Setting. Here we treat the update vectors $[h]^{\Delta}_{rm}$ as increments to $\boxed{h}$. To update $\boxed{h}$ from time s to time $t = s + 1$, we pool these increments within

[10] If exogeneous events are used (section 4.3.4), then the instantiations in equation (4.8) could include multiple events e that occurred at time s.

and across rules (much as in equation (4.3)–equation (4.4)) and increment by the result:

$$\boxed{h} \;\mathrel{+}=\; \sum_r \bigoplus_m^{\beta_r} [h]_{rm}^{\Delta} \tag{4.11}$$

We skip the update equation (4.11) if h has no update vectors. If we apply equation (4.11), we first set $\boxed{h}$ to $\mathbf{0}$ if it is null at time s, or has just been set to null at time s by a equation (4.2b) rule (docking).

A small difference from a standard LSTM is that our updated cell values $\boxed{h}$ are transformed into equally many output values $[\![h]\!]$ via equation (4.7), instead of through $\tanh$ and output gates. A more important difference is that in a standard LSTM, the model's state is a single large cell block. The state update when new input arrives depends on the entire current state. Our innovation is that the update to $\boxed{h}$ (a *portion* of the model state) depends on only a relevant *portion* of the current state, namely $[[\![e]\!]; [\![g_1]\!]; \ldots; [\![g_N]\!]]$. If there are many choices of this portion, equation (4.11) pools their effects across instantiations and sums them across rules.

Continuous-Time Setting. Here we use the continuous-time LSTM as defined in Chapter 3, in which cells **drift** between updates to record the passage of time. Each cell drifts according to some parametric function. We will update a cell's parameters just at times when a *relevant* event happens. A fact's embedding $[\![h]\!](t)$ at time t is still given by equation (4.7), but $\boxed{h}(t)$ in that equation is given by $\boxed{h}$'s parametric functions as most recently updated (at some earlier time $s < t$). section 4.C reviews the simple family of parametric functions used in the continuous-time LSTM, and specifies how we update the parameters using a collection of update vectors $[h]_{rm}^{\Delta}$ obtained from the $[h]_{rm}^{\leftarrow}$.

Remark. It is common for event atoms `e` to have $D_e = 0$. Then they still have time-varying probabilities (section 4.4.2)—often via `:-` rules whose conditions have time-varying embeddings—but have no embeddings. Even so, different events will result in different updates. This is thanks to Datalog's pattern matching: the event's atom `e` controls which update rules `head <- event, condits...` it triggers, and with what head and condition atoms (since variables in `event` are *reused* elsewhere in the rule). The update to the head atom then depends on the parameters of the selected rules and the current embeddings of their condition atoms.

4.5 Training and Inference

For training and inference, we follow the general recipes in Chapter 2. Recall that the model likelihood will involve a summation (at each time t) over the finite set of event types that are possible at time t. In the previous chapters, this set is assumed to be constant, i.e., $\mathcal{E}(t) = \{1, \ldots, K\}$. In the neural Datalog through time, this set $\mathcal{E}(t)$ may change over time, since the possible event types at any time are given by facts, which may be updated in response to stochastic events.

Note that our approach will never predict an impossible event type. For example, `help(eve,adam)` won't be in $\mathcal{E}(t)$ and thus will have *zero* probability if $[\![\texttt{rel(eve,adam)}]\!](t) = \mathsf{null}$ (maybe because `eve` stops having `opinion`s on anything that `adam` does anymore).

4.6 Related Work

Past work Sato, 1995; Poole, 2010; Richardson and Domingos, 2006; Raedt, Kimmig, and Toivonen, 2007; Bárány et al., 2017 has used logic programs to help define probabilistic relational models Getoor and Taskar, 2007. These models do not make

use of vector-space embeddings or neural networks. Nor do they usually have a temporal component. However, some other (directed) graphical model formalisms do allow the model architecture to be affected by data generated at earlier steps Minka and Winn, 2008; Meent et al., 2018.

Our "neural Datalog through time" framework uses a deductive database augmented with update rules to define and dynamically reconfigure the architecture of a neural generative model. Conditional neural net structure has been used for natural language—e.g., conditioning a neural architecture on a given syntax tree or string (Andreas et al., 2016; Lin et al., 2019). Also relevant are neural architectures that use external read-write memory to achieve coherent sequential generation, i.e., their decisions are conditioned on a possibly symbolic record of data generated from the model at earlier steps (Graves, Wayne, and Danihelka, 2014; Graves et al., 2016; Weston, Chopra, and Bordes, 2015; Sukhbaatar, Weston, Fergus, et al., 2015; Kumar et al., 2016a; Kiddon, Zettlemoyer, and Choi, 2016; Dyer et al., 2016; Lample et al., 2019; Xiao, Teichmann, and Arkoudas, 2019). We generalize some such approaches by providing a logic-based specification language.

Many papers have presented domain-specific sequential neural architectures (Natarajan et al., 2008; Heijden, Velikova, and Lucas, 2014; Shelton and Ciardo, 2014; Meek, 2014; Bhattacharjya, Subramanian, and Gao, 2018; Wang et al., 2019). The models closest to ours are **Know-Evolve** (Trivedi et al., 2017) and **DyRep** (Trivedi et al., 2019), which exploit explicit domain knowledge about how structured events depend on and modify the neural states of their participants. DyRep also conditions event probabilities on a temporal graph encoding binary relations among a fixed set of entities. In section 4.7, we will demonstrate that fairly simple programs in

our framework can substantially outperform these strong competitors by leveraging even richer types of knowledge, e.g.: ① Complex n-ary relations among entities that are constructed by join, disjunction, and recursion (section 4.3.1) and have derived embeddings (section 4.3.2). ② Updates to the set of possible events (section 4.3.5). ③ Embeddings of entities and relations that reflect selected past events (sections 4.3.4 and 4.3.6).

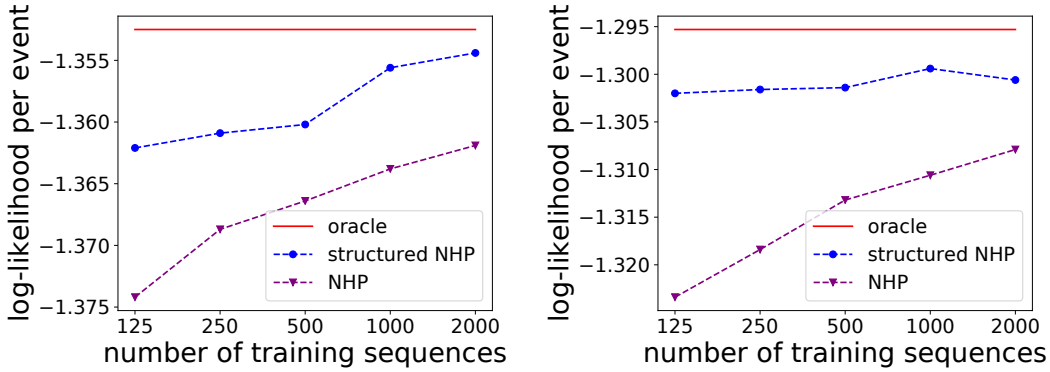
4.7 Experiments

In several continuous-time domains, we exhibit informed models specified using neural Datalog through time (NDTT). We evaluate these models on their held-out log-likelihood, and on their success at predicting the time and type of the next event. We compare with the unrestricted neural Hawkes process (NHP) and with Know-Evolve (KE) and DyRep.

We implemented our NDTT framework using PyTorch (Paszke et al., 2017) and pyDatalog (Carbonell et al., 2016). We then used it to implement our individual models—and to reimplement all three baselines, after discussion with their authors, to ensure a controlled comparison. Our code and datasets are available at the URL given in section 4.3. Experimental details are given in the appendices of Mei et al. (2020).

4.7.1 Synthetic Superposition Domain

The activities of strangers rarely influence each other, even if they are all observed within a single sequence. We synthesized a domain where each sequence is a superposition of data drawn from M different processes that do not interact with one another at all. Each process generates events of N types, so there are MN total event types `e(M,N)`.



(a) $M = 8$ **(b)** $M = 16$

Figure 4.2: Learning curves of structured model ● and NHP ▼, on sequences drawn from the structured model. The former is significantly better at each training size ($p < 0.01$, paired permutation test).

```
is_process(1).
⋮
is_process(M).
```

```
is_type(1).
⋮
is_type(N).
```

The baseline model is a neural Hawkes process (NHP). It assigns to each event type a separate embedding[11]

```
:- embed(is_event, 8).
is_event(1,1) :- is_process(1), is_type(1).
is_event(1,2) :- is_process(1), is_type(2).
  ⋮
```

This unrestricted model allows all event types to influence one another by depending on and affecting a `world` state:

```
:- event(e, 0).
:- embed(world, 8).
e(M,N) :- world, is_process(M), is_type(N).
world <- init.
world <- e(M,N), is_event(M,N), world.
```

[11] The list of facts like lines 8 and 9 can be replaced by a single rule if we use "parameter names" as explained in section 4.B.

Note that e(M,N) in line 14 has no embedding, since any such embedding would vary along with the probability. As explained in section 4.4.3, line 14 instead uses e(M,N) to draw in the embedding of is_event(M,N), which does not depend on world so is static, as called for by the standard NHP.

To obtain a *structured* NHP that recognizes that events from different processes cannot influence each other, we replace world with multiple local states: each e(M,N) only interacts with local(M). Replace lines 11–14 with

```
15  :- embed(local, 8).
16  e(M,N) :- local(M), is_type(N).
17  local(M) <- init, is_process(M).
18  local(M) <- e(M,N), is_event(M,N), local(M).
```

For various small N and M values (see section 4.F.2), we randomly set the parameters of the structured NHP model and draw training and test sequences from this distribution. We then generated learning curves by training the correclty structured model versus the standard NHP on increasingly long prefixes of the training set, and evaluating them on held-out data. Figure 4.2 shows that although NHP gradually improves its performance as more training sequences become available, the structured model unsurprisingly learns faster, e.g., only 1/16 as much training data to achieve a higher likelihood. In short, it helps to use domain knowledge of which events come from which processes.

4.7.2 Real-World Domains: IPTV and RoboCup

IPTV Domain (Xu, Luo, and Carin, 2018)**.** This dataset contains records of 1000 users watching 49 TV programs over the first 11 months of 2012. Each event has the form watch(U,P). Given each prefix of the test event sequence, we attempted to predict the next test event's time t, and to predict its program P given its actual time t

and user `U`.

We exploit two types of structural knowledge in this domain. First, each program `P` has (exactly) 5 out of 22 genre tags such as `action`, `comedy`, `romance`, etc. We encode these as known static facts `has_tag(P,T)`. We allow each tag's embedding ⟦`tag(T)`⟧ to not only influence the embedding of its programs (line 1) but also track which users have recently watched programs with that tag (line 2):

```
1  program(P) :- has_tag(P,T), tag(T).
2  tag(T) <- watch(U,P), has_tag(P,T).
```

As a result, a program's embedding ⟦`program(P)`⟧ changes over time as its tags shift in meaning.

Second, there is a dynamic hard constraint that a program cannot be watched until it is released, since only then is it added to the database:

```
3  program(P) <- release(P).
4  watch(U,P) :- user(U), program(P).
```

Here `release(P)` is an exogenous event with no embedding. More details can be found in section 4.F.3, including full NDTT programs that specify the architectures used by the KE and DyRep papers and by our model.

RoboCup Domain (Chen and Mooney, 2008)**.** This dataset logs actions of soccer players such as `kick(P)` and `pass(P,Q)` during RoboCup Finals 2001–2004. There are 528 event types in total. For each history, we made minimum Bayes risk predictions of the next event's time, and of that event's participant(s) given its time and action type.

Database facts change frequently in this domain. The ball is transferred between robot players at a high rate:

```
1  !has_ball(P) <- pass(P,Q). % ball passed from P
```

```
has_ball(Q)  <- pass(P,Q). % ball passed to Q
```

which leads to highly dynamic constraints on the possible events (since only the ball possessor can kick or pass):

```
pass(P,Q) :- has_ball(P), teammate(P,Q), ...
```

This example also illustrates how relations between players affect events: the ball can only be passed to a teammate. Similarly, only an opponent may steal the ball:

```
steal(Q,P) :- has_ball(P), opponent(P,Q), ...
```

We allow each event to update the states of involved players as both KE and DyRep do. We further allow the event observers such as the entire team to be affected as well:

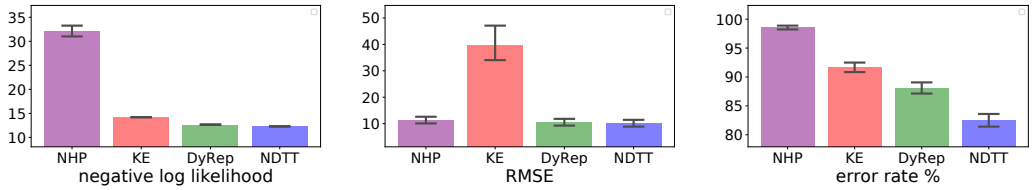
```
team(T) <- pass(P,Q), in_team(P,T), ....
```

so all players can be aware of this event by consulting their team states. More details can be found in section 4.F.5, including our full Datalog programs. The hard logical constraints on possible events are not found in past models.

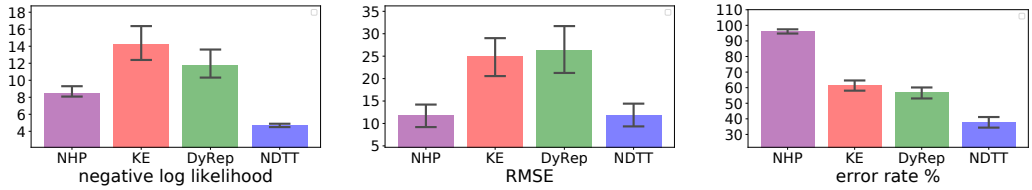
4.7.2.1 Prediction Results

After training, we used minimum Bayes risk (section 2.4) to predict events in test data. Figure 4.3 shows that our NDTT model enjoys consistently lower error than strong competitors, across datasets and prediction tasks. NHP performs poorly in general since it doesn't consider any knowledge. KE handles relational information, but doesn't accommodate dynamic facts such as `released(game_of_thrones)` and `has_ball(a8)` that reconfigure model architectures on the fly.

In the IPTV domain, DyRep handles dynamic facts (e.g., newly released programs) and thus substantially outperforms KE. Our NDTT model's moderate further improvement results from its richer :- and <- rules related to tags.



(a) IPTV Dataset



(b) RoboCup Dataset

Figure 4.3: Evaluation results with 95% bootstrap confidence intervals on the real-world datasets of our Datalog program vs. the neural Hawkes process (NHP), KnowEvolve (KE) and DyRep. The RMSE is the root of mean squared error for predicted time. Error rate % denotes the fraction of incorrect predictions of the watched TV program (in IPTV) or the specific player (in RoboCup), given the event time.

In the RoboCup domain, our reimplementation of DyRep allows deletion of facts (player losing ball possession), whereas the original DyRep only allowed addition of facts. Even with this improvement, it performs much worse than our full NDTT model. To understand why, we carried out further ablation studies, finding that NDTT benefits from its hybridization of logic and neural networks.

4.7.2.2 Ablation Study I: Taking Away Logic.

In the RoboCup domain, we investigated how the model performance degrades if we remove each kind of rule from the NDTT model. We obtained "NDTT−" by dropping the `team` states, and "DyRep++" by not tracking the ball possessor. The latter is still an enhancement to DyRep because it adds useful ← rules: the first "+" stands for the ← rules in which some conditions are not neighbors of the head, and the second "+" stands for the ← rules that update event observers.

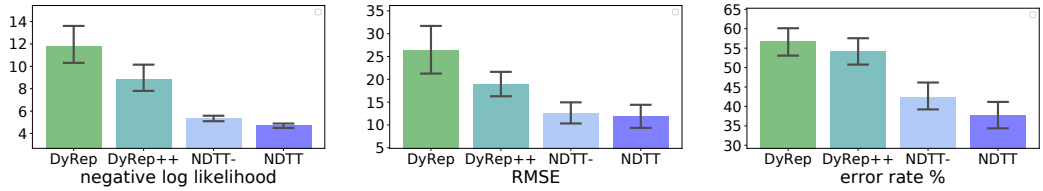
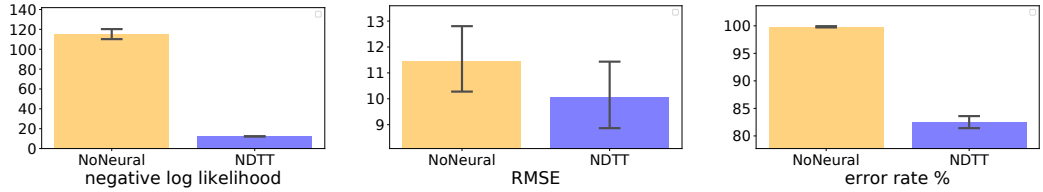
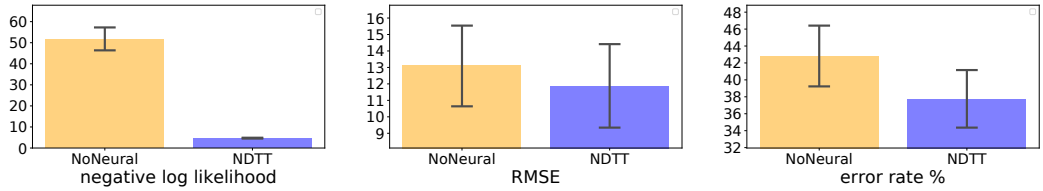


Figure 4.4: Ablation study in the RoboCup domain. "DyRep++" has the same <- rules as our structured model and "NDTT−" uses 0-dimensional `team` embeddings.



(a) IPTV Dataset



(b) RoboCup Dataset

Figure 4.5: Ablation study of taking away neural networks from our Datalog programs in the real-world domains. The format of the graphs is the same as in Figure 4.3. The results imply that neural networks have been learning useful representations that are not explicitly specified in the Datalog programs.

As Figure 4.4 shows, both ablated models outperform DyRep but underperform our full NDTT model. DyRep++ is interestingly close to NDTT on the participant prediction, implying that its neural states learn to track who possesses the ball—though such knowledge is not tracked in the logical database—thanks to rich <- rules that see past events.

4.7.2.3 Ablation Study II: Taking Away Neural Networks.

We also investigated how the performance of our structured model would change if we reduce the dimension of all embeddings to zero. The model still knows logically which events are possible, but events of the same type are now more interchangeable. As shown in Figure 4.5, the performance turns out to degrade greatly, indicating that the neural networks had been learning representations that are actually helpful for probabilistic modeling and prediction.

Note that each event type still has an extra dimension for its intensity (see section 4.4.2). However, the intensity of each possible event now depends only on *which* rules proved or updated that possible event (through the bias terms of those rules); it no longer depends on the embeddings of the specific atoms on the right-hand-sides of those rules. Two events may nonetheless have different intensities if they were proved by different :- rules, or proved or updated by different sequences of <- rules (where the difference may be in the identity of the <- rules or in their timing).

4.8 Conclusion

In this chapter, we showed how to specify a neural-symbolic probabilistic model simply by writing down the rules of a deductive database. "Neural Datalog" makes it simple to define a large set of structured objects ("facts") and equip them with embeddings and probabilities, using pattern-matching rules to explicitly specify which objects depend on one another.

To handle temporal data, we proposed an extended notation to support *temporal* deductive databases. "Neural Datalog through time" allows the facts, embeddings, and probabilities to change over time, both by gradual drift and in response to discrete

events. We demonstrated the effectiveness of our framework by generatively modeling event sequences in real-world domains.

Appendices

4.A Extensions to the Formalism

In this appendix, we consider possible extensions to our formalism. These illuminate interesting issues, and the extensions are compatible with our overall approach to modeling. Some of these extensions are already supported in our implementation at https://github.com/HMEIatJHU/neural-datalog-through-time, and more of them may be supported in future versions.

4.A.1 Cyclicity

Our embedding definitions in sections 4.3.2 and 4.4.1 assumed that the proof graph was acyclic. However, it is possible in general Datalog programs for a fact to participate in some of its own proofs.

For example, the following classical Datalog program finds the nodes in a directed graph that are reachable from the node `start`:

```
reachable(start).
reachable(V) :- reachable(U), edge(U,V).
```

In neural Datalog, the embedding of each fact of the form `reachable(V)` depends on all paths from `start` to `V`. However, if `V` appears on a cycle in the directed graph defined by the `edge` facts, then there will be infinitely many such paths, and our definition of $[\![$`reachable(V)`$]\!]$ would then be circular.

Restricting to acyclic proofs. One could define embeddings and probabilities in a cyclic proof graph by considering only the acyclic proofs of each atom h. This is expensive in the worst case, because it can exponentially increase the number of embeddings and probabilities that need to be computed. Specifically, if S is a (finite) set of atoms, let $[\![h/S]\!]$ denote the embedding constructed from acyclic proofs of h that do not use any of the atoms in the finite set S. We define $[\![h/S]\!]$ to be null if $h \in S$, and otherwise to be defined similarly to $[\![h]\!]$ but where equations (4.4) and (4.8) are modified to replace each $[\![g_i]\!]$ with $[\![g_i/(S \cup \{h\})]\!]$.[12] As usual, these formulas skip pooling over instantiations where any $[\![\cdot]\!]$ values in the body are null. The recursive definition terminates because S grows at each recursive step but its size is bounded above (section 4.3.7).

In particular, this scheme defines $[\![h/\emptyset]\!]$, the **acyclic embedding** of h, which we consider to be an output of the neural Datalog program. Similarly, in neural Datalog through time, the probability of an event e is derived from $\lambda_{e/\emptyset}$, which is computed in the usual way (section 4.4.2) as an extra dimension of the acyclic embedding $[\![e/\emptyset]\!]$.

Forward propagation. This is a more practical approach, used by Hamilton, Ying, and Leskovec (2017a) to embed the vertices of a graph. This method recomputes all embeddings in parallel, and repeats this for some number of iterations. In our case, for a given time t, each $[\![h]\!]$ is initialized to $\mathbf{0}$, and at each iteration it is recomputed via the formulas of sections 4.4.1 and 4.4.3, using the $[\![g_i]\!]$ values from the previous iteration (also at time t) and the cell block $\boxed{h}$ (determined by events at times $s < t$).

[12] For increased efficiency, one can simplify $S \cup \{h\}$ here to eliminate atoms that can be shown by static analysis or depth-first search not to appear in any proof of g_i. This allows more reuse of previously computed $[\![\cdot]\!]$ terms and can sometimes prevent exponential blowup. In particular, if it can be shown that all proofs of h are acyclic, then $[\![h/S]\!]$ can always be simplified to $[\![h/\emptyset]\!]$ and the computation of $[\![h/\emptyset]\!]$ is isomorphic to the ordinary computation of $[\![h]\!]$; the algorithm then reduces to the ordinary algorithm from the main chapter.

We suggest the following variant that takes the graph structure into account. At time t, construct the (finite) Datalog proof graph, whose nodes are the facts at time t. Visit its strongly connected components in topologically sorted order. Within each strongly connected component C, initialize the embeddings to $\mathbf{0}$ and then recompute them in parallel for $|C|$ iterations. If the graph is acyclic, so that each component C consists of a single vertex, then the algorithm reduces to an efficient and exact implementation of sections 4.4.1 and 4.4.3. In the general case, visiting the components in topologically sorted order means that we wait to work on component C until its strictly upstream nodes have "converged," so that the limited iterations on C make use of the best available embeddings of the upstream nodes. By choosing $|C|$ iterations for component C, we ensure that all nodes in C have a chance to communicate: information has the opportunity to flow end-to-end through all cyclic or acyclic paths of length $< |C|$, and this is enough to include all acyclic paths within C. Note that the embeddings computed by this algorithm (or by the simpler method of Hamilton, Ying, and Leskovec (2017a)) are well-defined: they depend only on the graph structure, not on any arbitrary ordering of the computations.

4.A.2 Negation in Conditions

A simple extension to our formalism would allow negation in the body of a rule (i.e., the part of the rule to the right of `:-` or `<-`). In rules of the form equation (4.1) or equation (4.2), each of the conditions $\texttt{condit}_i$ could optionally be preceded by the negation symbol `!`. In general, a rule only applies when the ordinary conditions are true and the negated conditions are false. The concatenation of column vectors in equations (4.4) and (4.8) omits $[\![g_i]\!]$ if $\texttt{condit}_i$ is negated, since then g_i is not a fact and does not have a vector (rather, $[\![g_i]\!] = \mathsf{null}$).

Many dialects of Datalog permit programs with negation. If we allow cycles (section 4.A.1), we would impose the usual restriction that negation may not appear on cycles, i.e., programs may use only **stratified negation**. This restriction ensures that the set of facts is well-defined, by excluding rules like `paradox :- !paradox.`

Example. Extending our example of section 4.3, we might say that a person can eventually grow up into an adult and acquire a gender. Whether person `X` grows up into (say) a woman, and the time at which this happens, depends on the probability or intensity (section 4.4.2) of the `growup(X,female)` event. We use negation to say that a `growup` event can happen only once to a person—after that, all `growup` events for that person become false atoms (have probability 0).

```
24  adult(X,G) <- growup(X,G).
25  adult(X) :- adult(X,G).
26  growup(X,G) :- person(X), gender(G), !adult(X).
27  gender(female).
28  gender(male).
29  gender(nonbinary).
      ⋮
```

As a result, an adult has exactly one gender, chosen stochastically. Female and male adults who know each other can procreate:

```
30  procreate(X,Y) :- rel(X,Y),
      adult(X,female), adult(Y,male).
```

4.A.3 Highway Connections

As convenient "syntactic sugar," we introduce a variant `:=` of the `:-` connector. The extra horizontal line introduces extra **highway connections** that skip a level in the neural network. A fact's embedding can now be directly affected by its grandparents in the proof DAG, not just its parents. This does not change the set of facts that are proved.

Highway connections of roughly this sort have been argued to help neural network training by providing shorter, more direct paths for backpropagation Srivastava, Greff, and Schmidhuber, 2015. They also increase the number of parameters in the model.

We use an example to show how they are specified in neural Datalog. Consider the following := rules. The first rule replaces line 4 from section 4.3 with a := version. The second rule is added to make the example more interesting. It uses a high-dimensional `teacher` embedding that represents the academic relationship between `X` and `Y` (which is presumably updated by every academic interaction between them).

```
31  rel(X,Y) := opinion(X,U), opinion(Y,U).
32  rel(X,Y) := teacher(X,Y).
```

The embeddings of `rel` facts are computed as before. However, the := rules in the definition of `rel` affect the interpretation of the other :-, :=, and <- rules in the program whose body contains `rel`. A simple example of such a rule is line 14 from section 4.3.5:

```
33  help(X,Y) :- rel(X,Y).
```

The following rules are now *automatically added to the program*:

```
34  help(X,Y) :- opinion(X,U), opinion(Y,U).
35  help(X,Y) :- teacher(X,Y).
```

As a result, an embedding such as ⟦`help(eve,adam)`⟧ is defined using *not only* ⟦`rel(eve,adam)`⟧, but *also* the embeddings of any lower-level facts that proved `rel(eve,adam)` via the := lines 31 and 32.

In the simple case where `rel(eve,adam)` has only one proof, this scheme is equivalent to augmenting ⟦`rel(eve,adam)`⟧ by concatenating it with the embeddings of its parent or parents. This higher-dimensional version of ⟦`rel(eve,adam)`⟧ now participates as usual in the computation of other embeddings such as ⟦`help(eve,adam)`⟧.

However, notice that the dimensionality of the augmented ⟦rel(eve,adam)⟧ will differ according to whether rel(eve,adam) was proved via line 31 or line 32. Therefore, different parameters must be used for the additional dimensions, associated with line 34 or line 35 respectively.

More generally, notice that ⟦help(eve,adam)⟧ will *sum* over the contributions from the two lines 34 and 35 (via equation (4.3) or equation (4.7)). The former contribution may itself involve *pooling* (via equation (4.4)) over all topics U about which eve and adam both have opinions. This pooling is performed separately from the pooling over U used in line 31: in particular, it may use a different β parameter.

Of course, the definition of rel may also include *non-highway* rules such as

```
36  rel(X,Y) :- married(X,U).
37  rel(X,Y) <- hire(X,Y).
```

Since line 33 is still in the program, however, proving rel(eve,adam) remains sufficient to prove the possible event help(eve,adam) even when rel(eve,adam) is proved by non-highway rules.

Longer highways can be created by chaining multiple := rules together. For example, if we replace line 33 with a := version,

```
38  help(X,Y) := rel(X,Y).
```

then lines 34–35 will also use :=. Hence, any rule whose body uses help will automatically acquire versions that mention rel, opinion, and teacher (by repeating the bodies of lines 33–35 respectively).

There are several subtleties in our highway program transformation:

The additional lines 34–35 were constructed by expanding ("inlining") the call to rel within the body of line 33. In logic programming, the inlining transformation

is known as **unfolding**. In general it may involve unification, as well as variable renaming to avoid capture.

When we unfold a rule condition, the original condition is usually deleted from the new (unfolded) version of the rule, since it is now redundant. However, the event that triggers an update rule cannot be deleted in this way. Consider the update line 11 from section 4.3.3:

```
39  grateful(Y,X) <- help(X,Y), person(Y).
```

Suppose `help(X,Y)` is defined using the highway line 38. The rule that we automatically add cannot be

```
40  grateful(Y,X) <- rel(X,Y), person(Y).
```

as one might expect, because `rel(X,Y)` is not even an event that can be used in this position. Instead, we must ensure that the event is still triggered by the original event:

```
41  grateful(Y,X) <- help(X,Y) : 0, rel(X,Y), person(Y) : 0.
```

As explained in section 4.B below, the `: 0` notation says that although the highway line 41 is *triggered* by the event `help(X,Y)`, it ignores the event's *embedding*. After all, the event's embedding is still considered by the original line 39 and does not need to be considered again. The contributions of these two rules will be summed by equation (4.3) or equation (4.7) before $\tanh$ is applied.

The above example also illustrates the handling of rule conditions that are *not* unfolded, such as `person`. The unfolded rule (e.g., line 41) marks these conditions with `: 0` as well, to say that while they are still boolean conditions on the update, their embeddings should also be ignored. Again, their embeddings are considered in the original line 39, so they do not need to be considered again.

Finally, notice that a rule body may contain multiple events and/or conditions that

are defined using highway rules. How do we expand

```
1  world <- e, f, g.
```

given the following highway definitions?

```
2  e := e1.
3  e := e2.
4  g := g1.
5  g := g2.
```

The general answer is that we unfold each of the body elements in parallel, to allow highway connections from that element. In this case we add 4 new rules:

```
6  world <- e : 0, e1, f : 0, g : 0.
7  world <- e : 0, e2, f : 0, g : 0.
8  world <- e : 0,     f : 0, g1.
9  world <- e : 0,     f : 0, g2.
```

4.A.4 Infinite Domains

section 4.3.7 explained that under our current formalism, any given model only allows a finite set of atoms. Thus, it is not possible for new persons to be born.

One way to accommodate that might be to relax Datalog's restriction on nesting.[13] This allows us to build up an infinite set of atoms from a finite set of initial entities:

```
42  birth(X,Y,child(X,Y)) <- procreate(X,Y).
```

Therefore, each new person would be named by a tree giving their ancestry, e.g., `child(eve,adam)` or `child(awan,child(eve,adam))`. But while this method may be useful in other settings, it unfortunately does not allow `eve` and `adam` to have *multiple* children.

[13] To be safe, we should allow *only* the <- rules (which are novel in our formalism) to derive new facts with greater nesting depth than the facts that appear in the body of the rule. This means that the nesting depth of the database may increase over time, by a finite amount each time an event happens. If we allowed that in traditional :- rules, for example `peano(s(X)) :- peano(X)`, then we could get an infinite set of facts at an *single* time. But then computation at that time might not terminate, and our $\bigoplus^{\beta}$ operators might have to aggregate over infinite sets (see section 4.3.7).

Instead, we suggest a different extension, which allows events to create new *anonymous* entities (rather than nested terms):

```
43  birth(X,Y,*) <- procreate(X,Y).
```

The special symbol $*$ denotes a new entity that is created during the update, in this case representing the child being born. Thus, the event `procreate(eve,adam)` will launch the fact `birth(eve,adam,cain)`, where `cain` is some internal name that the system assigns to the new entity. In the usual way when launching a fact, the cell block `birth(eve,adam,cain)` is updated from an initial value of $\mathbf{0}$ by equation (4.9) in a way that depends on $[\![$`procreate(eve,adam)`$]\!]$.

From the new fact `birth(eve,adam,cain)`, additional rules derive further facts, stating that `cain` is a person and has two parents:[14]

```
44  person(Z) :- birth(X,Y,Z).
45  parent(X,Z) :- birth(X,Y,Z).
46  parent(Y,Z) :- birth(X,Y,Z).
```

Notice that the embedding $[\![$`person(cain)`$]\!]$ initially depends on the state of his parents and their relationship at the time of his procreation. This is because it depends on $[\![$`birth(eve,adam,cain)`$]\!]$ which depends through its cell block on $[\![$`procreate(eve,adam)`$]\!]$, as noted above. $[\![$`person(cain)`$]\!]$ may be subsequently updated over time by events such as `help(eve,cain)`, which affect its cell block.

As another example, here is a description of a sequence of orders in a restaurant:

```
1  :- embed(dish, 5).
2  :- event(order, 0).
3  order(X) :- dish(X).
```

[14] Somewhat awkwardly, under our design, line 23 is not enough to remove `person(cain)` from the database, since that fact was established by a `:-` rule. We actually have to write a rule canceling `cain`'s birth: `!birth(X,Y,Z) <- die(Z).` Notice that this rule will remove not only `person(cain)` but also `parent(eve,cain)` and `parent(adam,cain)`. Even then, the entity `cain` may still be referenced in the database as a `parent` of his own children, until they die as well.

```
4  order(*).
5  dish(X) <- order(X).
```

This program says that the possible orders consist of any existing dish or a new dish. When used in the discrete-time setting, this model is similar to the Chinese restaurant process (CRP) (Aldous et al., 1985). Just as in the CRP,

- The relative probability of ordering a new dish at time $s \in \mathbb{N}$ is a (learned) constant (because line 4 has no conditions).
- The relative probability of each possible order(X) event, where X is an existing dish, depends on the embedding of dish(X) (line 3). That embedding reflects only the number of times X has been ordered previously (line 5), though its (learned) dependence on that number does not have to be linear as in the CRP.

Interestingly, in the continous-time case—or if we added a rule dish(X) <- tick that causes an update at every discrete time step (see section 4.A.5 below)—the relative probability of the order(X) event would also be affected by the time intervals between previous orders of X. It is also easy to modify this program to get variant processes in which the relative probability of X is also affected by previous orders of dishes Y $\neq$ X (cf. Blei and Lafferty, 2006) or by the exogenous events at the present time and at times when X was ordered previously (cf. Blei and Frazier, 2010).

section 4.A.6 below discusses how an event may trigger an unbounded number of dependent events that provide details about it. This could be used in conjunction with the $*$ feature to create a whole tree of facts that describe a new anonymous entity.

4.A.5 Uses of Exogenous Events

The extension to allow exogeneous events was already discussed in the main chapter (section 4.3.4). Here we mention two specific uses in the discrete-time case.

It is useful in the discrete-time case to provide an exogenous `tick` event at *every* $s \in \mathbb{N}$. (Note that this results in a second event at every time step; see footnote 10.) Any cell blocks that are updated by the exogenous `tick` events will be updated even at time steps s between the modeled events that affect those cell blocks. For example, one can write a rule such as `person(X) <- tick, person(X), world.` so that persons continue to evolve even when nothing is happening to them. This is similar to the way that in the continous-time case, cell blocks with $\delta \neq 0$ will drift via equation (4.11) during the intervals between the modeled events that affect those cell blocks.[15]

Another good use of exogenous events in discrete time is to build a conditional probability model such as a word sequence tagger. At every time step s, a word occurs as an exogenous event, at the same time that the model generates an tag event that supplies a tag for the word at the previous time step. These two events at time s *together* update the state of the model to determine the distribution over the next tag at time $t = s + 1$. Notice that the influences of the word and the tag on the update vector are summed (by the $\sum_r$ in equation (4.11)). This architecture is similar to a left-to-right LSTM tagger (cf. Ling et al., 2015; Tran et al., 2016).

4.A.6 Modeling Multiple Simultaneous Events

Section 4.4.2 explained how to model a discrete-time event sequence:

[15] In fact, tick events can *also* be used in the continuous case, if desired (Mei and Eisner, 2017). Then the drifting cells not only drift, but also undergo periodic learned updates that may depend on other facts (as specified by the `tick` update rules).

> To model a discrete-time event sequence, define the **probability** of an event of type h at time step t to be proportional to $\lambda_e(t)$, normalizing over all event types that are possible then.

In such a sequence, *exactly* one event is generated at each time t. To change this to "*at most* one event," an additional event type `none` can be used to encode "nothing occurred."

Our continuous-time models are also appropriate for data in which *at most* one event occurs at each time t, since almost surely, there are no times t with multiple events. Recall from section 4.4.2 that in this setting, the expected number of occurrences of e on the interval $[t, t + dt)$, divided by dt, approaches $\lambda_e(t)$ as $dt \to 0^+$. Thus, given a time t at which one event occurs, the expected total number of *other* events on $[t, t + dt)$ approaches 0 as $dt \to 0^+$.

However, there exist datasets in which multiple events do occur at time t—even multiple copies of the same event. By extending our formalism with a notion of **dependent events**, we can model such datasets generatively. The idea is that an event e at time t can stochastically generate dependent events that also occur at time t.

(When multiple events occur at time t, our model already specifies how to handle the <- rule updates that result from these events. Specifically, multiple events that simultaneously update the same head are pooled within and across rules by equation (4.11).)

To model the events that depend on e, we introduce the notion of an **event group**, which represents a group of competing events at a particular instant. Groups do not persist over time; they appear momentarily in response to particular events. If event e at time t **triggers** group g and g is non-empty at time t, then exactly one event e' in

g (perhaps none) will stochastically occur at time t as well.

Under some programs, it will be possible for multiple copies—that is, **tokens**—of the *same* event type to occur at the same time. For precision, we use e below for a particular event token at a particular time, using $\bar{e}$ to denote the Datalog atom that names its event type. Similarly, we use g for a particular token of a triggered group, using $\bar{g}$ to denote the Datalog atom that names the type of group. We write $[\![e]\!]$ and $[\![g]\!]$ for the token embeddings: this allows different tokens of the same type to have different embeddings at time t, depending on how they arose.

We allow new program lines of the following forms:[16]

```
:- eventgroup(functor, dimension).          (4.12a)

group <<- event, condit_1, ..., condit_N.   (4.12b)

event <-- group, condit_1, ..., condit_N.   (4.12c)
```

An `eventgroup` declaration of the form equation (4.12a) is used to declare that atoms with a particular functor refer to event groups, similar to an `event` declaration. We will display such functors with a double underline.

A rule of the form equation (4.12b) is used to trigger a group of possible dependent events. If e is an event token at time t, then it triggers a token g of group type $\bar{g}$ at time t, for each $\bar{g}$ and each rule r having at least one instantiation of the form $\bar{g}$ <<- $\bar{e}$, c_1, ..., c_N for which the c_i are all facts at time t. The embedding of

[16] Mnemonically, note that the "doubled" side of the symbol <<- or <-- is next to the group, since the group usually contains multiple events. This is also why group names are double-underlined in the examples below.

this group token g pools over all such instantiations of rule r (as in equation (4.4)):

$$[\![g]\!] \stackrel{\text{def}}{=} \bigoplus_{c_1,\ldots,c_N}^{\beta_r} \mathbf{W}_r \underbrace{[1; [\![e]\!]; [\![c_1]\!]; \ldots; [\![c_N]\!]]}_{\text{concatenation of column vectors}} \in \mathbb{R}^{D_g} \tag{4.13}$$

where all embeddings are evaluated at time t.

Rules of the form equation (4.12c) are used to specify the possible events in a group. Very similarly to the above, if the group g is triggered at time t, then it contains a token e' of event type $\bar{e}'$, for each $\bar{e}'$ and each rule r having at least one instantiation of the form $\bar{e}'$ `<--` $\bar{g}$`,` c_1`,` $\ldots$`,` c_N for which the c_i are all facts at time t. The embedding of this event token e' pools over all such instantiations of rule r:

$$[\![e']\!] \stackrel{\text{def}}{=} \bigoplus_{c_1,\ldots,c_N}^{\beta_r} \mathbf{W}_r \underbrace{[1; [\![g]\!]; [\![c_1]\!]; \ldots; [\![c_N]\!]]}_{\text{concatenation of column vectors}} \in \mathbb{R}^{D_g} \tag{4.14}$$

where all embeddings are evaluated at time t.

Since each e' in group g is an event, we compute not only an embedding $[\![e']\!]$ but also an unnormalized probability $\lambda_{e'}$, computed just as in section 4.4.2 (using $\exp$ rather than $\mathrm{softplus}$). Exactly one of the finitely many event tokens in g will occur at time t, with event type e' being chosen from g with probability proportional to $\lambda_{e'}$.

Training. In fully supervised training of this model, the dependencies are fully observed. For each dependent event token e' that occurs at time t, the training set specifies what it depends on—that it is a dependent event, which group g it was chosen from, and which rule r established that e' was an element of g. Furthermore, the training set must specify for g which event e triggered it and via which rule r. However, if these dependencies are not fully observed, then it is still possible to take the training objective to be the incomplete-data likelihood, which involves computing the total probability of the bag of events at each time t by summing over all possible

choices of the dependencies.

Marked events. To see the applicability of our formalism, consider a marked point process (such as the marked Hawkes process). This is a traditional type of event sequence model in which each event occurrence also generates a stochastic **mark** from some distribution. The mark contains details about the event. For example, each occurrence of `eat_meal(eve)` might generate a mark that specifies the food eaten and the location of the meal.

Why are marked point processes used in practice? An alternative would be to refine the atoms that describe events so that they contain the additional details. This leads to fine-grained event types such as `eat_meal(eve,apple,tree_of_knowledge)`. However, that approach means that computing $\lambda(t) \stackrel{\text{def}}{=} \sum_{e \in \mathcal{E}(t)} \lambda_e(t)$ during training (section 4.5) or sampling (section 4.F.2) involves summing over a large set of fine-grained events, which is computationally expensive. Using marks makes it possible to generate a coarse-grained event first, modeling its probability without yet considering the different ways to refine it. The event's details are considered only once the event has been chosen. This is simply the usual computational efficiency argument for locally normalized generative models.

Our formalism can treat an event's mark as a dependent event, using the neural architecture above to model the mark probability $p(\texttt{e}' \mid \texttt{e})$ as proportional to $\lambda_{\texttt{e}'}$. The set of possible marks for an event is defined by rules of the form equation (4.12) and may vary by event type and vary by time.

Multiply marked events. Our approach also makes it easy for an event to independently generate multiple marks, which describe different attributes of an event. For example, each meal at time t may select a dependent location,

```
:- eventgroup(restaurants, 5).
:- event(eat_at, 0).
restaurants <<- eat_meal(X).
eat_at(Y) <-- restaurants, is_restaurant(Y).
eat_at(home) <-- restaurants.
```

which associates some dependent restaurant `Y` (or `home`) with the meal.[17] At the same time, the meal may select a *set* of foods to eat, where each food `U`[18] is in competition with `none`[19] to indicate that it may or may not be chosen:

```
:- eventgroup(optdish, 7).
:- event(eat_dish, 0).
:- event(none, 0).
optdish(U) <<- eat_meal(X),
  food(U), opinion(X,U).
eat_dish(U) <-- optdish(U).
none <-- optdish(U) : 0.
```

Recursive marks. Dependent events can recursively trigger dependent events of their own, leading to a tree of event tokens at time t. This makes it possible to model the top-down generation of tree-structured metadata, such as a syntactically well-formed sentence that describes the event Zhang, Lu, and Lapata, 2016. Observing such sentences in training data would then provide evidence of the underlying embeddings of the events. For example, to generate derivation trees from a context-free grammar, encode each nonterminal symbol as an event group, whose events are the production rules that can expand that nonterminal. In general, the probability of a production rule depends on the sequence of production rules at its ancestors, as determined by a

[17] Notice that the choice of event `eat_at(Y)` depends on the person `X` who is eating the meal, through the embedding of this token of ⟦`restaurants`⟧, which depends on ⟦`eat_meal(X)`⟧.

[18] Notice that the unnormalized probability of including `U` in `X`'s meal depends on `X`'s opinion of `U`.

[19] The annotation `: 0` in the last line (explained in section 4.B below) is included as a matter of good practice. In keeping with the usual practice in binary logistic regression, it simplifies the computation of the normalized probabilities, without loss of generality, by ensuring that the unnormalized probability of `none` is constant rather than depending on `U`.

recurrent neural net.

A special case of a tree is a sequence: in the meal example, each dish could be made to generate the next dish until the sequence terminates by generating `none`. The resulting architecture precisely mimics the architecture of an RNN language model Mikolov et al., 2010.

Multiple agents. A final application of our model is in a discrete-time setting where there are multiple agents, which naturally leads to multiple simultaneous events. For example, at each time step t, every person stochastically chooses an action to perform (possibly `none`). This can be accomplished by allowing the `tick` event (section 4.A.5) to trigger one group for each person:

```
:- eventgroup(actions, 7).
actions(X) <<- tick, person(X).
help(X,Y) <-- actions(X), rel(X,Y).
  ⋮
```

This is a group-wise version of line 14 in the main chapter.

A similar structure can be used to produce a "node classification" model in which each node in a graph stochastically generates a label at each time step, based on the node's current embedding (Hamilton, Ying, and Leskovec, 2017b; Xu et al., 2020). The event group for a node contains its possible labels. The graph structure may change over time thanks to exogeneous or endogenous events.

Example. For concreteness, below is a fully generative model of a dynamic colored directed graph, using several of the extensions described in this appendix. The model can be used in either a discrete-time or continuous-time setting.

The graph's nodes and edges have embeddings, as do the legal colors for nodes:

```
:- embed(node, 8).
:- embed(edge, 4).
```

```
3  :- embed(color, 3).
```

In this version, edges are stochastically added and removed over time, one at a time. Any two unconnected nodes determine through their embeddings the probability of adding an edge between them, as well as the initial embedding of this edge. The edge's embedding may drift over time,[20] and at any time determines the edge's probability of deletion.

```
4  :- event(add_edge, 8).
5  :- event(del_edge, 0).
6  add_edge(U,V) :- node(U), node(V), !edge(U,V).
7  del_edge(U,V) :- edge(U,V).
8  edge(U,V) <- add_edge(U,V).
9  !edge(U,V) <- del_edge(U,V).
```

Adding `edge(U,V)` to the graph causes two dependent events that simultaneously and stochastically relabel both `U` and `V` with new colors. This requires triggering two event groups (unless `U=V`). A node's new color `C` depends stochastically on the embeddings of the node and its neighbors, as well as the embeddings of the colors:

```
10 :- eventgroup(labels, 8).
11 :- event(label, 8).
12 labels(U) <<- add_edge(U,V).
13 labels(V) <<- add_edge(U,V).
14 label(X,C) <-- labels(X), color(C), node(X), edge(X,Y), node(Y).
```

Finally, here is how a relabeling event does its work. The `has_color` atoms that are updated here are simply facts that record the current coloring, with no embedding. However, the rules below ensure that a *node*'s embedding records its *history* of colors (and that it has only one color at a time):

```
15 !has_color(U,D) <- label(U,C), color(D).
16 has_color(U,C) <- label(U,C).
```

[20] In the continuous-time setting, the drift is learned. In the discrete-time setting, we must explicitly specify drift as explained in section 4.A.5, via a rule such as `edge(U,V) <- tick`.

```
17  node(U) <- has_color(U,C), color(C).
```

The initial graph at time $t = 0$ can be written down by enumeration:

```
18  color(red).
19  color(green).
20  color(blue).
21  has_color(0,red).
22  has_color(1,blue)
23  has_color(2,red).
24  node(U) :- has_color(U,C).
25  edge(0,1) <- init.
```

Inheritance. As a convenience, we allow an event group to be used anywhere that an event can be used—at the start of the body of a rule of type equation (4.2a), equation (4.2b), or equation (4.12b). Such a rule applies at times when the group is triggered (just as a rule that mentions an event, instead of a group, would apply at times when that event occurred).

This provides a kind of inheritance mechanism for events:

```
47  :- eventgroup(act, 5).
48  act(X) <<- sleep(X).
49  act(X) <<- help(X,Y), person(Y).
      ⋮
50  person(Y) <- act(X), parent(X,Y), person(Y).
51  animal(Y) <- act(X), own(X,Y), animal(Y).
```

This means that whenever `X` takes any action—`sleep`, `help`, etc.—lines 50–51 will update the embeddings of `X`'s children and pets.

Adopting the terminology of object-oriented programming, `act(eve)` functions as a class of events (i.e., event type), whose subclasses include `help(eve,adam)` and many others. In this view, each particular instance (i.e., event token) of the subclass `help(eve,adam)` has a method that returns its embedding in $\mathbb{R}^{D_{\texttt{help}}}$. But lines 50–51 instead view this `help(eve,adam)` event as an instance of the superclass `act(eve)`,

and hence call a method of that superclass to obtain the embedding of the group token `act(eve)` in $\mathbb{R}^{D_{\texttt{act}}} = \mathbb{R}^5$, as defined via equation (4.13).

In the above example, the event group is actually empty, as there are no rules of type equation (4.12c) that populate it with dependent events. Thus, no dependent events occur as a result of the group being triggered. The empty event group is simply used as a class. One could, however, add rules such as

```
52 act_at(L) <-- act(X), location(L).
```

which marks each action (of any type) with a location.

4.B Parameter Sharing Details

Throughout section 4.4, the parameters $\mathbf{W}$ and β are indexed by the rule number r. (They appear in equations (4.4) and (4.8).) Thus, the number of parameters grows with the number of rules in our formalism. However, we also allow further flexibility to *name* these parameters with atoms, so that they can be shared among and within rules. This is achieved by explicitly naming the parameters to be used by a rule:

$$\texttt{head : beta :- : bias\_vec, condit}_1 \texttt{ : mat}_1\texttt{, ..., condit}_N \texttt{ : mat}_N\texttt{.}$$

Now β_r in equation (4.4) is replaced by a scalar parameter named by the atom `beta`. Similarly, the affine transformation matrix $\mathbf{W}_r$ in equation (4.4) is replaced by a parameter matrix that is constructed by *horizontally* concatenating the column vector and matrices named by the atoms `bias_vec`, $\texttt{mat}_1, \ldots, \texttt{mat}_N$ respectively.

To be precise, $\texttt{mat}_i$ will have $D_{\texttt{head}}$ rows and $D_{\texttt{condit}_i}$ columns. The computation equation (4.4) can be viewed as multiplying this matrix by the vector embedding of the atom that instiatiates $\texttt{condit}_i$, yielding a vector in $\mathbb{R}^{D_{\texttt{head}}}$. It then sums these vectors for $i = 1, \ldots, N$ as well as the bias vector (also in $\mathbb{R}^{D_{\texttt{head}}}$), obtaining a vector

in $\mathbb{R}^{D_{head}}$ that it provides to the pooling operator.

These parameter annotations with the `:` symbol are optional (and were not used in the previous sections). If any of them is not specified, it is set automatically to be rule- and position-specific: in the r^{th} rule, *beta* defaults to `params(`r`,beta)`, *bias_vec* defaults to `params(`r`,bias)`, and mat_i defaults to `params(`r`,`i`)`.

As shorthand, we also allow the form

$$head \ : \ beta \ :- \ condit_1, \ condit_N \ :: \ full\_matrix.$$

where *full_matrix* directly names the concatenation of matrices that replaces $\mathbf{W}_r$.

The parameter-naming mechanism lets us share parameters across rules by reusing their names. For example, blessings and curses might be inherited using the same parameters:

```
53  cursed(Y) :- cursed(X),parent(X,Y) :: inherit.
54  blessed(Y) :- blessed(X),parent(X,Y) :: inherit.
```

Conversely, to do *less* sharing of parameters, the parameter names may mention variables that appear in the head or body of the rule. In this case, different instantiations of the rule may invoke different parameters. (*beta* is only allowed to contain variables that appear in the head, because each way of instantiating the head needs a single β to aggregate over all the compatible instantations of its body.)

For example, we can modify lines 53 and 54 into

```
55  cursed(Y) : descendant(Y) :-
      cursed(X), parent(X,Y) :: inherit(X,Y).
56  blessed(Y) : descendant(Y) :-
      blessed(X), parent(X,Y) :: inherit(X,Y).
```

Now each `X`, `Y` pair has its own $\mathbf{W}$ matrix (shared by curses and blessings), and similarly, each `Y` has its own β scalar. This example has too many parameters to be

practical, but serves to illustrate the point.

If `X` or `Y` is an entity created by the $*$ mechanism (section 4.A.4), then the name will be constructed using a literal $*$, so that all newly created entities use the same parameters. This ensures that the number of parameters is finite even if the number of entities is unbounded. As a result, parameters can be trained by maximum likelihood and reused every time a sequence is sampled, even though different sequences may have different numbers of entities. Although novel entities share parameters, facts that differ only in their novel entities may nonetheless come to have different embeddings if they are created or updated in different circumstances.

The special parameter name `0` says to use a zero matrix:

```
57  cursed(Y) : descendant :-
      : inherit_bias,
      cursed(X) : inherit,
      parent(X,Y) : 0.
```

In this example, the condition `parent(X,Y)` must still be non-null for the rule to apply, but we ignore its embedding.

The same mechanism can be used to name the parameters of `<-` rules. In this case, *event* at the start of the body can also be annotated, as *event* : $matrix_0$. The horizontal concatenation of named matrices now includes the matrix named by $matrix_0$, and is used to replace $\mathbf{W}_r$ in equation (4.8).

For a `<-` rule, it might sometimes be desirable to allow finer-grained control over how the rule affects the drift of a cell block over time (see equation (4.16) in section 4.C below). For example, forcing $\underline{\mathbf{f}} = \mathbf{1}$ and $\underline{\mathbf{i}} = \mathbf{0}$ in equation (4.17) ensures via equation (4.18) that when the rule updates h, it will not introduce a discontinuity in the $\boxed{h}(t)$ function, although it might change the function's asymptotic value and decay

rate. (This might be useful for the `tick` rules mentioned in footnote 15, for example.) Similarly, forcing $\bar{\mathbf{f}} = \mathbf{1}$ and $\bar{\mathbf{i}} = \mathbf{0}$ in equation (4.17) ensures via equation (4.19) that the rule does not change the asymptotic value of the $\boxed{h}(t)$ function. These effects can be accomplished by declaring that certain values are $\pm\infty$ in the first column of $\mathbf{W}_r$ in equation (4.8) (as this column holds bias terms). We have not yet designed a syntax for such declarations.

We can also name the softplus temporal scale parameter τ in section 4.4.2. For example, we can rewrite line 13 of section 4.3.4 as

```
58  :- event(help, 8) : intervene.
```

and allow `harm` to share τ with `help`:

```
59  :- event(harm, 8) : intervene.
```

4.C Updating Drift Functions in the Continuous-Time LSTM

Here we give the details regarding continuous-time LSTMs, which were omitted from section 4.4.3 since we wouldn't like to distract readers from understanding the general idea by early presentation of technical details. We follow the design of Mei and Eisner (2017), in which each cell changes endogenously between updates, or "drifts," according to an exponential decay curve:

$$c(t) \stackrel{\text{def}}{=} \bar{c} + (\underline{c} - \bar{c}) \exp(-\delta(t-s)) \quad \text{where } t > s \tag{4.15}$$

This curve is parameterized by $(s, \underline{c}, \bar{c}, \delta)$, where

- s is a starting time—specifically, the time when the parameters were last updated
- $\underline{c}$ is the starting cell value, i.e., $c(s) = \underline{c}$

- $\bar{c}$ is the asymptotic cell value, i.e., $\lim_{t\to\infty} c(t) = \bar{c}$
- $\delta > 0$ is the rate of decay toward the asymptote; notice that the derivative $c'(t) = \delta \cdot (\bar{c} - \underline{c})$

In the present chapter, we similarly need to define the trajectory through $\mathbb{R}^{D_h}$ of the cell block $\boxed{h}$ associated with fact h. That is, we need to be able to compute $\boxed{h}(t) \in \mathbb{R}^{D_h}$ for any t. Since $\boxed{h}$ is not a single cell but rather a block of D_h cells, it actually needs to store not 4 parameters as above, but rather $1 + 3D_h$ parameters. Specifically, it stores $s \in \mathbb{R}$, which is the time that the block's parameters were last updated: this is shared by all cells in the block. It also stores vectors that we refer to as $\boxed{h}^{\underline{\mathbf{c}}}, \boxed{h}^{\bar{\mathbf{c}}}, \boxed{h}^{\boldsymbol{\delta}} \in \mathbb{R}^{D_h}$. Now analogously to equation (4.15), we define the trajectory of the cell block elementwise:

$$\boxed{h}(t) \stackrel{\text{def}}{=} \boxed{h}^{\bar{\mathbf{c}}} + (\boxed{h}^{\underline{\mathbf{c}}} - \boxed{h}^{\bar{\mathbf{c}}}) \exp(-\boxed{h}^{\boldsymbol{\delta}} \cdot (t - s)), \tag{4.16}$$

for all $t > s$ (up to and including the time of the next event that results in updating the block's parameters).

We now describe exactly *how* the block's parameters are updated when an event occurs at time s. Recall that for the discrete-time case, for each (r, m), we obtained $[h]^{\leftarrow}_{rm} \in \mathbb{R}^{3D_h}$ by evaluating equation (4.8) at time s. We then set $(\mathbf{f};\ \mathbf{i};\ \mathbf{z}) \stackrel{\text{def}}{=} \sigma([h]^{\leftarrow}_{rm})$. In the continuous-time case, we evaluate equation (4.8) at time s to obtain $[h]^{\leftarrow}_{rm} \in \mathbb{R}^{7D_h}$ (so $\mathbf{W}_r$ needs to have more rows), and accordingly obtain 7 vectors in $(0, 1)^{D_h}$,

$$(\underline{\mathbf{f}};\ \underline{\mathbf{i}};\ \underline{\mathbf{z}};\ \bar{\mathbf{f}};\ \bar{\mathbf{i}};\ \bar{\mathbf{z}};\ \mathbf{d}) \stackrel{\text{def}}{=} \sigma([h]^{\leftarrow}_{rm}) \tag{4.17}$$

which we use similarly to equation (4.10) to define update vectors for the current cell

values (time s) and the asymptotic cell values (time ∞), respectively

$$[\boxed{h}]^{\Delta \underline{\mathbf{c}}}_{rm} \stackrel{\text{def}}{=} (\underline{\mathbf{f}} - 1) \cdot \boxed{h}(s) + \underline{\mathbf{i}} \cdot (2\underline{\mathbf{z}} - 1) \tag{4.18}$$

$$[\boxed{h}]^{\Delta \bar{\mathbf{c}}}_{rm} \stackrel{\text{def}}{=} (\bar{\mathbf{f}} - 1) \cdot \boxed{h}^{\bar{\mathbf{c}}} \quad + \bar{\mathbf{i}} \cdot (2\bar{\mathbf{z}} - 1) \tag{4.19}$$

as well as a vector of proposed decay rates:[21]

$$[\boxed{h}]^{\boldsymbol{\delta}}_{rm} \stackrel{\text{def}}{=} \text{softplus}_1(\sigma^{-1}(\mathbf{d})) \quad \in \mathbb{R}^{D_h}_{>0} \tag{4.20}$$

We then pool the update vectors from different (r, m) and apply this pooled update, much as we did for the discrete-time cell values in equations (4.11)–(4.10):

$$\boxed{h}^{\mathbf{c}} \stackrel{\text{def}}{=} \boxed{h}(s) + \sum_r \bigoplus_m^{\beta_r} [\boldsymbol{h}]^{\Delta \underline{\mathbf{c}}}_{rm} \tag{4.21}$$

$$\boxed{h}^{\bar{\mathbf{c}}} \stackrel{\text{def}}{=} \boxed{h}^{\bar{\mathbf{c}}} \quad + \sum_r \bigoplus_m^{\beta_r} [\boldsymbol{h}]^{\Delta \bar{\mathbf{c}}}_{rm} \tag{4.22}$$

The special cases mentioned just below the update equation (4.11) are also followed for the updates equation (4.21)–equation (4.22).

The final task is to pool the decay rates to obtain $\boxed{h}^{\boldsymbol{\delta}}$. It is less obvious how to do this in a natural way. Our basic idea is that for the i^{th} cell, we should obtain the decay rate $(\boxed{h}^{\boldsymbol{\delta}})_i$ by a weighted harmonic mean of the decay rates $([\boldsymbol{h}]^{\boldsymbol{\delta}}_{rm})_i$ that were proposed by different (r, m) pairs. A given (r, m) pair should get a high weight in this harmonic mean to the extent that it contributed large updates $([\boldsymbol{h}]^{\Delta \underline{\mathbf{c}}}_{rm})_i$ or $([\boldsymbol{h}]^{\Delta \bar{\mathbf{c}}}_{rm})_i$.

Why harmonic mean? Observe that the exponential decay curve equation (4.15) has a **half-life** of $\frac{\ln 2}{\delta}$. In other words, at any moment t, it will take time $\frac{\ln 2}{\delta}$ for the curve to travel halfway from its current value $c(t)$ to $\bar{c}$. (This amount of time is

[21] equation (4.20) simply replaces the σ that produced $\mathbf{d}$ with softplus_1 (defined in section 4.4.2), since there is no reason to force decay rates into $(0, 1)$.

independent of t.) Thus, saying that the decay rate is a weighted harmonic mean of proposed decay rates is equivalent to saying that the half-life is a weighted arithmetic mean of proposed half-lives,[22] which seems like a reasonable pooling principle.

Thus, operating in parallel over all cells i by performing the following vector operations elementwise, we choose

$$\boxed{h}^{\boldsymbol{\delta}} \stackrel{\text{def}}{=} \left(\frac{\sum_r \sum_m \mathbf{w}_{rm} \cdot \left([h]^{\boldsymbol{\delta}}_{rm}\right)^{-1}}{\sum_r \sum_m \mathbf{w}_{rm}} \right)^{-1} \tag{4.23}$$

We define the vector of unnormalized non-negative weights $\mathbf{w}_{rm}$ from the updated $\boxed{h}^{\underline{\mathbf{c}}}$ and $\boxed{h}^{\bar{\mathbf{c}}}$ values by

$$\begin{aligned} \mathbf{w}_{rm} \stackrel{\text{def}}{=} & \left(\bigoplus_{m'}^{\beta_r} \left| [h]^{\Delta \underline{\mathbf{c}}}_{rm'} \right| \right) \cdot \frac{\left| [h]^{\Delta \underline{\mathbf{c}}}_{rm} \right|^{\beta_r}}{\sum_{m'} \left| [h]^{\Delta \underline{\mathbf{c}}}_{rm'} \right|^{\beta_r}} \\ & + \left(\bigoplus_{m'}^{\beta_r} \left| [h]^{\Delta \bar{\mathbf{c}}}_{rm'} \right| \right) \cdot \frac{\left| [h]^{\Delta \bar{\mathbf{c}}}_{rm} \right|^{\beta_r}}{\sum_{m'} \left| [h]^{\Delta \bar{\mathbf{c}}}_{rm'} \right|^{\beta_r}} \\ & + \left| \boxed{h}^{\bar{\mathbf{c}}} - \boxed{h}^{\underline{\mathbf{c}}} \right| \end{aligned} \tag{4.24}$$

The following remarks should be read elementwise, i.e., consider a particular cell i, and read each vector $\mathbf{x}$ as referring to the scalar $(\mathbf{x})_i$.

The weights defined in equation (4.24) are valid weights to use for the weighted harmonic mean equation (4.23):

- $\mathbf{w}_{rm} \geq 0$, because of the use of absolute value.
- $\mathbf{w}_{rm} > 0$ strictly unless $\boxed{h}^{\bar{\mathbf{c}}} = \boxed{h}^{\underline{\mathbf{c}}}$. Thus, the decay rate $\boxed{h}^{\boldsymbol{\delta}}$ as defined by

[22] It is also equivalent to saying that the (2/3)-life is a weighted arithmetic mean of proposed (2/3)-lives, since equation (4.15) has a (2/3)-life of $\frac{\ln 3}{\delta}$. In other words, there is nothing special about the fraction $1/2$. *Any* choice of fraction would motivate using the harmonic mean.

equation (4.23) can only be undefined (that is, $\frac{0}{0}$) if $\boxed{h}^{\bar{\mathbf{c}}} = \boxed{h}^{\underline{\mathbf{c}}}$, in which case that decay rate is irrelevant anyway.

The way to understand the first line of equation (4.24) is as a heuristic assessment of how much the cell's curve equation (4.15) was affected by (r, m) via $[h]_{rm}^{\Delta \underline{\mathbf{c}}}$'s effect on $\boxed{h}^{\underline{\mathbf{c}}}$. First of all, $\left(\bigoplus_{m'}^{\beta_r} \left|[h]_{rm'}^{\Delta \underline{\mathbf{c}}}\right|\right)$ is the pooled magnitude of *all* of the r^{th} rule's attempts to affect $\boxed{h}^{\underline{\mathbf{c}}}$. Using the absolute value ensures that even if large-magnitude attempts of opposing sign canceled each other out in equation (4.21), they are still counted here as large attempts, and thus give the r^{th} rule a stronger total voice in determining the decay rate $\boxed{h}^{\boldsymbol{\delta}}$. This pooled magnitude for the r^{th} rule is then partitioned among the attempts (r, m). In particular, the fraction in the first line denotes the portion of the r^{th} rule's pooled effect on $\boxed{h}^{\underline{\mathbf{c}}}$ that should be heuristically attributed to (r, m) specifically, given the way that equation (4.21) pooled over all m (recall that this invokes equation (4.6a)).

Thus, the first line of equation (4.24) considers the effect of (r, m) on $\underline{\mathbf{c}}$. The second line adds its effect on $\bar{\mathbf{c}}$. The third line effectively acts as smoothing so that we do not pay undue attention to the size ratio among different updates if these updates are tiny. In particular, if all of the updates $[h]_{rm}^{\Delta \underline{\mathbf{c}}}$ and $[h]_{rm}^{\Delta \bar{\mathbf{c}}}$ are small compared to the total height of the curve, namely $\left|\boxed{h}^{\bar{\mathbf{c}}} - \boxed{h}^{\underline{\mathbf{c}}}\right|$, then the third line will dominate the definition of the weights $\mathbf{w}_{rm}$, making them close to uniform. The third line is also what prevents inappropriate division by 0 (see the second bullet point above).

4.D Likelihood Computation Details

In this section we discuss the log-likelihood formulas in section 4.5.

For the discrete-time setting, the formula simply follows from the fact that the

log-probability of event $\mathtt{e}$ at time t was defined to be $\log\left(\lambda_{\mathtt{e}}(t)/\lambda(t)\right)$.

The log-likelihood formula has been discussed in section 2.2. The integral term is computed using Monte Carlo approximation. At each sampled time t, that method requires a summation over all events to obtain $\lambda(t)$. This summation can be expensive when there are many event types. Thus, we estimate the sum using a simple downsampling trick, as follows. At any time t that is sampled to compute the integral, let $\mathcal{E}(t)$ be the set of *possible* event types under the database at time t. We construct a bag $\mathcal{E}'(t)$ by uniformly sampling event types from $\mathcal{E}(t)$ with replacement, and estimate

$$\lambda(t) \approx \frac{|\mathcal{E}|}{|\mathcal{E}'|} \sum_{e \in \mathcal{E}'} \lambda_{\mathtt{e}}(t)$$

This estimator is unbiased yet remains much less expensive to compute especially when $|\mathcal{E}'| \ll |\mathcal{E}|$. In our experiments, we took $|\mathcal{E}'| = 10$ and still found empirically that the variance of the log-likelihood estimate (computed by running multiple times) was rather small.

Another computational expense stems from the fact that we have to make Datalog queries after every event to figure out the proof DAG of each provable Datalog atom. Queries can be slow, so rather than repeatedly making a given query, we just memoize the result the first time and look it up when it is needed again (Swift and Warren, 2012). However, as events are allowed to change the database, results of some queries may also change, and thus the memos for those queries become incorrect (stale). To avoid errors, we currently flush the memo table every time the database is changed. This obviously reduces the usefulness of the memos. An implementation improvement for future work is to use more flexible strategies that create memos and update them incrementally through change propagation (Acar and Ley-Wild, 2008; Hammer, 2012;

Filardo and Eisner, 2012).

4.E How to Predict Events

Figures 4.3 and 4.5 include a task-based evaluation where we try to predict the *time* and *type* of the next event. More precisely, for each event in each held-out sequence, we attempt to predict its time given only the preceding events, as well as its type given both its true time and the preceding events.

To carry out the predictions, we follow section 2.4 and use the minimum Bayes risk (MBR) principle to predict the time and type with lowest expected loss. Notice that our approach will never predict an impossible event type. For example, `help(eve,adam)` won't be in $\mathcal{E}(t_i)$ and thus will have *zero* probability if $[\![\texttt{rel(eve,adam)}]\!](t_i) = \mathsf{null}$ (maybe because `eve` stops having `opinion`s on anything that `adam` does anymore).

In some circumstances, one might also like to predict the most likely type out of a *restricted* set $\mathcal{E}'(t_i) \subsetneq \mathcal{E}(t_i)$. This allows one to answer questions like "If we know that some event `help(eve,Y)` happened at time t_i, then which person `Y` did `eve` `help`, given all past events?" The answer will simply be $\arg\max_{e \in \mathcal{E}'(t_i)} \lambda_e(t_i)$.

4.F Experimental Details

4.F.1 Dataset Statistics

Table 4.1 shows statistics about each dataset that we use in this chapter (section 4.7).

4.F.2 Details of Synthetic Dataset and Models

We synthesized data for section 4.7.1 by sampling event sequences from the structured NHP specified by our Datalog program in that section. We chose $N = 4$ and $M = 4, 8, 16$, and thus end up with three different datasets.

DATASET	$\lvert\mathcal{K}\rvert$	# OF EVENT TOKENS			# OF SEQUENCES		
		TRAIN	DEV	TEST	TRAIN	DEV	TEST
SYNTHETIC $M = 4$	16	42000	2100	2100	2000	100	100
SYNTHETIC $M = 8$	32	42000	2100	2100	2000	100	100
SYNTHETIC $M = 16$	64	42000	2100	2100	2000	100	100
IPTV	49000	27355	4409	4838	1	1	1
ROBOCUP	528	2195	817	780	2	1	1

Table 4.1: Statistics of each dataset.

For each M, we set the sequence length $I = 21$ and then used the thinning algorithm (Mei and Eisner, 2017; Mei, Qin, and Eisner, 2019) to sample the first I events over $[0, \infty)$. We set $T = t_I$, i.e., the time of the last generated event. We generated 2000, 100 and 100 sequences for each training, dev and test set respectively. We showed the learning curves for $M = 8$ and 16 in Figure 4.2 and left out the plot for $M = 4$ because it is boringly similar.

For the unstructured NHP baseline, the program given in section 4.7.1 is not quite accurate. To exactly match the architecture of the neural Hawkes process, we have to use the notation of section 4.B to ensure that each of the MN event types uses its its own parameters for its embedding and probability:

```
is_process(1).
⋮
is_process(M).

is_type(1).
⋮
is_type(N).

:- embed(world, 8).
:- embed(is_event, 8).
:- event(e, 0).
is_event(M,N) :- is_process(M), is_type(N) :: emb(M,N).
e(M,N) :- world, is_process(M), is_type(N) :: prob(M,N).
```

```
12  world <- init.
13  world <- e(M,N), is_event(M,N), world.
```

As section 4.7.1 noted, an event's probability is carried by an `e` fact, but its embedding is carried by an `is_event` fact. This is because the NHP uses dynamic event probabilities (which depend on `world`) but static event embeddings (which do not). Otherwise, we could merge the two by using dimension 8 for `e` in line 9, and removing `is_event` by deleting it from line 13 and deleting lines 8 and 10.

4.F.3 Details of IPTV Dataset and our NDTT Model

For the IPTV domain, the time unit is 1 minute. Thus, in the graph for time prediction, an error of 1.5 (for example) means an error of 1.5 minutes. The exogenous `release` events were not included in the dataset of Xu, Luo, and Carin (2018), but Xu, Luo, and Carin (p.c.) kindly provided them to us.

For our experiments in section 4.7.2, we used the events of days 1–200, days 201–220, and days 221–240 as training, dev and test data respectively—so there is just one long sequence in each case. (We saved the remaining days for future experiments.)

We evaluated the ability of the trained model to extrapolate from days 1–200 to future events. That is, for dev and test, we evaluated the model's predictive power on the held-out dev and test events respectively. However, when predicting each event, the model was still allowed to condition on the *full* history of that event (starting from day 1). This full history was needed to determine the facts in the database, their embeddings, and the event intensities.

Each observed event has one of the forms

```
1  init
2  release(P)
3  watch(U,P)
```

For example, watch(u4,p9) occurs when user u4 watches television program p9.

The dataset also provides time-invariant facts of the form

```
has_tag(P,T)
```

which tag programs with attributes.[23] For example:

```
has_tag(p1,comedy).
  ⋮
has_tag(p49,romance).
```

We develop our NDTT program as follows. A television program is added to the database only when it is released:

```
program(P) <- release(P).
```

Now that P is a program, it can be watched:

```
watch(U,P) := user(U), program(P).
```

The probability of a watch event depends on the current embeddings of the user and the program:

```
embed(user, 8).
embed(program, 8).
```

Of course, we have to declare that 'watch' is an event:

```
event(watch,8).
```

Notice that we equipped watch with a 8-dimensional embedding as well as a probability. The embedding encodes some details of the event (who watched what). This detailed watch event then updates what we know about both the user and the program, in order to predict future watch events:

```
user(U)    <- watch(U,P).
program(P) <- watch(U,P).
```

[23] Users could also have tags, to record their demographics or interests. However, the IPTV dataset does not provide such tags.

The := connector in line 8 requested highway connections around watch (section 4.A.3), so these update lines 12 and 13 not only consider ⟦watch(U,P)⟧ but also directly consider ⟦user(U)⟧ and ⟦program(P)⟧. This is similar to a traditional LSTM update, and in our initial pilot experiments we found it to work better than simply using :- in line 8.

Where do the user facts come from? Line 12 would automatically add user(U) to the database upon the first time they watched a program. But such an event watch(U,P) is not itself possible (line 8) until user(U) is already in the database. To break this circularity, we must populate the database with users in advance.

If we simply declared these users as

```
14  user(u1).
15  user(u2).
      ⋮
```

then the model would include separate parameters for each of these rules. However, fitting user-specific parameters would be hard for users who have only a small amount of data. Instead, we make all the user rules share parameters (see section 4.B):

```
16  user(u1) :: user_init.
17  user(u2) :: user_init.
      ⋮
```

Thus, all users start out in the same place,[24] and a user's embedding only depends entirely on programs that they've watched so far. An update to the user's embedding (line 12) could be either material or epistemic: that is, it may reflect *actual* changes over time in the user's taste, or merely changes in our *knowledge* of the user's taste.

[24] We suspect that it would have been adequate for that initial user embedding to be the **0** vector, which we could have specified by writing :: 0 instead of :: user_init. That is how we treated programs in this model (line 19 below), and how we treated both users and programs in section 4.F.4. We regret the discrepancy.

Ultimately, the training procedure learns whatever updates help the model to better predict the user's future `watch` events.

There is one more subtlety regarding user embeddings. In the program above, `user(u1)` is true at all times, but is "launched" (in the sense of section 4.4.3) only by the first event of the form `watch(u1,P)`. Thus, we learn nothing about the user from the fact that time has elapsed without their having yet watched any programs: they do not yet have a cell block that can drift to track the passage of time. To fix this, we add the following rule so that all users are simultaneously launched at time 0 by the exogenous `init` event:

```
18  user(U) <- init, user(U).
```

This ensures that the user has an LSTM cell block starting at time 0, which can drift to mark the passage of time even before the user has watched any programs. This rule for users is analogous to line 7 for programs.

Where do the `program` facts come from? We declare them much as we declared the `user` facts:[25]

```
19  program(p1) :: 0.
20  program(p2) :: 0.
      ⋮
```

However, a program's embedding should also be affected by its tags:[26]

```
21  program(P) :- has_tag(P,T), tag(T).
```

where each tag is declared separately:

```
22  embed(tag, 8).
```

[25] Actually, if `p1` has at least one tag, then we can omit line 19 because line 21 below will be enough to prove that `p1` is a program. In the IPTV dataset, every program does have at least one tag, so we omit all rules like 19, which do not affect the facts or their embeddings.

[26] Recall that facts like `has_tag(p1,comedy)` were declared in the initial database, have no embeddings, and never change.

```
23  tag(adventure).
24  tag(comedy).
      ⋮
```

Note that the rules like 23 and 24 introduce tag-specific parameters. For example, the bias vector of line 23 provides an embedding of the `adventure` tag. As each tag has a lot of data, these tag-specific parameters should be easier to learn than user-specific parameters.

The initial embedding of a tag is then affected by who watches programs with that tag, and when. In other words, just as the `watch` events update our understanding of individual users, they also track how the meaning of each tag changes over time:

```
25  tag(T) <- init, tag(T).
26  tag(T) <- watch(U,P), has_tag(P,T), tag(T).
```

As before, these updates are rich because the `watch` event has an embedding and also supplies highway connections.

We finish with a final improvement to the model. Above, `program(P)` is affected both by `P`'s tags via the `:-` line 21 and by its history of `watch` events via the `<-` line 13. The NDTT equations would simply add these influences via rule (7). Instead, we edit the program to combine these influences nonlinearly. This gives a deeper architecture:

```
27  program(P) :- program_profile(P), program_history(P).
28  program_profile(P) :- has_tag(P,T), tag(T).
29  program_history(P) <- release(P).
30  program_history(P) <- watch(U,P), user(U), program(P).
```

where lines 28–30 replace rules 21, 7, and 13 respectively.

In principle, facts with different functors can be embedded in vector spaces of different dimensionality, as needed. But in all of our experiments, we used the same dimensionality for all functors, so as to have only a single hyperparameter to tune.

If the hyperparameter were 8, for example, our Datalog program would have the declarations

```
31  :- embed(user, 8).
32  :- embed(program, 8).
33  :- embed(profile, 8).
34  :- embed(released, 0).
35  :- embed(watchhistory, 8).
36  :- embed(tag, 8).
37  :- event(watch, 8).
```

where watch has an extra dimension for its intensity. The hyperparameter tuning method and its results are described in section 4.F.7 below.

4.F.4 Baseline Programs on IPTV Dataset

We also implemented baseline models that were inspired by the Know-Evolve Trivedi et al., 2017 and DyRep Trivedi et al., 2019 frameworks. Our architectures are not identical: for example, our line 3 below models each event probability using a feed-forward network in place of a bilinear function. However, Trivedi (p.c.) agrees that the architectures are similar. Note that these prior papers did not apply their frameworks specifically to the IPTV dataset (nor to RoboCup).

The Know-Evolve and DyRep programs specify the same `user`, `program`, and `has_tag` facts as in section 4.F.3, except that the initial embedding `user_init` is fixed to **0** (see footnote 24).

The Know-Evolve program continues as follows.

Whereas a watch fact in section 4.F.3 carried both a probability and an embedding, here we split off the embedding into a separate fact and compute it differently from the probability, to be more similar to Trivedi et al. (2017):

```
1  :- event(watch, 0).
```

```
2   := embed(watch_emb, 8).
3   watch(U,P) := user(U), program(P).
4   watch_emb(U,P) := user(U) : pair, program(P) : pair.
```

Notice that line 4 in effect multiplies the sum $[\![\texttt{user(U)}]\!] + [\![\texttt{program(P)}]\!]$ by the `pair` matrix before applying $\tanh$.

The cell blocks are now launched and updated as follows:

```
5   user(U)    <- init, user(U).
6   program(P) <- init, program(P).
7   user(U)    <- watch(U,P), watch_emb(U,P).
8   program(P) <- watch(U,P), watch_emb(U,P).
```

Of course, when the embedding of `user(U)` or `program(P)` is updated, the embedding of `watch_emb(U,P)` also changes to reflect this.

What are the differences from section 4.F.3? Since Trivedi et al. (2017) did not support changes over time to the set of possible events, we omitted this feature from our Know-Evolve program above. Specifically, the program does not use the `release` events in the dataset—it treats all programs as having been released by `init` at time 0. The program also has no highway connections, nor the deeper architecture at lines 27–30 of section 4.F.3, and it does not make use of the program tags.

Our DyRep version of the program makes a few changes to follow the principles of Trivedi et al., 2019. The main ideas of DyRep are as follows:

- Entities are represented as nodes in a graph (here: programs, users, and tags).
- Each node has an embedding.
- The properties of an entity are represented by labeled edges that link it to other nodes (here: `has_tag(P,T)`).
- The graph structure can change due to exogenous forces (see line 9 below).

- Any pair of entities can communicate at any time. (These communications are the events in our temporal event sequences, such as `watch(U,P)`.)
- The probability of an event depends on the embeddings of the two nodes that communicate (here: line 3).
- When an event occurs, it updates the embeddings of (only) the two nodes that communicate (see lines 10 and 11 below).
- An update to a node's embedding also considers the embeddings of its neighbors in the graph (see line 12 below).

Thus, we replace lines 6–8 above with

```
9  program(P) <- release(P).
10 user(U)    <- watch(U,P), user(U)    :: event.
11 program(P) <- watch(U,P), program(P) :: event.
```

Thus, DyRep now permits the set of watchable programs (nodes) to change over time, but the `user` and `program` updates are less well-informed than in Know-Evolve: the updates to the user embedding no longer look at the current program embedding, nor vice-versa.[27] Indeed, DyRep no longer uses `watch_emb` and can drop line 4.

Where our Know-Evolve program did not use tags, our DyRep program can encode tags using `has_tag` edges. Thus, when a program `P` is watched, the update to the program's embedding depends in part on its tags:

```
12 program(P) <- watch(U,P), tag(T), has_tag(P,T).
```

The embedding $[\![\texttt{tag(T)}]\!]$ is defined as in our full model of section 4.F.3, except that it is now static (except for drift). It is no longer updated by watch events, because the `watch(U,P)` event only updates `U` and `P`. In contrast, the Datalog line 26 in

[27] To allow better-informed updates within the DyRep formalism, we could have included edges between all users and all programs. But then every update would depend on all users and all programs—which is exactly the "everything-affects-everything" problem that our work aims to cure (section 4.1)!

section 4.F.3 was able to draw T into the computation by joining watch(U,P) to has_tag(P,T).

4.F.5 Details of RoboCup Dataset and our NDTT Model

For the RoboCup domain, the time unit is 1 second. Thus thus in the graph for time prediction, an error of 1.5 (for example) means an error of 1.5 seconds.

For our experiments in section 4.7.2, we used Final 2001 and 2002, Final 2003, and Final 2004 as training, dev, and test data respectively. Each sequence is a single game and each dataset contains multiple sequences.

Each observed event has one of the forms

```
kickoff(P)
kick(P)
goal(P)
pass(P,Q)
steal(Q,P)
init
```

which we will describe shortly. The database also contains facts about the teams. There are 2 teams, each with 11 robot players. Any pair of players P and Q are either teammates or opponents:

```
teammate(P,Q) :- in_team(P,T), in_team(Q,T), not_eq(P,Q).
opponent(P,Q) :- in_team(P,T), in_team(Q,S), not_eq(T,S).
```

These relations are induced using the database facts

```
in_team(a1,a).
  ⋮
in_team(a11,a).
in_team(b1,b).
  ⋮
in_team(b11,b).
```

together with an inequality relation on entities, not_eq, which can be spelled out with

a quadratic number of additional facts if the Datalog implementation does not already provide it as a built-in relation:

```
not_eq(a1, a2).      % players
not_eq(a1, a3).
  ⋮
not_eq(b11, b10).
not_eq(a, b).        % teams
not_eq(b, a).
```

We allow the ball to be in the possession of either a specific player, or a team as a whole. A game starts with team `a` taking possession of the ball:[28]

```
has_ball(a) <- init.
```

A random player `P` in team `a` now assumes possession of the ball, taking it from the team as a whole.[29] This is called a `kickoff` event, although in RoboCup—unlike human soccer—`P` does not kick the ball off into the distance but retains it.

```
kickoff(P) :- in_team(P,T), has_ball(T).
!has_ball(T) <- kickoff(P), in_team(P,T).
has_ball(P)  <- kickoff(P).
```

Thereafter, the player who has possession of the ball can kick it to a nearby location while retaining possession ("dribbling"),

```
kick(P) :- has_ball(P).
```

or can pass the ball to a teammate,

```
pass(P,Q) :- has_ball(P), teammate(P,Q).
!has_ball(P) <- pass(P,Q).
has_ball(Q)  <- pass(P,Q).
```

[28] It is a convention in the IPTV dataset that team `a` is the one that takes possession first. If the starting team were decided by a coin flip, then we would use the "event groups" extension in section 4.A.6 to decide whether `init` causes `has_ball(a)` or `has_ball(b)`. This would allow us to learn the weight of the coin (for example, on the IPTV dataset, we would learn that the coin *always* chooses team `a`); or if we knew it was a fair coin, we could model that by declaring that certain parameters are 0.

[29] Notice that in our program, the possible kickoff events all have equal intensity, leading to a uniform distribution over players `a1`, . . . , `a11`. We will learn that this intensity is high, since the kickoff happens at a time close to 0.

or can score a goal,

```
goal(P) :- has_ball(P).
```

Scoring a goal instantly updates the database to transfer the ball to the other team,

```
!has_ball(P) <- goal(P).
has_ball(S)  <- goal(P), in_team(P,T), not_eq(T,S).
```

after which someone in the other team can kick off the ball and continue the game.

When a player `P` has the ball, a player `Q` in the other team can steal it:

```
steal(Q,P) :- has_ball(P), opponent(P,Q).
!has_ball(P) <- steal(Q,P).
has_ball(Q)  <- steal(Q,P).
```

In our experiments, we got the best results by declaring non-zero embeddings of both teams and players, such as

```
:- embed(team, 8).
:- embed(player, 8).
```

Since there are only two teams, the embeddings of the two teams jointly serve as a kind of global state—but one that may be smaller than the global state we would use for a simple NHP model. In our actual experiments (section 4.7.2), hyperparameter search (section 4.F.7) chose 32-dimensional NDTT embeddings, giving a total of 64 dimensions for the pair of teams. In contrast, it chose a 128-dimensional global state for the simple NHP baseline model.

Ideally, we would like the embedding $[\![\texttt{player(P)}]\!]$ to track our probability distribution over the state of the robot player, such as its latent position on the field and its latent energy level. We would also like the embedding of a team to track our probability distribution over the state of the team and the latent position of the ball. We do not observe these latent properties in our dataset. However, they certainly affect the progress of the game. For example, if two players pass or steal, they must be near each

other; so if we have pass(P,Q) and steal(R,Q) nearby in time, then by the triangle inequality, P and R must be close together, which raises the probability of steal(P,R). Changes in the mean and variance of these probability distributions are then tracked by updates and drift of the embeddings, with the variance generally decreasing when an event occurs (because it gives information) and increasing between events (because uncertainty about the latent changes accumulates over time, as in a drunkard's walk).

The team and player embeddings are launched at time 0 using the exogenous init event:

```
34  team(T)   <- init, in_team(P,T).
35  player(P) <- init, in_team(P,T).
```

A player's embedding is updated whenever that player participates in an event. We elected to reduce the number of parameters by sharing parameters not only across players, but also across similar kinds of events (this was also done by the prior work DyRep).

```
36  player(P) <- kickoff(P) :: individual.
37  player(P) <- kick(P)    :: individual.
38  player(P) <- goal(P)    :: individual.
39  player(P) <- pass(P,Q)  :: individual_agent.
40  player(Q) <- pass(P,Q)  :: individual_patient.
41  player(Q) <- steal(Q,P) :: individual_agent.
42  player(P) <- steal(Q,P) :: individual_patient.
```

The parameter sharing notation was explained in section 4.B. The above rules use the linguistic names "agent" and "patient" to refer to the player who acts and the player who is acted upon, respectively.

A team's embedding is also updated when any player acts. We could have done this by saying that the team's embedding pools over all of its players, so it is updated when they are updated,

```
43  team(T) :- player(P), in_team(P,T)
```

but instead we directly updated the team embeddings using update rules parallel to the ones above. For example, line 37 also has a variant that affects not the player P that kicked the ball, but that player's team T, as well as a second variant that affects the opposing team.

```
44  team(T) <- kick(P), in_team(P,T) :: team.
45  team(S) <- kick(P), in_team(P,T), not_eq(T,S) :: team_other.
```

We similarly have variants of lines 39–42:

```
46  team(T) <- pass(P,Q), in_team(P,T) :: team_agent.
47  team(S) <- pass(P,Q), in_team(P,T), not_eq(T,S) :: team_nonagent.

48  team(T) <- steal(P,Q), in_team(P,T) :: team_agent.
49  team(S) <- steal(P,Q), in_team(P,T), not_eq(T,S) :: team_nonagent.
```

Here "non-agent" refers to the team that does not contain the agent (in the case of line 47, it does not contain the patient either).

Finally, we can improve the model by enriching the dependencies. Earlier, we embedded the kick event using line 22, repeated here:

```
50  kick(P) :- has_ball(P).
```

But then the probability that robot player P kicks at time t (if it has the ball) would be constant with respect to both P and t. We want to make this probability sensitive to the states at time t of the player P, the player's team T, and the other team S. So we modify the rule to add those facts as conditions (in blue):

```
51  kick(P) :=
      has_ball(P), player(P), team(T), team(S), in_team(P,T),
      not_eq(T,S).
```

Because this rule uses := to request highway connections, all three of these states will also be consulted directly when a kick(P) event updates the states of player P

and both teams (via lines 22, 44 and 45). To deepen the network, we further give the event `kick(P)` its own embedding, which is a nonlinear combination of all of these states, and which is also consulted when the event causes an update.

```
52    :- event(kick,8).
```

We handle the other event types similarly to `kick`. In the case of an event that involves two players `P` and `Q`, we also add the state of player `Q` (the patient) as a fourth blue condition. For example, we expand the old line 23 to

```
53  pass(P,Q) :-
      has_ball(P), teammate(P,Q), player(P), player(Q), team(T),
      team(S), has_ball(P), in_team(P,T), not_eq(T,S).
```

4.F.6 Baseline Programs on RoboCup Dataset

As before, we also implemented baseline models that are inspired by the Know-Evolve and DyRep frameworks (Trivedi, p.c.). The non-embedded database facts about players and teams are specified just as in section 4.F.5 (lines 7–17).

Like the Know-Evolve program for IPTV, the Know-Evolve program for RoboCup has no embeddings for its events:

```
1   :- event(kickoff, 0).
2   :- event(kick, 0).
3   :- event(goal, 0).
4   :- event(pass, 0).
5   :- event(steal, 0).
```

As in IPTV, the embeddings are handled by separate facts. Know-Evolve's embedding of an event does not depend on the event's type, but only on its set of participants. Thus, the `kickoff`, `kick`, and `goal` events are simply represented by the embedding of the single player that participates in those events, which is defined exactly as in our full model of section 4.F.5:

```
6   :- embed(player, 8).
7   player(P) <- init, in_team(P,T).
```

For the pass and steal events, we also need an embedding for each *unordered pair* of players (analogous to watch_emb in section 4.F.4 line 4):

```
8   :- embed(players, 8).
9   players(P,Q) :- player(P) : pair, player(Q) : pair.
```

All of these embeddings evolve over time. Since teams do not participate directly in events, they do not have embeddings, in contrast to our full model in section 4.F.5.

Each event's probability depends nonlinearly on the concatenated embeddings of its participants, e.g.,

```
10  kick(P)    :- player(P).
11  pass(P,Q)  :- player(P), player(Q), teammate(P,Q).
```

Note that because Know-Evolve does not allow changes over time in the set of possible events, it assigns a positive probability to the above events even at times when P does not have the ball.

Actually, Trivedi et al. (2017) and Trivedi et al. (2019) allow any event to take place at any time between any pair of entities. Our Know-Evolve and DyRep programs take the liberty of going beyond this to impose some *static* domain-specific restrictions on which events are possible. For example, in RoboCup, line 11 only allows passing between teammates, and line 10 only allows kicking from a player to itself (i.e., the "pair" of participants for kick(P) has only one unique participant).

An event updates the embeddings of its participants, e.g.,

```
12  player(P) <- : kick kick(P), player(P) : only.
13  player(P) <- : pass pass(P,Q), players(P,Q) : agent.
14  player(Q) <- : pass pass(P,Q), players(P,Q) : patient.
```

where the bias vector is determined by the event type (e.g., `kick` or `pass`), while the weight matrix is determined by the role played in the event of the participant being updated (`agent`, `patient`, or `only`—see section 4.F.5). Both types of parameters are shared across multiple rules.

For the DyRep program, the same events are possible as for Know-Evolve, and most of the rules are the same. However, recall from section 4.F.4 that DyRep permits us to define a graph of entities. Robot players are entities, of course. We also consider the ball to be an entity, which is connected to player `P` by an edge when `P` possesses the ball. This allows DyRep to update the embeddings of the participants in a `pass` or `steal` event to record the fact that the one who had the ball now lacks it, and vice-versa. The model can therefore learn that `pass(P,Q)` and `steal(Q,P)` are much more probable when `P` has the ball.

DyRep requires the following new rules to handle the ball:

```
15  :- embed(ball, 8).
16  ball <- init.
```

as well as all of the rules from section 4.F.5 that update `has_ball`, which manage the edges of the evolving graph. Note that ⟦`ball`⟧ may drift over time but is never updated, since `ball` is never one of the participants in an event.

Now we mechanically obtain the DyRep model by replacing Know-Evolve rules such as lines 12–14 with DyRep-style versions:

```
17  player(P) <- : kick kick(P),   player(P) :: event.
18  player(P) <- : pass pass(P,Q), player(P) :: event.
19  player(Q) <- : pass pass(P,Q), player(Q) :: event.
```

and then mechanically adding influences from the neighbors of `P` and `Q` (where the ball is the only possible neighbor):

```
20  player(P) <- kick(P), ball : ball, has_ball(P).
```

```
21  player(P) <- pass(P,Q), ball : ball, has_ball(P).
22  player(Q) <- pass(P,Q), ball : ball, has_ball(Q).
```

Remarks. Recall that the DyRep model can unfortunately generate domain-impossible event sequences in which `P` kicks or passes the ball without actually having it. However, such events never happen in *observed* data. As a result, the above rules can be simplified if we are only updating embeddings based on observed events (which is true in our experiments). We can then remove the explicit `has_ball(P)` condition from lines 20 and 21 because it is surely true when these rules are triggered by observed events. And we can remove line 22 altogether, because its condition `has_ball(Q)` is surely false when this rule is triggered by an observed event. But then `has_ball` plays no role in the DyRep model anymore! This shows that in effect, the model tracks the ball's possessor only by updating `player(P)` whenever it observes an event with participant `P` in which `P` has the ball. This type of tracking is imprecise (in particular, it does not immediately detect when `P` *acquires* the ball), which is why the DyRep model cannot learn from data to assign probability ≈ 0 to domain-impossible events.

4.F.7 Training Details

For every model in section 4.7, including the baseline models, we had to choose the dimension D that is specified in the `embed` and `event` declarations of its NDTT program. For simplicity, all declarations within a given program used the same dimension D, so that each program had a single hyperparameter to tune. We tuned this hyperparameter separately for each combination of program, domain, and training size (e.g., each point in Figure 4.2 and each bar in Figures 4.3–4.5), always choosing the D that achieved the best performance on the dev set. Our search space was {4,

8, 16, 32, 64, 128}. In practice, the optimal D for a model of a non-synthetic dataset (section 4.7.2) was usually 32 or 64.

To train the parameters for a given D, we used the Adam algorithm (Kingma and Ba, 2015) with its default settings and set the minibatch size to 1. We performed early stopping based on log-likelihood on the held-out dev set.

Chapter 5

Attentive Models

The Transformer (Vaswani et al., 2017) has enjoyed remarkable success at learning representations for *discrete-time* sequences, such as natural language sentences. In this chapter, we generalize its success to *continuous-time* event sequences. We design the *attentive* version of the neural Hawkes process based on a **continuous-time Transformer**. An actual or possible event at time t is embedded using attention over actual events at times $< t$. More generally, we propose a new way to compute fact embeddings and event probabilities for neural Datalog through time. Our Transformers replace the LSTM-style mechanism to capture temporal dependencies. Our novel architectures match or surpass the previous versions—as well as other strong competitors—on several synthetic and real-world datasets, evaluated by likelihood and prediction accuracy.

5.1 An Overview of Continuous-Time Transformer

We propose an elegant way to embed events into a vector space $\mathbb{R}^D$. An actual or possible event at time $t \in \mathbb{R}$ is contextually embedded using attention over actual events at other times, in a way that considers their timing. Good event embeddings

could be used in a variety of downstream tasks. In particular, we investigate a left-to-right generative model of event sequences: at each time $t \in \mathbb{R}$, all of the possible events are embedded using masked attention over the actual events at times $< t$, and those embeddings yield the instantaneous probabilities of the possible events at time t.

Our approach is a *continuous-time* generalization of the Transformer (Vaswani et al., 2017). Transformers scale up well to very large datasets and have achieved astonishing success at generative modeling of *discrete-time* sequences, such as natural-language documents (Radford et al., 2019; Brown et al., 2020) and proteins (Rao et al., 2021). We show in this chapter that they are also effective for continuous-time sequences, even in lower-data regimes.

For generality, we show how to derive models from arbitrary neural Datalog through time (NDTT) programs. Recall from Chapter 4: an NDTT program specifies logical rules that derive facts within a deductive database and change them in response to events; the facts in the database at any time are embedded using a deep recurrent neural architecture that is automatically determined by the facts' provenance. Such an informed architecture can outperform a generic one, particularly under limited training data, by sharing parameters and making domain-appropriate conditional independence assumptions. We modify the NDTT architectures to use attention. A fact that can be derived by multiple rules uses multiple attention heads to examine the past.

The models in Chapters 3 and 4 summarize the past using recurrent neural networks (in particular, LSTMs). Our new attentive versions have three advantages over them:

① We do not summarize. Our predictions at time t can examine an unboundedly large representation of the past ($\mathbb{R}^d$ embeddings of *every* event before t), not just a fixed-dimensional summary that was computed greedily from left to right

(an RNN's state at time t).

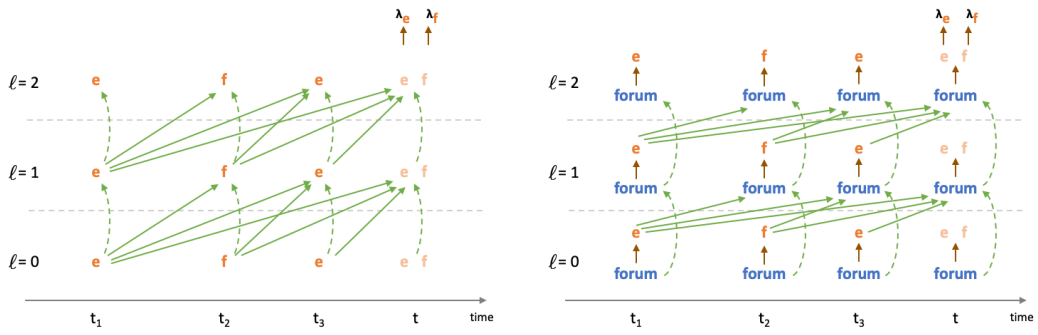
② Our architecture is broader but shallower, allowing greater parallelism. The layer-ℓ embeddings can be computed in parallel during training, as they depend only on layer $\ell - 1$ and not on one another.

③ Our architecture is simpler and more natural. To describe how embeddings vary continuously during the period $(t_i, t_{i+1}]$ between two events, we did not need to explicitly choose an arbitrary family of parametric decay functions on t (Mei and Eisner, 2017), nor design a complex method for pooling the parameters of these decay functions across multiple NDTT rules (Mei et al., 2020). Our embeddings at each time t are simply constructed "from scratch" by looking back at the set of previous events, using t-specific query vectors that include a continuous positional embedding of t. As t ranges over $(t_i, t_{i+1}]$, the set of previous events is fixed, and the attention weights vary continuously with t, so the embeddings do so too.

5.2 Continuous-Time Transformer to Embed Events

Suppose we observe I events over a fixed time interval $[0, T)$. Each event is denoted mnemonically as $e@t$ (i.e., "type e *at* time t"), where $e \in \{1, 2, \ldots, E\}$ is the event's type. The observed **event sequence** is $e_1@t_1, e_2@t_2, \ldots, e_I@t_I$, where $0 < t_1 < t_2 < \ldots < t_I < T$.

For any observed or possible event $e@t$, we can compute an **embedding** $[\![e]\!](t) \in \mathbb{R}^D$ by attending to a set $\mathcal{H}(e@t)$ of *relevant* events. (For the moment, imagine that $\mathcal{H}(e@t)$ consists of all the observed events $e_i@t_i$.) More precisely, $[\![e]\!](t)$ is the concatenation of layer-wise embeddings $[\![e]\!]^{(0)}(t), [\![e]\!]^{(1)}(t), \ldots, [\![e]\!]^{(L)}(t)$. For $\ell > 0$,



(a) A-NHP **(b)** A-NDTT

Figure 5.1: These figures show how embeddings in the model flow through layers (bottom to top) and through time (left to right). There are two possible event types, e and f, which represent email messages. At the upper right corner of each figure, we obtain their modeled intensities at a certain time t, $\lambda_{\mathtt{e}}(t)$ and $\lambda_{\mathtt{f}}(t)$, based on the embeddings of the three previous, irregularly spaced observed events. This requires embedding e and f at time t as if they were observed. If either one actually occurs at time t, we will keep its embeddings, which will then affect embeddings of events at times $> t$. Figure (a) shows the basic model of section 5.3, in which each event's embedding at layer ℓ depends (⟶) on all preceding events at layer $\ell - 1$. (The dashed arrows - -> reflect residual connections as well as the fact that each event or fact also attends to itself.) section 5.4 explains that the e ⟶ f influences can be prevented by dropping the rule f <- e. Figure (b) shows an A-NDTT model from section 5.5, in which the company forum's embedding at layer ℓ depends (⟶) on all preceding events at layer $\ell - 1$ (via <- rules). The events or possible events at layer ℓ do *not* depend directly on preceding events; instead, their embeddings at time t are derived (⟶) from the forum's embedding at time t (via :- rules).

the layer-ℓ embedding of $e@t$ is computed as

$$[\![e]\!]^{(\ell)}(t) \stackrel{\text{def}}{=} \underbrace{[\![e]\!]^{(\ell-1)}(t)}_{\text{residual connection}} + \tanh\left(\sum_{f@s \in \mathcal{H}(e@t)} \frac{\mathbf{v}^{(\ell)}(f@s)\ \alpha^{(\ell)}(f@s, e@t)}{\sum_{f@s \in \mathcal{H}(e@t)} \alpha^{(\ell)}(f@s, e@t)}\right) \tag{5.1}$$

where the unnormalized attention weight on each relevant event $f@s \in \mathcal{H}(k@t)$ is

$$\alpha^{(\ell)}(f@s, e@t) \stackrel{\text{def}}{=} \exp\left(\frac{1}{\sqrt{D}}\ \mathbf{k}^{(\ell)}(f@s)^\top \mathbf{q}^{(\ell)}(e@t)\right) \in \mathbb{R} \tag{5.2}$$

In layer ℓ, $\mathbf{v}^{(\ell)}$, $\mathbf{k}^{(\ell)}$, and $\mathbf{q}^{(\ell)}$ are known as the **value**, **key**, and **query** vectors and are

extracted from the layer-(ℓ-1) event embeddings using learned layer-specific matrices $\mathbf{V}^{(\ell)}, \mathbf{K}^{(\ell)}, \mathbf{Q}^{(\ell)}$:

$$\mathbf{v}^{(\ell)}(e@t) \stackrel{\text{def}}{=} \mathbf{V}^{(\ell)} \left[1; [\![t]\!]; [\![e]\!]^{(\ell-1)}(t)\right] \tag{5.3a}$$

$$\mathbf{k}^{(\ell)}(e@t) \stackrel{\text{def}}{=} \mathbf{K}^{(\ell)} \left[1; [\![t]\!]; [\![e]\!]^{(\ell-1)}(t)\right] \tag{5.3b}$$

$$\mathbf{q}^{(\ell)}(e@t) \stackrel{\text{def}}{=} \mathbf{Q}^{(\ell)} \left[1; [\![t]\!]; [\![e]\!]^{(\ell-1)}(t)\right] \tag{5.3c}$$

As the base case, $[\![e]\!]^{(0)}(t) \stackrel{\text{def}}{=} [\![e]\!]^{(0)}$ is a learned embedding of the event type e. $[\![t]\!]$ denotes an embedding of the time t; as proposed by Zuo et al. (2020), we use the same sinusoidal embedding as in Vaswani et al. (2017), but with t now being real instead of integer. We concatenate $[\![t]\!]$ to the rest of the embedding rather than adding it (cf. He et al., 2021); furthermore we do so again at every layer.

According to equations (5.1)–(5.3), to compute the layer-ℓ embedding of an event, we only need the layer-(ℓ−1) embeddings of the relevant events. This makes it possible to compute the layer-ℓ embeddings of all events in parallel.

The set of relevant events $\mathcal{H}(e@t)$ may be defined in a task-specific way. For example, to pretrain BERT-like embeddings (Devlin et al., 2018), we might use a corrupted version of $\{e_1@t_1, \ldots, e_I@t_I\}$ in which some $e_i@t_i$ have been removed or replaced with $\texttt{mask}@t_i$. Such embeddings could be pretrained with a BERT-like objective and then fine-tuned to predict properties of the observed events.

5.3 Generative Continuous-Time Transformer

In this chapter, we focus on the task of predicting future events given past ones. At any time t, we would like to know what will happen at that time, given the actual events that happened *before* t. Our generative model is analogous to a Transformer

language model (Radford et al., 2019; Brown et al., 2020), which, at each time $t \in \mathbb{N}$, defines a probability distribution over the words in the vocabulary.

In our setting, however, $t \in \mathbb{R}$. With probability 1, nothing happens at time t. Each possible event e in our vocabulary has only an infinitesimal probability of occurring at time t. We write this probability as $\lambda_e(t)\mathrm{d}t$ where $\lambda_e(t) \in \mathbb{R}$ is the intensity of type-e events at time t. More formally, the probability of such an event occurring during $[t, t+\epsilon)$ approaches $\lambda_e(t)\,\epsilon$ as $\epsilon \to^+ 0$.

We model $\lambda_e(t)$ as a function of the top-layer embedding of the *possible* event $e@t$:

$$\lambda_e(t) \stackrel{\text{def}}{=} \mathrm{softplus}(\mathbf{w}_e^\top [1; [\![e]\!]^L(t)], \tau_e) \tag{5.4}$$

where $\mathrm{softplus}(x, \tau) = \tau \log(1 + \exp(x/\tau)) > 0$ and $\mathbf{w}_e$ and $\tau_e > 0$ are learnable parameters. We do this separately for each $e@t$, computing the embedding $[\![e]\!]^L(t)$ using equations (5.1)–(5.3). Notice that this technique differs from a Transformer language model, which instead models a distribution over words at time t as $\mathrm{softmax}(\mathbf{W}[1; [\![e]\!]^L(t-1)])$ where $[\![e]\!]^L(t-1)$ is the top-layer embedding of the *previous* word.

To ensure that our model is generative, we compute $[\![e]\!]^L(t)$ from only the previous events. Thus, we take $\mathcal{H}(e@t)$ in equation (5.1) to consist of all the previously generated events $e_i@t_i$ for $t_i < t$, as well as the possible event $e@t$ itself. Including $e@t$ allows $e@t$ itself to draw most of the attention if none of the previous events are very relevant (e.g., they are too far in the past), or all of the attention in the special case where no previous events exist (i.e., it ensures that $\mathcal{H}(e@t) \neq \emptyset$, preventing division by 0 in (5.1)).

This model falls into the family of multivariate point processes. Its log-likelihood is given by

$$\sum_{i=1}^{I} \log \lambda_{e_i}(t_i) - \int_{t=0}^{T} \sum_{e=1}^{E} \lambda_e(t) \mathrm{d}t \tag{5.5}$$

This formula has been derived and discussed in Chapter 2. Intuitively, during parameter training, each $\log \lambda_{e_i}(t_i)$ is increased to explain why the observed event e_i happened at time t_i, while $\int_{t=0}^{T} \sum_{e=1}^{E} \lambda_e(t) \mathrm{d}t$ is decreased to explain why no event of any possible type k ever happened at other times. That is why we need to embed possible events and compute their intensities.

5.4 Multi-Head Selective Attention

We now enrich the formalism to allow *selective* attention. As in a graphical model, not all events should be able to influence one another. Consider a scenario with two event types: e means that Eve emails Adam, while f means that Frank emails Eve. As Frank does not know when Eve emailed Adam, past events of type e cannot influence his behavior. Therefore, $\mathcal{H}(\mathtt{f}@t)$ should include past events of type f but not e, so that the embedding of $\mathtt{f}@t$ and hence the intensity function $\lambda_{\mathtt{f}}(t)$ pay zero attention to e events. In contrast, $\mathcal{H}(\mathtt{e}@t)$ should still include past events of both types, since both are visible to Eve and can influence her behavior. We describe this situation with the edges f <- f, e <- e, e <- f. These edges are called **rules**, for reasons that will become clear in section 5.5 (or if you have read Chapter 4).

Furthermore, when Eve decides whether to email Adam ($\mathtt{e}@t$), we may reasonably suppose that she *separately* considers the embeddings of the past e events (e.g., "when was my last relevant email to Adam?") versus the past f events (e.g., "what have I heard from Frank?"). For this reason, we associate different attention heads with the

two rules that affect e, namely e <- e and e <- f. These heads may have different parameters, so that they seek out and obtain different information from the past via different queries, keys, and values.

In general, we replace equation (5.1) with

$$[\![e]\!]^{(\ell)}(t) \stackrel{\text{def}}{=} [\![e]\!]^{(\ell-1)}(t) + \tanh\left(\sum_r \boxed{e}_r^{(\ell)}(t)\right) \tag{5.6}$$

$$\boxed{e}_r^{(\ell)}(t) \stackrel{\text{def}}{=} \sum_{f@s \in \mathcal{H}_r(e@t)} \frac{\mathbf{v}_r^{(\ell)}(f@s)\ \alpha_r^{(\ell)}(f@s, e@t)}{\sum_{f@s \in \mathcal{H}_r(e@t)} \alpha_r^{(\ell)}(f@s, e@t)} \tag{5.7}$$

where r in the summation ranges over rules e <- $\cdots$. The history $\mathcal{H}_r(\underline{e}@t)$ contains only those past events $f@s$ that rule r makes visible to e, as well as $e@t$ itself as discussed in section 5.3. The r-specific α and $\mathbf{v}$ quantities are defined using versions of equations (5.2)–(5.3) with r-specific parameters.[1]

5.5 Attentive Neural Datalog Through Time

Edges such as e <- f can be regarded as simple NDTT rules. More generally, an NDTT program (Chapter 4) defines how facts should be automatically asserted into and retracted from a database, due to the occurrence of events and/or the presence of other facts.

Manually specifying such a program relieves the neural system of the burden of representing discrete facts about the world and learning how to modify them—which is properly the job of a database. However, if a proposition h appears as a fact in the NDTT database at time t, then it will also have an embedding $[\![h]\!](t)$—a *learned representation* of that fact at that time, capturing additional contextual properties.

[1] The NDTT formalism does allow parameters to be explicitly shared across rules when desired. Its :: notation (section 4.B) can also be used in the setup of this chapter.

Such a representation can be trained to be useful in downstream tasks.

In our setting, for example, if e denotes the time-varying boolean proposition that an event of type e is possible, then $[\![e]\!](t)$ will be defined at all times when that proposition is true—i.e., at all times when e is possible—and is used in equation (5.4) to model the intensity function λ_e, which governs when e is likely to actually occur.

Our goal in this chapter is to provide new formulas for the embeddings $[\![h]\!](t)$, based on Transformer-style attention rather than LSTM-style recurrence. We call this **attentive NDTT**, or **A-NDTT**. The potential advantages for accuracy, efficiency, and simplicity were explained in section 5.1.

We have seen simple "launch" rules so far, but more generally, NDTT allows three kinds of rules:

- h `:-` g_1, ..., g_n says to **deduce** h at any time t when g_1, ..., g_n are all true.
- h `<-` f, g_1, ..., g_n says to **launch** h at any time s when event f occurs and the g_i are all true.
- `!`h `<-` f, g_1, ..., g_n says to **dock** h at any time s satisfying the same conditions.

h is true (i.e., appears in the database) at time t if either (1) h is deduced at time t, or (2) h was launched at some time $s < t$ and never docked at any time in (s, t). The layer-ℓ embedding of h is then given, under our A-NDTT approach, by

$$[\![h]\!]^{(\ell)}(t) \stackrel{\text{def}}{=} [\![h]\!]^{(\ell-1)}(t) + \tanh\left([h]^{(\ell)}(t) + \sum_r \boxed{h}_r^{(\ell)}(t)\right) \tag{5.8}$$

which is a generalization of equation (5.6). For each rule r that launches h, say h `<-` $f, g_1, \ldots, g_n$, the intermediate vector $\boxed{h}_r^{(\ell)}(t)$ is computed as in equation (5.7),

but with two enhancements. First, $\mathcal{H}_r(h@t)$ contains past events only if (2) above holds for some s. It then contains just the past events of type f since the earliest such s—namely $e_i@t_i$ where $e_i = f$ and $t_i \in [s, t)$. It also contains $h@t$ itself, as before, even though h is not necessarily an event. Second, the r-specific versions of equations (5.3) now include all of the $[\![g_i]\!]^{(\ell-1)}$ in their concatenations of vectors, not just $[\![h]\!]^{(\ell-1)}$.

How about the intermediate vector $[h]^{(\ell)}(t)$ in equation (5.8)? For each rule that deduces h, say h :- $g_1, \ldots g_n$, we compute a nonlinear transform of the concatenated embeddings $[\![g_i]\!]^{(\ell)}$. The resulting vectors are pooled to obtain $[h]^{(\ell)}(t)$. (If there are no such rules, then $[h]^{(\ell)}(t) = \mathbf{0}$.) Here we follow exactly the recipe of Chapter 4, except that we now do it anew at each level ℓ.

Chapter 4 give many examples of NDTT programs. Here is a simple example to illustrate the use of :- rules. Possible events are shown in orange and other possible facts in blue. `e` means that Eve posts a message to the company forum, while `f` means that Frank does so. Once the forum is created by a `create` event, its existence is a fact (`forum`) whose embedding ($[\![\texttt{forum}]\!]$) reflects all messages posted to the forum. Until the forum is destroyed, Eve and Frank can post to it, and the embedding and intensity of such messages depends on the current state of the forum:

```
forum <- create.
forum <- e.
forum <- f.
e :- forum.
f :- forum.
!forum <- destroy.
```

The resulting neural architecture is drawn in Figure 5.1b. If the company grows from 2 to K employees, then the program needs $O(K)$ rules and hence $O(K)$ parameters,

which define how each employee's messages affect the forum and vice-versa. Without the :- rules, we would have to list out $O(K^2)$ rules such as `e <- f` and hence would need $O(K^2)$ parameters, which define how each employee's messages affect every other employee's messages directly; this case is drawn in Figure 5.1a.

Actually, if the company has many employees `E` and a forum for each team `T`, NDTT rules can use capitalized variables to define the whole system concisely, using only $O(1)$ rules and $O(1)$ parameters.[2] Here the possible facts and events have structured names like `message(eve,sales)`, which would be an event in which employee `eve` posts a message to the `sales` team's forum.

```
forum(T) <- create(T).
forum(T) <- message(E,T).
message(E,T) :- forum(T), member(E,T).
!forum(T) <- destroy(T).
```

Additional rules could describe how the `member` facts and their embeddings change over time, which may in turn affect the messages.

5.6 Training and Inference

For training and inference, we follow the general recipes in Chapter 2. One can also refer to section 4.5 for details regarding neural models specified by temporal logic programs.

5.7 Related Work

Multivariate point processes have been widely used in real-world applications, including document stream modeling (He et al., 2015; Du et al., 2015a), learning Granger causality (Xu, Farajtabar, and Zha, 2016; Zhang et al., 2020b), network analysis (Choi

[2] Although a notation is available to demand employee-specific and forum-specific parameters if desired.

et al., 2015; Etesami et al., 2016), recommendation systems (Du et al., 2015b), and social network analysis (Guo et al., 2015; Lukasik et al., 2016).

Over the recent years, various neural models have been proposed to expand the expressiveness of point processes. They mostly use recurrent neural networks, or LSTMs (Hochreiter and Schmidhuber, 1997): in particular Du et al. (2016), Mei and Eisner (2017), Xiao et al. (2017b), Xiao et al. (2017c), Omi, Ueda, and Aihara (2019), Shchur, Biloš, and Günnemann (2020), Mei et al. (2020), and Boyd et al. (2020). Models of this kind enjoy continuous and infinite state spaces, as well as flexible transition functions, thus achieving superior performance on many real-world datasets, compared to classical models such as the Hawkes process (Hawkes, 1971).

The **Transformer Hawkes process** (Zuo et al., 2020) and **self-attentive Hawkes process** (Zhang et al., 2020a) are pioneering work in using generative Transformers (Vaswani et al., 2017; Radford et al., 2019; Brown et al., 2020) in point processes. The Transformer architecture allows their models to enjoy unboundedly large representations of histories, as well as great parallelism during training (see ① and ② in section 5.1). But they only embed *actual* events with attention; the intensities of *possible* events rely on a simple linear extrapolation of the embedding of the most recent actual event.[3] Our model embeds all *possible* events at all *possible* times with attention as well (see ③ in section 5.1), thus being more flexible (see section 5.3) and achieving better performance (see section 5.8).

[3]The Transformer Hawkes process defines $\lambda_e(t) \overset{\text{def}}{=} \text{softplus}(\mathbf{w}_e^\top [1; t/t_i; [\![e_i]\!](t_i)])$ where $e_i @ t_i$ is the most recent event before t; the self-attentive Hawkes process conditions the linear projection $\mathbf{w}$ on $[\![e]\!](t_\text{i})$ as well.

5.8 Experiments

On several synthetic and real-world datasets, we evaluate our model—in comparison with multiple strong competitors—on the held-out log-likelihood, and on the success at predicting the time and type of the next event. Experimental details are given in section 5.A.

We implemented our A-NDTT framework using PyTorch (Paszke et al., 2017) and pyDatalog (Carbonell et al., 2016), borrowing substantially from the public implementation of NDTT. We then used it to implement our individual models.

5.8.1 Comparison of Different Transformer Architectures

We first verify the comparative advantages of our continuous-time Transformer over the following state-of-the-art neural event models. They are

Neural Hawkes process (NHP) (Mei and Eisner, 2017)**.** At any time t, NHP uses a continuous-time LSTM to summarize the previous events into a multi-dimensional state vector, and conditions the intensities of all event types on that state.

Transformer Hawkes process (THP) (Zuo et al., 2020)**.** THP directly applies the discrete-time generative Transformer, as discussed in section 5.7.

Self-Attentive Hawkes Process (SAHP) (Zhang et al., 2020a)**.** SAHP is very similar to THP, but its temporal drift coefficient is also contextual, as discussed in section 5.7.

For these models, we make use of their published implementation.[4] Details are in section 5.A.2.

[4]On some datasets, our replicated results are different from their papers. We confirmed that our results are correct via personal communication with the lead authors of Zhang et al. (2020a) and Zuo et al. (2020).

In this section, we use our unstructured generative model from section 5.3. It allows each possible event to look at all previous events with a shared set of attention parameters. This parameter-sharing mechanism resembles NHP, except that we now use a Transformer in place of an LSTM. So we call this model the **attentive neural Hawkes process (A-NHP)**.[5]

In a pilot experiment, we draw sequences from A-NHP (with random parameters), and fit all these models on this synthetic data. As shown in Figure 5.2, this data can not be well modeled by THP or SAHP, but it can be successfully modeled by—unsurprisingly—A-NHP and—maybe surprisingly—NHP. Notably, THP fits the time intervals poorly, due to its restrictive handling of the passage of time (see section 5.7). More details about this experiment can be found in section 5.A.1.1.

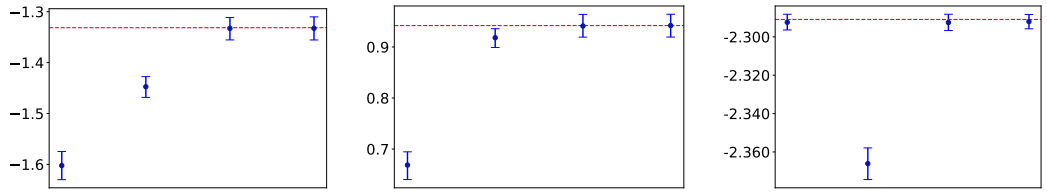
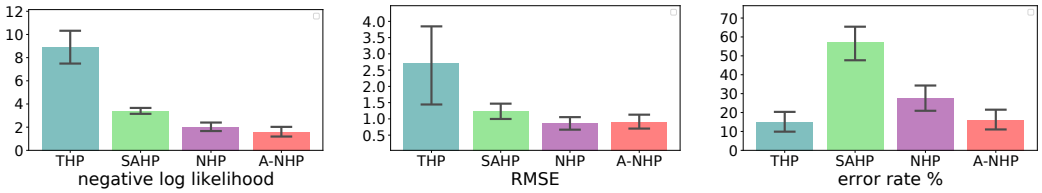


Figure 5.2: Log-likelihood (nats per event, with 95% bootstrap confidence intervals) of each model on held-out synthetic data. Larger values are better. Figures from left to right are log-likelihood on the entire sequence, time interval, and event type. Within each figure, the models (from left to right) are THP, SAHP, NHP, and A-NHP. The red dashed lines represent the log-likelihood of the model that generated the data. Note that log-likelihood for continuous variables can be positive, since it uses the log of a probability density that may be > 1.

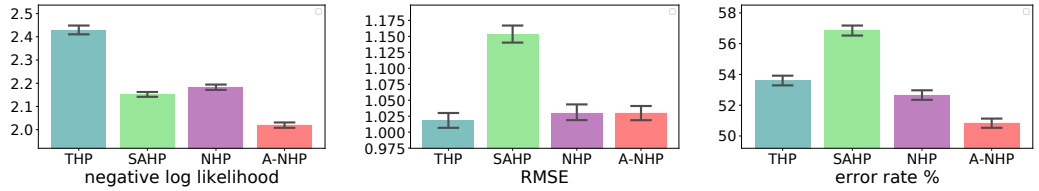
We then evaluated all these models on following two benchmark real-world datasets.

MIMIC-II (Lee et al., 2011)**.** This dataset is a collection of de-identified clinical visit records of Intensive Care Unit patients for 7 years.[6] Each patient has a

[5]For users' convenience, we made a standalone GPU-friendly PyTorch implementation of A-NHP.
[6]The documentation says: "Requirement for individual patient consent was waived [by the IRB]



(a) MIMIC-II



(b) StackOverflow

Figure 5.3: Evaluation results with 95% bootstrap confidence intervals on the real-world datasets of our A-NHP model vs. NHP, SAHP and THP. The RMSE is the root of mean squared error for predicted time. Error rate % denotes the fraction of incorrect predictions of the next event type, given the event time.

sequence of hospital visit events, and each event records its time stamp and disease diagnosis.

StackOverflow (Leskovec and Krevl, 2014)**.** This dataset represents two years of user awards on a question-answering website: each user received a sequence of badges (of 22 different types).

For these datasets, we used the preprocessed versions as in Chapters 3 and 4. More details about them can be found in section 5.A.1.2.

On MIMIC-II data (Figure 5.3a), our A-NHP is always a co-winner on each of these tasks; but the other co-winner (NHP or THP) varies across tasks. On StackOverflow data (Figure 5.3b), our A-NHP is clearly a winner on 2 out of 3 tasks.

Compared to NHP, A-NHP also enjoys a computational advantage, as discussed

because the project did not impact clinical care and all protected health information was deidentified."

in sections 5.1 and 5.2. Empirically, training an A-NHP only took a fraction of time that is needed to train an NHP, when sequences are reasonably long. Details can be found in Table 5.2 of section 5.A.3.

5.8.2 A-NDTT vs. NDTT

Now we evaluate A-NDTT vs. NDTT on the RoboCup dataset as in Chapter 4.

RoboCup (Chen and Mooney, 2008)**.** This dataset logs actions (e.g., `kick`, `pass`) of robot players during RoboCup Finals 2001–2004. The set of possible event types dynamically changes over time (e.g., only ball possessor can kick or pass) as the ball is frequently transferred between players (by passing or stealing). There are $K = 528$ event types over all time, but only about 20 of them are possible at any given time. For each history, we made minimum Bayes risk predictions of the next event's time, and that event's participant(s) given its time and action type.

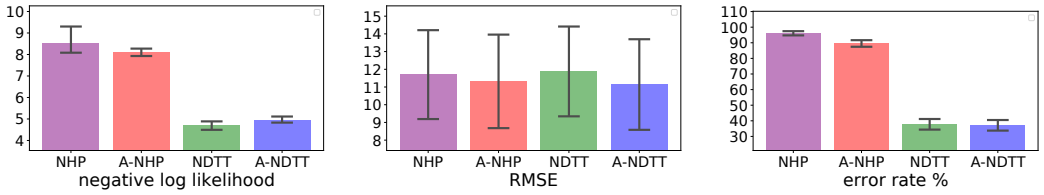


Figure 5.4: Evaluation results with 95% bootstrap confidence intervals on the RoboCup dataset. Evaluation methods are the same as in Figure 5.3.

On this dataset, we used the NDTT program provided in Chapter 4. The rules are unchanged; the only difference is that our A-NDTT has the new continuous-time Transformer in lieu of the LSTM architecture. We also evaluated NHP and A-NHP on this dataset.

The results are shown in Figure 5.4. As in section 5.8.1, we found that A-NHP performed better than NHP on all the evaluation metrics; the difference on log-likelihood

and event type prediction is significant. The NDTT program injects appropriate domain knowledge (e.g., only the ball possessor can kick or pass) into both LSTM and Transformer architectures, so both NDTT and A-NDTT substantially and significantly outperform A-NHP.

There is no significant difference between A-NDTT and NDTT. Perhaps both have roughly converged to the true predictive distribution. NDTT already fixes some of the problems with NHP, so it is not sorely in need of the Transformer's ability to scan an unbounded history for relevant events. (While NDTT still uses a fixed-dimensional history—① in section 5.1—the dimensionality is often very high, as the NDTT's state consists of embeddings of many individual facts. Moreover, each fact's NDTT embedding is computed by rule-specific LSTMs that see only events that are relevant to that fact, so there is no danger that intervening irrelevant events will displace the relevant ones in the fixed-dimensional LSTM states.) But while A-NDTT does not *improve* accuracy for this particular NDTT program and dataset, it does achieve *comparable* accuracy with a simpler and shallower architecture (②–③ in section 5.1) that could, as future work, be trained on a GPU with parallelism.

5.9 Conclusion

In this chapter, we showed how to generalize the Transformer architecture to continuous-time event sequences. Through our novel *continuous-time* Transformer, we can incrementally build up rich embeddings of actual and possible events at any time t, from lower-level representations of contextual events. The resulting probability model is highly flexible, and enjoys great parallelism during training. It outperforms other Transformer-based models on multiple synthetic and real-world datasets.

We also showed how to integrate this architecture with NDTT, a neural-symbolic

framework that automatically derives neural models from logic programs. Our new version has shown competitive performance, despite having a simpler and shallower architecture.

Appendices

5.A Experimental Details

5.A.1 Dataset Details

Table 5.1 shows statistics about each dataset that we use in this chapter.

5.A.1.1 Synthetic Data Details

In this experiment, we draw data from a randomly initialized 2-layer A-NHP, where the number of event is 10 and the dimension of time embeddings and event embeddings are both 32. We draw 800, 100, and 100 sequences for training, validation and testing, respectively. For each sequence, the sequence length is drawn from Uniform$(49, 99)$.

5.A.1.2 Other Data Details

For MIMIC-II and StackOverflow, we used the version processed by Du et al. (2016); more details (e.g., about processing) can be found in their chapter.

For RoboCup, we used the version processed by Chen and Mooney (2008); please refer to their paper for more details (e.g., data description, processing method, etc)

5.A.2 Implementation Details

For NHP, our implementation is based on the public Github repositories at `https://github.com/HMEIatJHU/neurawkes` (Mei and Eisner (2017), with MIT License) and `https://github.com/HMEIatJHU/neural-hawkes-particle-`

Dataset	K	# of Event Tokens			Sequence Length		
		Train	Dev	Test	Min	Mean	Max
Synthetic	10	59904	7425	7505	49	75	99
MIMIC-II	75	1930	252	237	2	4	33
StackOverflow	22	345116	38065	97233	41	72	736
RoboCup	528	2195	817	780	780	948	1336

Table 5.1: Statistics of each dataset.

`smoothing` (Mei, Qin, and Eisner (2019), with BSD 3-Clause "New" or "Revised" License). We made a considerable amount of modifications to their code (e.g., model, thinning algorithm), in order to migrate it to PyTorch 1.7. We built the standalone GPU implementation of A-NHP upon our NHP code.

For NDTT, we use the public Github repository at `https://github.com/HMEIatJHU/neural-datalog-through-time` (Mei et al. (2020), with MIT License). We built A-NDTT upon NDTT.

For THP, we use the public Github repository at `https://github.com/SimiaoZuo/Transformer-Hawkes-Process` (Zuo et al. (2020), no license specified).

For SAHP, we use the public Github repository at `https://github.com/QiangAIResearcher/sahp_repo` (Zhang et al. (2020a), no license specified).

5.A.3 Training Details

For each model in section 5.8, including the baseline models, we had to choose the dimension of their neural state (i.e., D); for the attention-based models, we had to choose the number of layers (i.e., L) in the attention mechanism as well. We tuned these hyperparameters for each combination of model, dataset, and training size (e.g., each bar in Figures 5.2–5.4), always choosing the combination of D

Dataset	Training Time (milliseconds) / Sequence	
	NHP	A-NHP
Synthetic	208.7	56.3
MIMIC-II	2.9	32.6
StackOverflow	156.6	65.7

Table 5.2: Training time of NHP and A-NHP for experiments in section 5.8.1.

and L that achieved the best performance on the dev set. Our search spaces were $D \in \{4, 8, 16, 32, 64, 128\}$ and $L \in \{1, 2, 3, 4, 5, 6\}$. In practice, the optimal D for a model was usually 32 or 64; the optimal L was usually 1, 2, or 3.

To train the parameters for a given model, we used the Adam algorithm (Kingma and Ba, 2015) with its default settings. We performed early stopping based on log-likelihood on the held-out dev set.

We also tried adding feed-forward layers and layer normalization (as in Vaswani et al. (2017)) in our preliminary experiments, but they didn't help.

For the experiments in section 5.8.1, we used the standalone PyTorch implementations for NHP and A-NHP, which are GPU-friendly. We trained each model on an NVIDIA K80 GPU. Table 5.2 shows their training time per sequence on each dataset.

We run our NDTT and A-NDTT models only on CPUs. This follows Chapter 4, where we did not find an efficient method to leverage GPU parallelism for training NDTT models. The machines we used for NDTT and A-NDTT are 6-core Haswell architectures. On RoboCup, the training time of NDTT and A-NDTT is 62 and 149 seconds per sequence, respectively. We used the same CPU implementation for NHP and A-NHP on RoboCup, rather than the standalone GPU implementation, since the RoboCup sequences are too long to fit in the memory of the K80 GPU. We did

this by expressing these models as NDTT programs. The NDTT program for NHP can be found in section 4.F.2; the A-NDTT program for A-NHP can be found in section 5.A.5.1 below. The training time is 66 and 95 seconds per sequence for NHP and A-NHP, respectively.

5.A.4 Training Parallelism

We point out that in the future, GPU parallelism could be exploited through the following procedure, given a GPU with enough memory to handle long training sequences. (The layers can be partitioned across multiple GPUs if needed.) For each training minibatch, the first step is to play each event sequence $e_1@t_1, e_2@t_2, \ldots, e_I@t_I$ forward to determine the contents of the database on each of the intervals $(0, t_1]$, $(t_1, t_2], \ldots, (t_{i-1}, t_I], (t_I, T]$. This step runs on CPU, and computes only the boolean facts (Datalog through time) without their embeddings (neural Datalog through time). Let $\mathcal{F}$ be the set of facts that ever appeared in the database during this minibatch and let $\mathcal{R}$ be the set of rules that were ever used to deduce or launch them (section 5.5).

A computation graph of size $O(|\mathcal{R}| \cdot I)$ can now be constructed, as illustrated in Figure 5.1b, to compute the embeddings $[\![h]\!](t)$ of all facts $h \in \mathcal{F}$ at all event times $t = t_i$ as well as all times t that are sampled for the Monte Carlo integral (section 2.2). The layer-ℓ embeddings at time t depend on the layer-$\ell - 1$ embeddings at times $s \leq t$, according to the launch rules in $\mathcal{R}$. The layer-ℓ embedding of a fact that is deduced at time t also depends on the layer-ℓ embeddings at time t of the facts that it is deduced from, according to the deduction rules in $\mathcal{R}$; this further increases the depth of the computation graph.

For a given fact $h \in \mathcal{F}$, $[\![h]\!]^{(\ell)}(t)$ can be computed in parallel for *all event sequences* and *all times* t (even times t when h is not true, although those embeddings

will not be used). Multiple facts that are governed by the same NDTT rule can also be handled in parallel. Thus, a GPU can be effective for this phase. The computation of $\boxed{h}_r^{(\ell)}(t)$ in equation (5.8) must take care to limit its attention to just those earlier times when an event occurred that launched h via rule r, and the computation of $[h]^{(\ell)}(t)$ in equation (5.8) must take care to consider only rules r that in fact deduce h at time t because their conditions are true at time t. This masks unwanted parts of the computation, rendering parts of the GPU idle. GPU parallelism will still be worthwhile if a substantial fraction of the computation remains unmasked—which is true for relatively homogenous settings where most facts in $\mathcal{F}$ hold true for a large portion of the observed interval $[0, T)$, even if their embeddings fluctuate.

5.A.5 A-NDTT Programs

5.A.5.1 A-NHP Datalog Program

In this section, we show how to specify A-NHP (section 5.3) by an A-NDTT program:

```
is_type(1).
  ⋮
is_type(N).
:- embed(world, 8).
:- event(e, 8).
e(N) :- world, is_type(N) :: prob(N).
world <- init.
world <- e(N).
```

where the dashed underline for `init` indicates that it is an "exogenous" event that is not predicted by our model. The `init` event always happens at time $t = 0$. The `::` notation was introduced in section 4.B.

This program is different from the NDTT program that specifies the NHP model: in that program, rule-7 will be replaced by a slightly fancier version, i.e., `world <- e(N), world` (see section 4.F.2). That is because the `world` as an additional condition

is necessary to resemble the traditional structure of an LSTM: the new LSTM cell of `world` is a linear combination of its old value and some input, where the linear coefficients (i.e., input and forget gates) and the input depend on both `e(N)` and the old value of `world`. Had we just used `world <- e(N)` instead, the coefficients and the input would only depend on `e(N)`, giving a less rich recurrence. However, this additional richness is not necessary in our A-NHP because our multi-layer attention already captures sufficiently rich contextual information (hence "attention is all you need").

Chapter 6

Efficient Training: Noise-Contrastive Estimation

In Chapter 3 and Chapter 4, we trained the models by maximum likelihood estimation, a popular training method for generative models. Their likelihood is improved not only by raising the probability of the observed events, but by lowering the probabilities of the events that were observed *not* to occur. There are infinitely many times at which no event of any type occurred; to predict these *non*-occurrences, the likelihood must integrate the infinitesimal event probability for each event type over the entire observed time interval. Therefore, the likelihood is expensive to compute, particularly when there are many possible event types.

In this chapter, we show how to instead apply a version of noise-contrastive estimation—a general parameter estimation method with a less expensive stochastic objective. Our specific instantiation of this general idea works out in an interestingly non-trivial way and has provable guarantees for its optimality, consistency and efficiency. On several synthetic and real-world datasets, our method shows benefits: for the model to achieve the same level of log-likelihood on held-out data, our method needs considerably fewer function evaluations and less wall-clock time.

6.1 A Review of Maximum Likelihood Estimation

Recall from section 2.1 that, to ensure that we have a point process, the intensity functions must be chosen such that the total number of events on any bounded interval is almost surely finite. Models of this form include Poisson processes (section 2.1.1), in which the intensity functions ignore the history, as well as Hawkes processes (section 2.1.2), and their modern neural versions, such as our models presented in the previous chapters. Most models use intensity functions that are continuous between events. Our analysis requires only

Assumption 1 (Continuity)**.** *For any event sequence* $x_{[0,T)}$ *and event type* $k \in \{1, \dots, K\}$, $\lambda_k(t \mid x_{[0,t)})$ *is Riemann integrable, i.e., bounded and continuous almost everywhere w.r.t. time* t.

In practice, we parameterize the intensity functions by $\boldsymbol{\theta}$. We write $p_{\boldsymbol{\theta}}$ for the resulting probability density over event sequences. When learning $\boldsymbol{\theta}$ from data, we make the conventional assumption that the true point process p^* actually falls into the chosen model family:

Assumption 2 (Existence)**.** *There exists at least one parameter vector* $\boldsymbol{\theta}^*$ *such that* $p_{\boldsymbol{\theta}^*} = p^*$.

Then as proved in section 6.A, such a $\boldsymbol{\theta}^*$ can be found as an argmax of

$$J_{\mathrm{LL}}(\boldsymbol{\theta}) \stackrel{\text{def}}{=} \mathbb{E}_{x_{[0,T)} \sim p^*}[\log p_{\boldsymbol{\theta}}(x_{[0,T)})] \tag{6.1}$$

Given assumption 1, the $\boldsymbol{\theta}$ values that maximize $J_{\mathrm{LL}}(\boldsymbol{\theta})$ are exactly the set Θ^* of values for which $p_{\boldsymbol{\theta}} = p^*$: any $\boldsymbol{\theta}$ for which $p_{\boldsymbol{\theta}} \neq p^*$ would end up with a strictly

smaller $J_{\mathrm{LL}}(\boldsymbol{\theta})$ by increasing the cross entropy $-p^* \log p_{\boldsymbol{\theta}}$ over some interval (t, t') for a set of histories with non-zero measure.

If we modify equation (6.1) to take the expectation under the empirical distribution of event sequences $x_{[0,T)}$ in the training dataset, then $J_{\mathrm{LL}}(\boldsymbol{\theta})$ is proportional to the log-likelihood of $\boldsymbol{\theta}$. For any $x_{[0,T)}$ that satisfies the condition in assumption 1, the log-density used in equation (6.1) can be expressed in terms of $\lambda_k(t \mid x_{[0,t)})$, just like equation (2.6) in section 2.2. For convenient reference, we have a copy here:

$$\log p_{\boldsymbol{\theta}}(x_{[0,T)}) = \sum_{t:x_t \neq \varnothing} \log \lambda_{x_t}(t \mid x_{[0,t)}) - \int_{t=0}^{T} \sum_{k=1}^{K} \lambda_k(t \mid x_{[0,t)}) \mathrm{d}t \tag{6.2}$$

It is *expensive* to compute in the following cases:

- The total number of event types K is large, making $\sum_{k=1}^{K}$ slow.
- The integral $\int_{t=0}^{T}$ is slow to estimate well, e.g., via Monte Carlo approximation $\frac{T}{J} \sum_{j=1}^{J} \sum_{k=1}^{K} \lambda_k(t_j)$ where each t_j is randomly sampled from the uniform distribution over $[0, T)$; see section 2.2.
- The chosen model architecture makes it hard to parallelize the $\lambda_k(t_j)$ computation over j and k. E.g., under an NDTT model, intensities of different event types may be computed via different parametric functions, which take as input different subsets of previous events; see Chapter 4.

6.2 Noise-Contrastive Estimation in Discrete Time

For autoregressive models of *discrete-time* sequences, a similar computational inefficiency can be tackled by applying the principle of noise-contrastive estimation (Gutmann and Hyvärinen, 2010), as follows. For each history $x_{0:t} \stackrel{\text{def}}{=} x_0 x_1 \dots x_{t-1}$ in training data, NCE trains the model $p_{\boldsymbol{\theta}}$ to discriminate the actually observed datum x_t from some noise samples whose distribution q is known. The intuition is: optimal

performance is obtained *if and only if* $p_{\boldsymbol{\theta}}$ matches the true distribution p^*.

More precisely, given a bag $\{x_t^0, x_t^1, \ldots, x_t^M\}$, where exactly one element of the bag was drawn from p^* and the rest drawn i.i.d. from q, consider the log-posterior probability (via Bayes' Theorem[1]) that x_t^0 was the one drawn from p^*:

$$\log \frac{p^*(x_t^0 \mid x_{0:t}) \prod_{m=1}^{M} q(x_t^m \mid x_{0:t})}{\sum_{m=0}^{M} p^*(x_t^m \mid x_{0:t}) \prod_{m' \neq m} q(x_t^{m'} \mid x_{0:t})} \tag{6.3}$$

The "ranking" variant of NCE (Jozefowicz et al., 2016) substitutes $p_{\boldsymbol{\theta}}$ for p^* in this expression, and seeks $\boldsymbol{\theta}$ (e.g., by stochastic gradient ascent) to maximize the expectation of the resulting quantity when x_t^0 is a random observation in training data,[2] $x_{0:t}$ is its history, and $x_t^1, \ldots, x_t^M$ are drawn i.i.d. from $q(\cdot \mid x_{0:t})$.

This objective is really just conditional maximum log-likelihood on a supervised dataset of $(M+1)$-way classification problems. Each problem presents an unordered set of $M+1$ samples—one drawn from p^* and the others drawn i.i.d. from q. The task is to guess *which* sample was drawn from p^*. Conditional MLE trains $\boldsymbol{\theta}$ to maximize (in expectation) the log-probability that the model assigns to the correct answer. In the infinite-data limit, it will find $\boldsymbol{\theta}$ (if possible) such that these log-probabilities *match* the true ones given by equation (6.3). For that, it is *sufficient* for $\boldsymbol{\theta}$ to be such that $p_{\boldsymbol{\theta}} = p^*$. Given assumption 2, Ma and Collins (2018) show that $p_{\boldsymbol{\theta}} = p^*$ is also *necessary*, i.e., the NCE task is sufficient to find the true parameters. Although the NCE objective does not learn to predict the full observed sample x_t as MLE does, but only to distinguish it from the M noise samples, their theorem implies that

[1] The product $p^*(x_t^m \mid x_{0:t}) \prod_{m' \neq m} q(x_t^{m'} \mid x_{0:t})$ is the likelihood of x_t^m being the one drawn from p^*. The prior is uniform since any m in the unordered bag was *a priori* equally probable.

[2] In practice, it is more convenient to maximize the expected *sum* over t in a sequence drawn uniformly from the set of sequences in the training dataset. This scales the objective up by the average sequence length, preserving the property that longer sequences have more weight.

in expectation over all possible sets of M noise samples, it actually retains all the information (provided that $M > 0$ and q has support everywhere that p^* does).

This NCE objective is computationally cheaper than MLE when the distribution $p_{\boldsymbol{\theta}}(\cdot \mid x_{0:t})$ is a softmax distribution over $\{1, \ldots, K\}$ with large K. The reason is that the expensive normalizing constants in the numerator and denominator of equation (6.3) need not be computed. They cancel out because all the probabilities are conditioned on the same (actually observed) history.

6.3 Noise-Contrastive Estimation in Continuous Time

The expensive $\int \sum$ term in equation (6.2) is rather similar to a normalizing constant,[3] as it sums over non-occurring events. We might try to avoid computing it[4] by discretizing the time interval $[0, T)$ into finitely many intervals of width Δ and applying NCE. In this case, we would be distinguishing the true sequence of events on an interval $[i\Delta, (i+1)\Delta)$ from corresponding noise sequences on the same interval, given the same (actually observed) history $x_{[0,i\Delta)}$. Unfortunately, the distribution $p_{\boldsymbol{\theta}}(\cdot \mid x_{[0,i\Delta)})$ in the objective still involves an $\int \sum$ term where the integral is over $[i\Delta, (i+1)\Delta)$ and the inner sum is over k.

[3] Our model does not need any normalization: $p(x_t = \varnothing) + \sum_{k=1}^{K} p(x_t = k) = 1 +$ (infinitesimal quantities) $= 1$.

[4] While this chapter's speedup over the MLE objective equation (6.2) comes from avoiding the integral, an alternative would be to estimate the integral more efficiently. One might try randomized adaptive quadrature (Baran, Demaine, and Katz, 2008) modified for our discontinuous intensity functions and GPU hardware; or importance sampling of (t, k) pairs where the proposal distribution is roughly proportional to $\lambda_k(t)$—much like the noise distribution we will develop for NCE.

6.3.1 Our Training Objective

Our solution is to shrink the intervals to *infinitesimal width* dt. Then the log-posterior over each of them becomes

$$\log \frac{p_{\boldsymbol{\theta}}(x^0_{[t,t+\mathrm{d}t)} \mid x^0_{[0,t)}) \prod_{m=1}^{M} q(x^0_{[t,t+\mathrm{d}t)} \mid x^0_{[0,t)})}{\sum_{m=0}^{M} p_{\boldsymbol{\theta}}(x^m_{[t,t+\mathrm{d}t)} \mid x^0_{[0,t)}) \prod_{m' \neq m} q(x^{m'}_{[t,t+\mathrm{d}t)} \mid x^0_{[0,t)})} \tag{6.4}$$

We will define the noise distribution q in terms of finite intensity functions λ_k^{q}, like the ones λ_k that define $p_{\boldsymbol{\theta}}$. As a result, at a *given* time t, there is only an infinitesimal probability that *any* of $\{x_t^0, x_t^1, \ldots, x_t^M\}$ is an event. Nonetheless, at *each* time $t \in [0, T)$, we will consider generating a noise event (for each $m > 0$) conditioned on the actually observed history $x_{[0,t)}$. Among these uncountably many times t, we may have some for which $x_t^0 \neq \varnothing$ (the observed events), or where $x_t^m \neq \varnothing$ for some $1 \leq m \leq M$ (the noise events).

Almost surely, the set of times t with a real or noise event remains finite. Our NCE objective is the expected sum of equation (6.4) over all such times t in an event sequence, when the sequence is drawn uniformly from the set of sequences in the training dataset—as in footnote 2—and the noise events are then drawn as above.

Our objective ignores all other times t, as they provide no information about $\boldsymbol{\theta}$. After all, when $x_t^0 = \cdots = x_t^M = \varnothing$, the probability that x_t^0 is the one drawn from the true model must be $1/(M+1)$ by symmetry, regardless of $\boldsymbol{\theta}$. At these times, the ratio in equation (6.4) does reduce to $1/(M+1)$, since all probabilities are 1.

At the times t that we do consider, how do we compute equation (6.4)? Almost surely, exactly one of $x_t^0, \ldots, x_t^M$ is an event k for some $k \neq \varnothing$. As a result, exactly one factor in each product is infinitesimal (dt times the λ_k or λ_k^{q} intensity), and the other factors are 1. Thus, the dt factors cancel out between numerator and denominator,

and equation (6.4) simplifies to

$$\log \frac{\lambda_k(t \mid x^0_{[0,t)})}{\lambda_k(t \mid x^0_{[0,t)}) + M\lambda^{\mathrm{q}}_k(t \mid x^0_{[0,t)})} \qquad \text{if } x^0_t = k \text{ for some } k \in \{1, \ldots, K\} \tag{6.5a}$$

$$\log \frac{\lambda^{\mathrm{q}}_k(t \mid x^0_{[0,t)})}{\lambda_k(t \mid x^0_{[0,t)}) + M\lambda^{\mathrm{q}}_k(t \mid x^0_{[0,t)})} \qquad \text{if } x^0_t = \varnothing \tag{6.5b}$$

When a gradient-based optimization method adjusts $\boldsymbol{\theta}$ to increase equation (6.5), the intuition is as follows. If $x^0_t = k$, the model intensity $\lambda_k(t)$ is *increased* to explain why an event of type k occurred at this particular time t. If $x^0_t = \varnothing$, the model intensity $\lambda_k(t)$ is *decreased* to explain why an event of type k did *not* actually occur at time t (it was merely a noise event $x^m_t = k$, for some $m \neq 0$). These cases achieve the same qualitative effects as following the gradients of the first and second terms, respectively, in the log-likelihood equation (6.2).

Our full objective is an expectation of the sum of finitely many such log-ratios:[5]

$$J_{\mathrm{NC}}(\boldsymbol{\theta}) \stackrel{\text{def}}{=} \mathbb{E}_{x^0_{[0,T)} \sim p^*, x^{1:M}_{[0,T)} \sim q}\Big[\sum_{t: x^0_t \neq \varnothing} \log \frac{\lambda_{x^0_t}(t \mid x^0_{[0,t)})}{\underline{\lambda}_{x^0_t}(t \mid x^0_{[0,t)})} + \sum_{m=1}^{M} \sum_{t: x^m_t \neq \varnothing} \log \frac{\lambda^{\mathrm{q}}_{x^m_t}(t \mid x^0_{[0,t)})}{\underline{\lambda}_{x^m_t}(t \mid x^0_{[0,t)})} \Big] \tag{6.6}$$

where $\underline{\lambda}_k(t \mid x^0_{[0,t)}) \stackrel{\text{def}}{=} \lambda_k(t \mid x^0_{[0,t)}) + M\lambda^{\mathrm{q}}_k(t \mid x^0_{[0,t)})$. The expectation is estimated by sampling: we draw an observed sequence $x^0_{[0,T)}$ from the training dataset, then draw noise events $x^{1:M}_{[0,T)}$ from q conditioned on the prefixes (histories) given by this observed sequence, as explained in the next section. Given these samples, the bracketed term is easy to compute (and we then use backprop to get its gradient w.r.t. $\boldsymbol{\theta}$, which is a stochastic gradient of the objective in equation (6.6)). It eliminates the $\int \sum$ of equation (6.2) as desired, replacing it with a sum over the noise events. For each real

[5] We remark that $J_{\mathrm{NC}}(\boldsymbol{\theta})$ is the expected log-*probability* of a discrete choice, whereas $J_{\mathrm{LL}}(\boldsymbol{\theta})$ was the expected log-*density* of an observation that includes continuous times. A density must be integrated to yield a probability.

or noise event, we compute only two intensities—the true and noise intensities of that event type at that time.

6.3.2 Relation to Generative Adversarial Networks

Our NCE framework is much like the generative adversarial network (GAN) (Goodfellow et al., 2014): the generator (noise distribution q) generates fake data (noise events and non-events), and the discriminator (constructed with model $p_{\boldsymbol{\theta}}$ and noise distribution q) learns to discriminate the actually observed events and non-events against the fake data by maximizing equation (6.6).

Our goal is to train the model $p_{\boldsymbol{\theta}}$, which is part of the discriminator; that is a little different from GAN, which primarily aims to train the generator. In principle, one can take the noise distribution q to be the model $p_{\boldsymbol{\theta}}$ with the current value of $\boldsymbol{\theta}$, which keeps improving during training; that is more like GAN since the generator is now what we aim to train. However, this design has a couple of computational issues: first, sampling from $p_{\boldsymbol{\theta}}$ is often expensive, and that is why we propose a specific class of noise models in section 6.3.3; second, one may have to redraw noise samples as $p_{\boldsymbol{\theta}}$ improves, and that will slow down the training as shown in section 6.5.1.

As in GAN, we expect the discrimination task to be most challenging and informative when the noise distribution q is close to the true data distribution p^*—or more precisely, when the noise intensity λ_k^{q} at time t is close to the true intensity $\lambda_k^*(t \mid x_{[0,t)}^0)$. Therefore we give the function λ_k^{q} access to the true history $x_{[0,t)}^0$, and will train it to predict something like the true intensity. Section 6.3.3 will discuss other benefits of conditioning t on the true history $x_{[0,t)}^0$.

6.3.3 Efficient Sampling of Noise Events

The thinning algorithm is a standard method for drawing an event sequence over a given observation interval $[0, T)$ from a continuous-time autoregressive process. A full recipe can be found in section 2.3; we recap it here for convenient reference. Suppose we have already drawn the first $i-1$ times, namely $t_1, \ldots, t_{i-1}$. For every future time $t \geq t_{i-1}$, let $\mathcal{H}(t)$ denote the context $x_{[0,t)}$ consisting only of the events at those times, and define $\lambda(t \mid \mathcal{H}(t)) \stackrel{\text{def}}{=} \sum_{k=1}^{K} \lambda_k(t \mid \mathcal{H}(t))$. If $\lambda(t \mid \mathcal{H}(t))$ were constant at $\overline{\lambda}$, we could draw the next event time as $t_i \sim t_{i-1} + \text{Exp}(\overline{\lambda})$. We would then set $x_t = \varnothing$ for all of the intermediate times $t \in (t_{i-1}, t_i)$, and finally draw the type x_{t_i} of the event at time t_i, choosing k with probability $\lambda_k(t_i \mid \mathcal{H}(t)) \,/\, \overline{\lambda}$. But what if $\lambda(t \mid \mathcal{H}(t))$ is not constant? The thinning algorithm still runs the foregoing method, taking $\overline{\lambda}$ to be any upper bound: $\overline{\lambda} \geq \lambda(t \mid \mathcal{H}(t))$ for all $t \geq t_{i-1}$. In this case, there may be "leftover" probability mass not allocated to any k. This mass is allocated to $\varnothing$. A draw of $x_{t_i} = \varnothing$ means there was no event at time t_i after all (corresponding to a rejected proposal). Either way, we now continue on to draw t_{i+1} and $x_{t_{i+1}}$, using a version of $\mathcal{H}(t)$ that has been updated to include the event or non-event x_{t_i}. The update to $\mathcal{H}(t)$ affects $\lambda(t \mid \mathcal{H}(t))$ and the choice of $\overline{\lambda}$.

How to Sample Noise Sequences. To draw a sequence $x^m_{[0,t)}$ of noise events, we run the thinning algorithm, using the noise intensity functions λ^{q}_k. However, there is a modification: $\mathcal{H}(t)$ is now defined to be $x^0_{[0,t)}$—the history from the *observed* event sequence, rather than the previously sampled *noise* events—and is updated accordingly. This is because in equation (6.6), at each time t, all of $\{x^0_t, x^1_t, \ldots, x^M_t\}$ are conditioned on $x^0_{[0,t)}$ (akin to the discrete-time case). This is not essential to the NCE approach, since in principle the $M+1$ elements of the bag could all be drawn

from different distributions. However, the homogeneity simplifies equations (6.5)–(6.6), and not having to keep track of previous noise samples simplifies bookkeeping. Furthermore, as discussed in section 6.3.2, it helps make q to be close to the true data distribution p^*. The full pseudocode is given in Algorithm 6.1 in the supplementary material.

Coarse-to-Fine Sampling of Event Types. Although our NCE method has eliminated the need to integrate over t, the thinning algorithm above still sums over k in the definition of $\lambda^{\mathrm{q}}(t \mid \mathcal{H}(t))$. For large K, this sum is expensive if we take the noise distribution on each training minibatch to be, for example, the $p_{\boldsymbol{\theta}}$ with the current value of $\boldsymbol{\theta}$. That is a *statistically* efficient choice of noise distribution, but we can make a more *computationally* efficient choice. A simple scheme is to first generate each noise event with a coarse-grained type $c \in \{1, \ldots, C\}$, and then stochastically choose a refinement $k \in \{1, \ldots, K\}$:

$$\lambda_k^{\mathrm{q}}(t \mid x_{[0,t)}^0) \stackrel{\text{def}}{=} \sum_{c=1}^{C} q(k \mid c) \lambda_c^{\mathrm{q}}(t \mid x_{[0,t)}^0) \text{ for } k = 1, 2, \ldots, K \tag{6.7}$$

This noise model is parameterized by the functions λ_c^{q} and the probabilities $q(k \mid c)$. The total intensity is now $\lambda^{\mathrm{q}}(t \mid \mathcal{H}(t)) = \sum_{c=1}^{C} \lambda_c^{\mathrm{q}}(t)$, so we now need to examine only C intensity functions, not K, to choose $\overline{\lambda}$ in the thinning algorithm. If we *partition* the K types into C coarse-grained clusters (e.g., using domain knowledge), then evaluating the noise probability in equation (6.7) within the training objective equation (6.6) is also fast because there is only one non-zero summand c in equation (6.7). This simple scheme works well in our experiments. However, it could be elaborated by replacing $q(k \mid c)$ with $q(k \mid c, x_{[0,t)}^0)$, by partitioning the event vocabulary automatically, by allowing overlapping clusters, or by using multiple levels of refinement: all of these

elaborations are used by the fast hierarchical language model of Mnih and Hinton (2009).

How to Draw M Sequences. An efficient way to draw the union of M i.i.d. noise sequences is to run the thinning algorithm once, with all intensities multiplied by M. In other words, the expected number of noise events on any interval is multiplied by M. This scheme does not tell us which specific noise sequence m generated a particular noise event, but the NCE objective equation (6.6) does not need to know that. The scheme works only because every noise sequence m has the same intensities $\lambda_k^{\mathrm{q}}(t \mid x_{[0,t)}^0)$ (not $\lambda_k^{\mathrm{q}}(t \mid x_{[0,t)}^m)$) at time t: there is no dependence on the previous events from that sequence. Amusingly, NCE can now run even with non-integer M.

Fractional Objective. One view of the thinning algorithm is that it accepts the proposed time t_i with probability $\mu = \lambda(t_i)/\overline{\lambda}$, and in that case, labels it as k with probability $\lambda_k(t_i)/\lambda(t_i)$. To get a greater diversity of noise samples, we can accept the time with probability 1, if we then scale its term in the objective equation (6.6) by μ. This does not change the expectation equation (6.6) but may reduce the sampling variance in estimating it. Note that increasing the upper bound $\overline{\lambda}$ now has an effect similar to increasing M: more noise samples.[6]

6.3.4 Computational Cost Analysis

Our models use neural networks whose state summarizes the history and is updated after each event. So to train on a single event sequence x with $I \geq 0$ events, both MLE and NCE must perform I updates to the neural state. Both MLE and NCE then

[6]This trick does carry computational cost: we need to train (via backpropagation) on proposals that might not have been accepted otherwise. This cost is perhaps not worth it when $\mu(t)$ is too low: it might be better spent on increasing M or running more training epochs for a fixed M. As a compromise, if μ is small (≤ 0.05 in our current experiments), we revert to the original approach of accepting the time with probability μ and not scaling it.

evaluate the intensities $\lambda_k(t \mid x_{[0,t)})$ of these I events, and also the intensities of a number of events that did *not* occur, which almost surely fall at other times.[7]

Consider the *number of intensities evaluated*. For MLE, assume the Monte Carlo integration technique mentioned in section 6.1. MLE computes the intensity λ for I observed events and for all K possible events at each of J sampled times. We take $J = \rho I$ (with randomized rounding to an integer), where $\rho > 0$ is a hyperparameter. Hence, the expected total number of intensity evaluations is $I + \rho IK$.

For NCE with the coarse-to-fine strategy, let J be the total number of times *proposed* by the thinning algorithm. Observe that $\mathbb{E}[I] = \int_0^T \lambda^*(t \mid x_{[0,t)})dt$, and $\mathbb{E}[J] = M \cdot \int_0^T \overline{\lambda}(t \mid x_{[0,t)})dt$. Thus, $\mathbb{E}[J] \approx M \cdot \mathbb{E}[I]$ if (1) $\overline{\lambda}$ at any time is a tight upper bound on the noise event rate λ^{q} at that time and (2) the average noise event rate well-approximates the average observed event rate (which should become true very early in training). To label or reject each of the J proposals, NCE evaluates C noise intensities λ_c^{q}; if the proposal is accepted with label k (perhaps fractionally), it must also evaluate its model intensity λ_k. The noise and model intensities λ_c^{q} and λ_k must also be evaluated for the I observed events. Hence, the total number of intensity evaluations is at most $(C+1)J + 2I$, which $\approx (C+1)MI + 2I$ in expectation.

Dividing by I, we see that making $(M+1)(C+1) \leq \rho K$ suffices to make NCE's stochastic objective take less work per observed sequence than MLE's stochastic objective. $M = 1$ and $C = 1$ is a valid choice. But NCE's objective is less informed for smaller M, so its stochastic gradient carries less information about $\boldsymbol{\theta}^*$. In section 6.5, we empirically investigate the effect of M and C on NCE and compare to MLE with different ρ.

[7] In between the events, even if the neural state remains constant, the intensity functions need not be constant.

6.3.5 Correct Classification Guarantees High Likelihood

The following theorem implies that stochastic gradient ascent on NCE converges to a correct $\boldsymbol{\theta}$ (if one exists):

Theorem 1 (Optimality)**.** *Under assumptions 1 and 2,* $\boldsymbol{\theta} \in \arg\max_{\boldsymbol{\theta}} J_{\mathrm{NC}}(\boldsymbol{\theta})$ *if and only if* $p_{\boldsymbol{\theta}} = p^*$.

This theorem falls out naturally when we rearrange the NCE objective as

$$J_{\mathrm{NC}}(\boldsymbol{\theta}) = \int_{t=0}^{T} \sum_{x^0_{[0,t)}} p^*(x^0_{[0,t)}) \sum_{k=1}^{K} \underline{\lambda}^*_k(t \mid x^0_{[0,t)}) H_{\boldsymbol{\theta}}(k, t, x^0_{[0,t)}) \mathrm{d}t$$

where $H_{\boldsymbol{\theta}}(k, t, x^0_{[0,t)})$ is a negative cross entropy, spelled out as

$$H_{\boldsymbol{\theta}}(k, t, x^0_{[0,t)}) = \frac{\lambda^*_k(t \mid x^0_{[0,t)})}{\underline{\lambda}^*_k(t \mid x^0_{[0,t)})} \log \frac{\lambda_k(t \mid x^0_{[0,t)})}{\underline{\lambda}_k(t \mid x^0_{[0,t)})} + M \frac{\lambda^q_k(t \mid x^0_{[0,t)})}{\underline{\lambda}^*_k(t \mid x^0_{[0,t)})} \log \frac{\lambda^q_k(t \mid x^0_{[0,t)})}{\underline{\lambda}_k(t \mid x^0_{[0,t)})}$$

where λ^*_k is the intensity under p^* and $\underline{\lambda}^*_k$ is defined analogously to $\underline{\lambda}_k$: see full derivation in section 6.B.1. Obviously, $p_{\boldsymbol{\theta}} = p^*$ is *sufficient* to maximize the negative cross-entropy for any k given any history and thus maximize $J_{\mathrm{NC}}(\boldsymbol{\theta})$. It turns out to be also *necessary* because any $\boldsymbol{\theta}$ for which $p_{\boldsymbol{\theta}} \neq p^*$ would, given assumption 1, end up decreasing the negative cross-entropy for some k over some interval (t, t') given a set of histories with non-zero measure. A full proof can be found in section 6.B.2: as we'll see there, although it resembles Theorem 3.2 of Ma and Collins (2018), the proof of our Theorem 1 requires new analysis to handle continuous time, since Ma and Collins (2018) only worked on discrete-time sequential data.

Moreover, our NCE method is strongly consistent for any $M \geq 1$ and approaches *Fisher efficiency* when M is large. These properties are the same as in Ma and Collins (2018) and the proofs are also similar. Therefore, we leave the related theorems

together with their assumptions and proofs to sections 6.B.3 and 6.B.4.

6.4 Related Work

The original "binary classification" NCE principle was proposed by Gutmann and Hyvärinen (2010) to estimate parameters for joint models of the form $p_{\boldsymbol{\theta}}(x) \propto \exp(\text{score}(x, \boldsymbol{\theta}))$. Gutmann and Hyvärinen (2012) applied it to natural image statistics. It was then widely applied to natural language processing problems such as language modeling (Mnih and Teh, 2012), learning word representations (Mikolov et al., 2013) and machine translation (Vaswani et al., 2013). The "ranking-based" variant (Jozefowicz et al., 2016)[8] is better suited for conditional distributions (Ma and Collins, 2018), including those used in autoregressive models, and has shown strong performance in large-scale language modeling with recurrent neural networks.

Guo, Li, and Liu (2018) tried NCE on (univariate) point processes but used the binary classification version. They used discrimination problems of the form: "Is event k at time t' the true next event following history $x_{[0,t]}$, or was it generated from a noise distribution?" Their classification-based NCE variant is *not* well-suited to conditional distributions (Ma and Collins, 2018): this complicates their method since they needed to build a parametric model of the local normalizing constant, giving them weaker theoretical guarantees and worse performance (see section 6.5). In contrast, we choose the ranking-based variant: our key idea of how to apply this to continuous time is new (see section 6.3) and requires new analysis (see sections 6.A and 6.B).

[8] Jozefowicz et al. (2016) considered it a competitor to NCE; Ma and Collins (2018) argued for regarding it as a variant.

6.5 Experiments

We evaluate our NCE method on several synthetic and real-world datasets, with comparison to MLE, Guo, Li, and Liu (2018) (denoted as b-NCE), and least-squares estimation (LSE) (Eichler, Dahlhaus, and Dueck, 2017). b-NCE has the same hyper-parameter M as our NCE, namely the number of noise events. LSE's objective involves an integral over times $[0, T)$, so it has the same hyper-parameter ρ as MLE.

On each of the datasets, we will show the estimated log-likelihood on the held-out data achieved by the models trained on the NCE, b-NCE, MLE and LSE objectives, as training consumes increasing amounts of computation—measured by the number of intensity evaluations and the elapsed wall-clock time (in seconds).[9] We always set the minibatch size B to exhaust the GPU capacity, so smaller ρ or M allows larger B. Larger B in turn increases the number of epochs per unit time (but decreases the possibly beneficial variance in the stochastic gradient updates).

6.5.1 Synthetic Datasets

In this section, we work on two synthetic datasets with $K = 10000$ event types. We choose the neural Hawkes process (NHP) (Chapter 3) to be our model $p_{\boldsymbol{\theta}}$.[10] For the noise distribution q, we choose $C = 1$ and also parametrize its intensity function as a neural Hawkes process.

The first dataset has sequences drawn from the randomly initialized q such that we can check how well our NCE method could perform with the "ground-truth" noise distribution $q = p^*$; the sequences of the second dataset were drawn from a randomly

[9] Our code is written in PyTorch (Paszke et al., 2017) and can be found at https://github.com/HMEIatJHU/nce-mpp. Our experiments were run on NVIDIA Tesla K80.

[10] Our method can also be used for other models with parametric intensity functions.

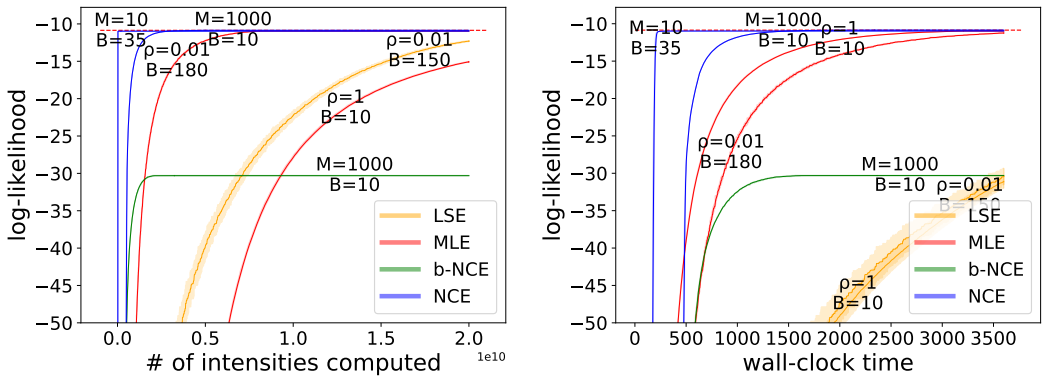
initialized neural Hawkes process to evaluate both methods in the case that the model family $p_{\boldsymbol{\theta}}$ is well-specified. We show (the zoomed-in views of the interesting parts of) multiple learning curves on each dataset in Figure 6.1: NCE is observed to consume substantially fewer intensity evaluations and less wall-clock time than MLE to achieve competitive log-likelihood, while b-NCE and LSE are slower and only converge to lower log-likelihood. Note that the wall-clock time may not be proportional to the number of intensities because computing intensities is not all of the work (e.g., there are LSTM states of both $p_{\boldsymbol{\theta}}$ and q to compute and store on GPU).

We also observed that models that achieved comparable log-likelihood—no matter how they were trained—achieved comparable prediction accuracies (measured by root-mean-square-error for time and error rate for type). Therefore, our NCE still beats other methods at converging quickly to the highest prediction accuracy.

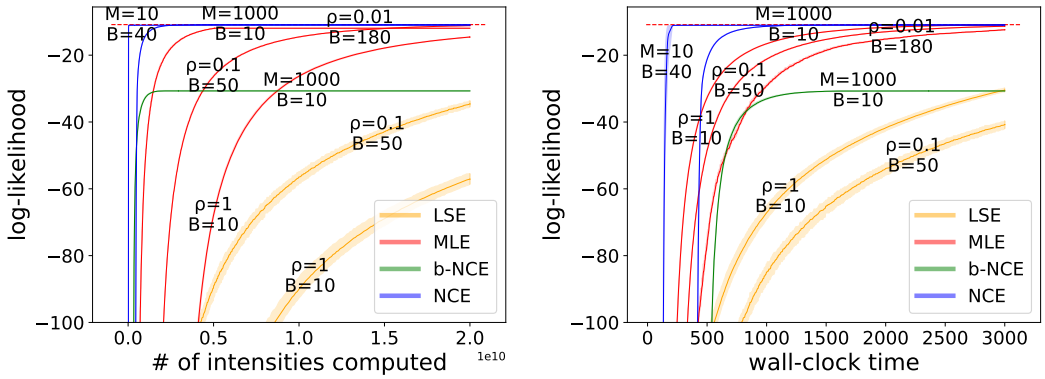
Ablation Study I: Always or Never Redraw Noise Samples. During training, for each observed data, we can choose to either redraw a new set of noise samples every time we train on it or keep reusing the old samples: we did the latter for Figure 6.1. In experiments doing the former, we observed better generation for tiny M (e.g., $M = 1$) but substantial slow-down (because of sampling) with no improved generalization for large M (e.g, 1000). Such results suggest that we always reuse old samples as long as M is reasonably large: it is then what we do for all other experiments throughout the chapter. See section 6.D.4 for more details of this ablation study, including learning curves of the "always redraw" strategy in Figure 6.6.

6.5.2 Real-World Social Interaction Datasets with Large K

We also evaluate the methods on several real-world social interaction datasets that have many event types: see section 6.D.1 for details (e.g, data statistics, pre-processing, data



(a) Synthetic-1: $p^* = q$.



(b) Synthetic-2: p^* and $p_{\boldsymbol{\theta}}$ are of the same family.

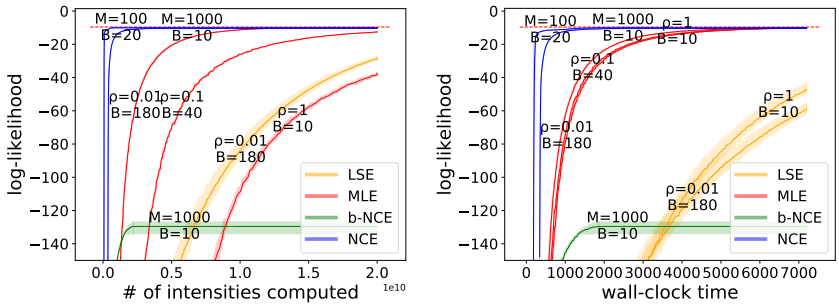
Figure 6.1: Learning curves of MLE and NCE on synthetic datasets. The displayed ρ and M values are among the better ones that we found during hyperparameter search. The horizontal red line marks the highest held-out log-likelihood achieved by MLE. The shaded area of each curve shows the range of log-likelihood of three independent runs; most of them are too narrow to be easily noticed.

splits, etc). In this section, we show the learning curves on two particularly interesting datasets (explained below) in Figure 6.2 and leave those on the other datasets (which look similar) to section 6.D.3.

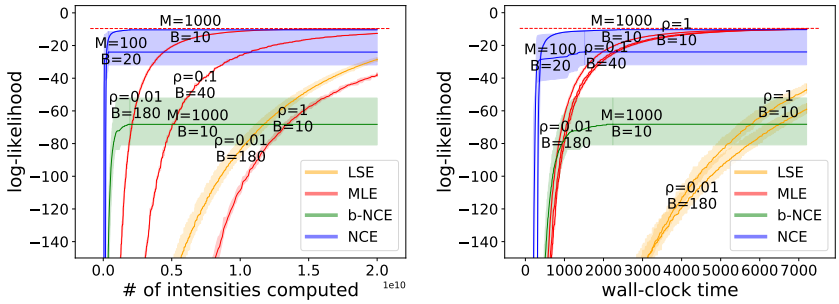
EuroEmail (Paranjape, Benson, and Leskovec, 2017)**.** This dataset contains time-stamped emails between anonymized members of a European research institute. We work on a subset of 100 most active members and then end up with $K = 10000$ possible event types and 50000 training event tokens.

BitcoinOTC (Kumar et al., 2016b)**.** This dataset contains time-stamped rating (positive/negative) records between anonymized users on the BitcoinOTC trading platform. We work on a subset of 100 most active users and then end up with $K = 19800$ (self-rating not allowed) possible event types but only 1000 training event tokens: this is an extremely data-sparse setting.

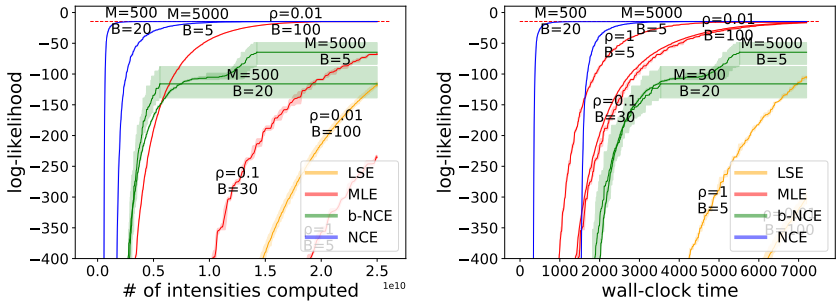
On these datasets, our model $p_{\boldsymbol{\theta}}$ is still a neural Hawkes process. For the noise distribution q, we experiment with not only the coarse-to-fine neural process with $C = 1$ but also a homogeneous Poisson process. As shown in Figure 6.2, our NCE tends to perform better with the neural q: this is because a neural model can better fit the data and thus provide better training signals, analogous to how a good generator can benefit the discriminator in the generative adversarial framework (Goodfellow et al., 2014). NCE with Poisson q also shows benefits through the early and middle training stages, but it might suffer larger variance (e.g., Figure 6.2a2) and end up with slightly worse generalization (e.g., Figure 6.2b2). MLE with different ρ values all eventually achieve the highest log-likelihood (≈ -10 on EuroEmail and ≈ -15 on BitcoinOTC), but most of these runs are so slow that their peaks are out of the current views. The b-NCE runs with different M values are slower, achieve worse



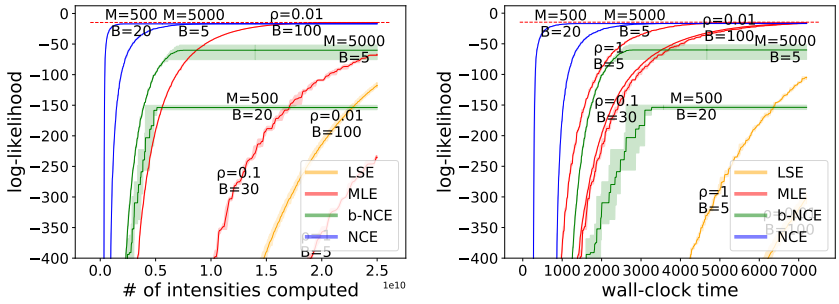
(a1) EuroEmail: neural q



(a2) EuroEmail: Poisson q



(b1) BitcoinOTC: neural q.



(b2) BitcoinOTC: Poisson q.

Figure 6.2: Learning curves of MLE and NCE on the real-world social interaction datasets.

generalization and suffer larger variance than our NCE; interestingly, b-NCE prefers Poisson q to neural q (better generalization on EuroEmail and smaller variance on BitcoinOTC). In general, LSE is the slowest, and the highest log-likelihood it can achieve (≈ -30 on EuroEmail and ≈ -25 on BitcoinOTC) is lower than that of MLE and our NCE.

Ablation Study II: Trained vs. Untrained q. The noise distributions (except the ground-truth q for Synthetic-1) that we have used so far were all pretrained on the same data as we train $p_{\boldsymbol{\theta}}$. The training cost is cheap: e.g., on the datasets in this section, the actual wall-clock training time for the neural q is less than 2% of what is needed to train $p_{\boldsymbol{\theta}}$, and training the Poisson q costs even less.[11][12] We also experimented with untrained noise distributions and they were observed to perform worse (e.g., worse generalization, slower convergence and larger variance). See section 6.D.5 for more details, including learning curves (Figure 6.7).

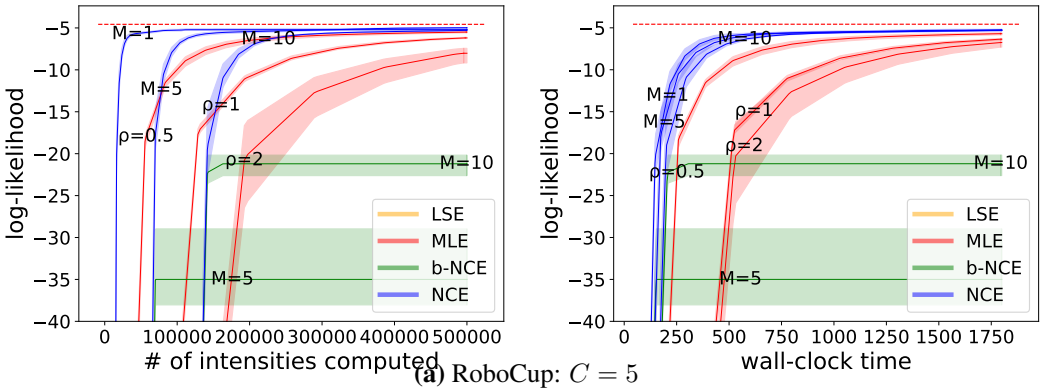
6.5.3 Real-World Dataset with Dynamic Facts

In this section, we let $p_{\boldsymbol{\theta}}$ be a neural Datalog through time (NDTT) model (Chapter 4). Such a model can be used in a domain in which new events dynamically update the set of event types and the structure of their intensity functions. We evaluate our method on training the domain-specific models presented in Chapter 4, on the same datasets they used:

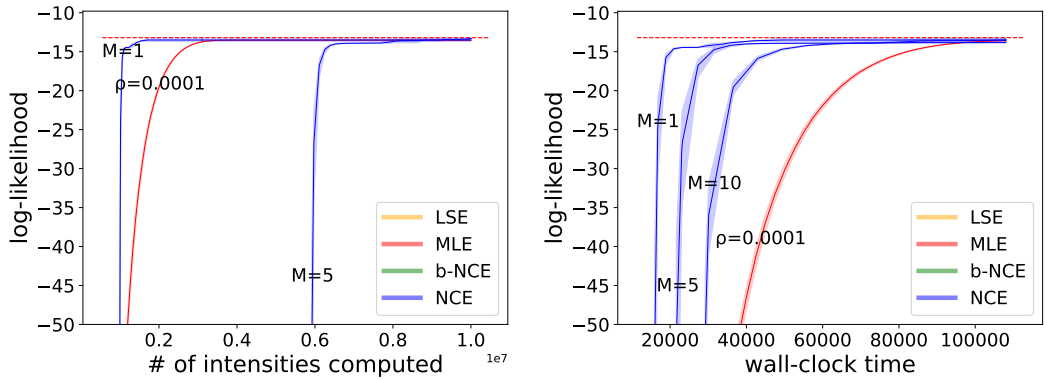
RoboCup (Chen and Mooney, 2008)**.** This dataset logs actions of robot players during RoboCup soccer games. The set of possible event types dynamically changes

[11] We train q by MLE: summing C intensities is not expensive when C is small. In section 6.C.2, we document an alternative strategy that uses q as the noise distribution to train itself by NCE.

[12] For the experiments in section 6.5.3, training the neural q takes only $< 1/100$ of what needed to train $p_{\boldsymbol{\theta}}$.



(a) RoboCup: $C = 5$



(b) IPTV: $C = 49$

Figure 6.3: Learning curves of MLE and NCE on RoboCup and IPTV datasets.

over time (e.g., only ball possessor can kick or pass) as the ball is frequently transferred between players (by passing or stealing). There are $K = 528$ event types over all time, but only about 20 of them are possible at any given time.

IPTV (Xu, Luo, and Carin, 2018)**.** This dataset contains time-stamped records of 1000 users watching 49 TV programs over 2012. The users are not able to watch a program until it is released, so the number of event types grows from $K = 0$ to $K = 49000$ as programs are released one after another.

The learning curves are displayed in Figure 6.3. On RoboCup, NCE only progresses faster than MLE at the early to middle training stages: $M = 5$ and $M = 10$ eventually achieved the highest log-likelihood at the same time as MLE and $M = 1$ ended up with worse generalization. On IPTV, NCE with $M = 1$ turned out to learn as well as and much faster than MLE. The dynamic architecture makes it hard to parallelize the intensity computation; MLE in particular performs poorly in wall-clock time, and we needed a remarkably small ρ to let MLE finish within the shown time range. On both datasets, b-NCE and LSE drastically underperform MLE and NCE: their learning curves increase so slowly and achieve such poor generalization that only b-NCE with $M = 5$ and $M = 10$ are visible on the graphs.

Ablation Study III: Effect of C. In the above figures, we used the coarse-to-fine neural model as q. On RoboCup, each action (kick, pass, etc.) has a coarse-grained intensity, so $C = 5$. On IPTV, we partition the event vocabulary by TV program, so $C = 49$. We also experimented with $C = 1$: this reduces the number of intensities computed during sampling on both datasets, but has (slightly) worse generalization on RoboCup (since q becomes less expressive). See section 6.D.6 for more details, including learning curves (Figure 6.8).

6.6 Comparison to A Better Version of MLE

So far, we have only compared our NCE method to the Monte-Carlo MLE method that we presented in Chapter 2. However, as we have stated in that chapter, it is possible to find a lower-variance estimate for the integral of the log-likelihood, yielding a better version of MLE. In this chapter, we will introduce such a better MLE variant.

6.6.1 A Better Estimate for the Log-Likelihood

Recall from Chapter 2 that we can regard the integral $\int_{t=0}^{T} \sum_{k=1}^{K} \lambda_k(t \mid x_{[0,t)}) \mathrm{d}t$ as the expected total number of events yielded by the point process where the intensities $\lambda_k(t \mid x_{[0,t)})$ depend on the true history $x_{[0,t)}$. That is, for any given observed sequence $x_{[0,T)}$, we can rewrite the integral to be

$$\int_{t=0}^{T} \sum_{k=1}^{K} \lambda_k(t \mid x_{[0,t)}) \mathrm{d}t = \int_{t=0}^{T} \sum_{k=1}^{K} \mathbb{E}_{x_t^1 \sim p}[\mathbb{1}(x_t^1 = k)] = \int_{t=0}^{T} \mathbb{E}_{x_t^1 \sim p}[\mathbb{1}(x_t^1 \neq \varnothing)] \tag{6.8}$$

where $\mathbb{1}(\cdot)$ is the indicator function. Note that the superscripted x_t^1 is not the actual event or non-event x_t; it is the event or non-event *proposed* by the model distribution p given the *actual* history $x_{[0,t)}$. The full sequence $x_{[0,T)}^1$ is a joint sample of all the x_t^1 values for $t \in [0, T)$; it can be obtained using the modified thinning algorithm as described in section 6.3.3.

Given any proposal $x_{[0,T)}^1$, we can estimate this rewritten integral as $\sum_{t:x_t^1 \neq \varnothing} 1$; we can also average over multiple proposals $x_{[0,T)}^1, \ldots, x_{[0,T)}^M$ to reduce variance: $\sum_{m=1}^{M} \sum_{t:x_t^m \neq \varnothing} \frac{1}{M}$. Then the new MLE training objective is given as

$$\sum_{t:x_t \neq \varnothing} \log \lambda_{x_t}(t \mid x_{[0,t)}) - \sum_{m=1}^{M} \sum_{t:x_t^m \neq \varnothing} \frac{1}{M} \tag{6.9}$$

But how can we adjust the model parameters $\boldsymbol{\theta}$ to maximize this new objective as

the estimated integral seems to be independent of $\boldsymbol{\theta}$? The trick is to rewrite each 1 as $\lambda_{x_t^m}(t \mid x_{[0,t)})\mathrm{d}t / \lambda^{\text{cur}}_{x_t^m}(t \mid x_{[0,t)})\mathrm{d}t$; λ is a function of $\boldsymbol{\theta}$ while λ^{cur} denotes the intensity *values* computed using the current parameter *values*, which won't change with $\boldsymbol{\theta}$. The infinitesimal $\mathrm{d}t$ factors cancel out. Then the new MLE objective becomes:

$$\sum_{t:x_t \neq \varnothing} \log \lambda_{x_t}(t \mid x_{[0,t)}) - \sum_{m=1}^{M} \sum_{t:x_t^m \neq \varnothing} \frac{1}{M} \frac{\lambda_{x_t^m}(t \mid x_{[0,t)})}{\lambda^{\text{cur}}_{x_t^m}(t \mid x_{[0,t)})} \tag{6.10}$$

This is actually the *importance reweighting*: each 1 is sampled with probability $\lambda^{\text{cur}}_{x_t^m}(t \mid x_{[0,t)})\mathrm{d}t$ and then reweighted by considering its actual probability $\lambda_{x_t^m}(t \mid x_{[0,t)})\mathrm{d}t$—although their ratio is trivial (i.e., always 1), only the numerator changes with the model parameters $\boldsymbol{\theta}$, so the gradient is not 0. To increase equation (6.10), gradient ascent will have to adjust the parameters to decrease the intensities of the proposed events. The denominator means that the adjustment will be stronger for the *proposed* events that currently have low-intensities and thus are sampled less often.

Since equation (6.10) can be regarded as importance sampling, we can in principle change the sampling distribution—also called the proposal distribution—to anything else, as long as we also change the denominator accordingly. We would like the proposal distribution to be fairly close to the true distribution, so a good choice is the noise distribution q that we have trained on the same training data. Now, each 1 is sampled with probability $\lambda^q_{x_t^m}(t \mid x_{[0,t)})\mathrm{d}t$, and we correct that 1 by multiplying it with $\lambda_{x_t^m}(t \mid x_{[0,t)})\mathrm{d}t / \lambda^q_{x_t^m}(t \mid x_{[0,t)})\mathrm{d}t$ (where the dt factors cancel)—we still "count" the samples, but each proposed (t, k) contributes $\lambda_k(t \mid x_{[0,t)}) / \lambda^q_k(t \mid x_{[0,t)})$ (instead of 1) to the total. Then the new MLE objective becomes:

$$\sum_{t:x_t \neq \varnothing} \log \lambda_{x_t}(t \mid x_{[0,t)}) - \sum_{m=1}^{M} \sum_{t:x_t^m \neq \varnothing} \frac{1}{M} \frac{\lambda_{x_t^m}(t \mid x_{[0,t)})}{\lambda^q_{x_t^m}(t \mid x_{[0,t)})} \tag{6.11}$$

which is still an unbiased estimate of the true log-likelihood.

Optimizing equation (6.11) has a couple of advantages over equation (6.10). First, the noise distribution q is cheaper to sample from—thanks to its coarse-to-fine structure (section 6.3.3). Second, we can reuse proposals drawn from q, just like we reused noise samples in section 6.5. This suffices to work well because q is already close to the true distribution p^*. In contrast, equation (6.10) requires us to repeatedly call the modified thinning algorithm (section 6.3.3) as the model p improves, which would slow the training down just like we have discussed in section 6.5.1.

This new MLE method has a better sample efficiency than the original MLE that we used in section 6.5. Remember that when using unnormalized importance sampling to estimate the expectation of a non-negative random variable, the optimal proposer uses a distribution that is proportional to that variable. That is, in our case, we'd like to sample (t, k) more often when it has a higher intensity—that is what the thinning algorithm does. Additionally, the new MLE method has the same hyper-parameter M—namely, the number of noise sequences—and the same runtime complexity as our NCE method (section 6.3.4).

6.6.2 Experiments with MLE-IS

In this section, we compare our NCE method to the new MLE, which we call as MLE-IS where IS stands for importance sampling. In our experiments, to get a greater diversity of proposals, we use the *fractional* version of the sampling algorithm (section 6.3.3). This leads us to the *fractional* version of the MLE-IS training objective: each summand in the integral estimate is multiplied by the probability of that sample being accepted by the thinning algorithm.

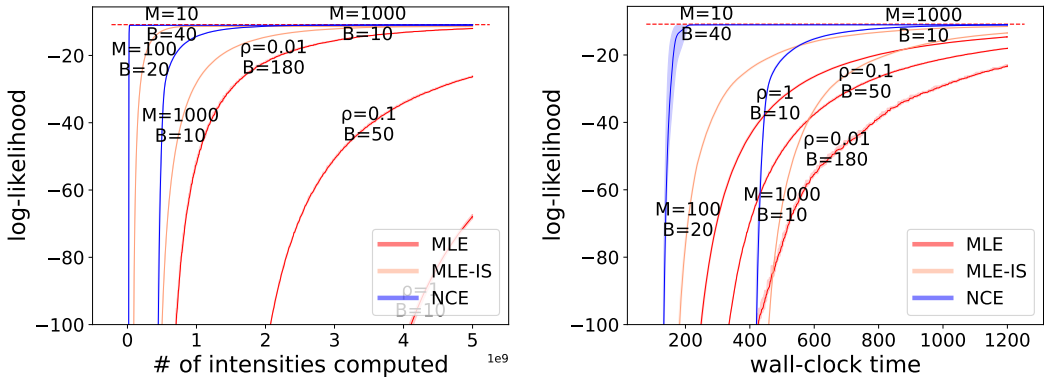


Figure 6.4: Learning curves of MLE, MLE-IS, and NCE on the Synthetic-2 dataset. These figures are Figure 6.1b zoomed-in, with MLE-IS added.

MLE-IS was not included in Mei, Wan, and Eisner (2020). However, using MLE-IS allows us to more fairly compare the MLE and NCE objectives, since MLE-IS and NCE estimate these respective objectives in the same way. Namely, MLE-IS and NCE both sample events (t, k) from the same coarse model q of the data (i.e., the "proposal" or "noise" distribution), using the same fractional thinning algorithm. Both also contrast these sampled events with the actual events; however, they do so using slightly different objectives.

Our original MLE method (which uses Algorithm 2.1 to estimate the integral) can be regarded as a higher-variance version of MLE-IS in which the proposal distribution is effectively uniform over $\{1, \ldots, K\} \times [0, T)$.

In Figure 6.4, we replot the MLE and NCE learning curves from Figure 6.1b, but we now add the learning curves for MLE-IS. As expected, MLE-IS outperforms the original MLE, but NCE is still the best at converging quickly, at least on this one dataset (Synthetic-2). We did not experiment on other datasets.

6.7 Conclusion

In this chapter, we introduced a novel instantiation of the general NCE principle for training a multivariate point process model. Our objective has the same optimal parameters as the log-likelihood objective (if the model is well-specified), but needs fewer expensive function evaluations and much less wall-clock time in practice. This benefit is demonstrated on several synthetic and real-world datasets. Moreover, our method is provably consistent and efficient under mild assumptions.

Appendices

6.A Proof Details for MLE

In this section, we prove the claim in section 6.1 that $\arg\max_{\boldsymbol{\theta}} J_{\mathrm{LL}}(\boldsymbol{\theta}) = \Theta^* \stackrel{\text{def}}{=} \{\boldsymbol{\theta}^* : p_{\boldsymbol{\theta}^*} = p^*\}$. For this purpose, we first rearrange $J_{\mathrm{LL}}(\boldsymbol{\theta}) = \mathbb{E}_{p^*(x_{[0,T)})}[\log p_{\boldsymbol{\theta}}(x_{[0,T)})]$ as below:

$$\sum_{x_{[0,T)}} p^*(x_{[0,T)}) \log p_{\boldsymbol{\theta}}(x_{[0,T)}) \tag{6.12a}$$

$$= \int_{t=0}^{T} \sum_{x_{[0,t)}} p^*(x_{[0,t)}) \underbrace{\sum_{x_{[t,t+\mathrm{d}t)}} p^*(x_{[t,t+\mathrm{d}t)} \mid x_{[0,t)}) \log p_{\boldsymbol{\theta}}(x_{[t,t+\mathrm{d}t)} \mid x_{[0,t)})}_{\text{call it } H_{\boldsymbol{\theta}}(t, x_{[0,t)})} \tag{6.12b}$$

The intuition for equation (6.12b) is that due to the form of the autoregressive model, $\log p_{\boldsymbol{\theta}}(x_{[0,T)})$ in equation (6.12a) can be broken up into a sum of log (infinitesimal) probabilities of $x_{[t,t+\mathrm{d}t)}$ on the infinitesimal intervals $[t, t+dt)$, each probability being conditioned on the past history $x_{[0,t)}$. When we take the expectation under p^*, each summand gets weighted by the probability that $x_{[0,t)}$ and $x_{[t,t+\mathrm{d}t)}$ would take on the values in that summand. This gives a form equation (6.12b) that aggregates the infinitesimal quantities $H_{\boldsymbol{\theta}}(t, x_{[0,t)})$ over possible times $t \in [0, T)$ and possible histories $x_{[0,t)}$.

Proof. We first observe that $H_{\boldsymbol{\theta}}(t, x_{[0,t)})$ is the negative cross-entropy between the

conditional distributions of p^* and $p_{\boldsymbol{\theta}}$ at time t (both conditioned on history $x_{[0,t)}$). Technically, $x_{[t,t+\mathrm{d}t)}$ will have an event of type k with probability $\lambda_k^*(t)\mathrm{d}t$ under p^* ($\lambda_k(t)\mathrm{d}t$ under $p_{\boldsymbol{\theta}}$) or has no event at all with probability $1 - \sum_{k=1}^K \lambda_k^*(t)\mathrm{d}t$ under p^* ($1 - \sum_{k=1}^K \lambda_k(t)\mathrm{d}t$ under $p_{\boldsymbol{\theta}}$). So the term $H_{\boldsymbol{\theta}}(t, x_{[0,t)})$ is actually the negative cross entropy between the following two discrete distributions over $\{\varnothing, 1, \ldots, K\}$:

$$\left[\left(1 - \sum_{k=1}^K \lambda_k^*(t \mid x_{[0,t)})\mathrm{d}t\right),\ \lambda_1^*(t \mid x_{[0,t)})\mathrm{d}t,\ \ldots\ ,\ \lambda_K^*(t \mid x_{[0,t)})\mathrm{d}t\right] \tag{6.13a}$$

$$\left[\left(1 - \sum_{k=1}^K \lambda_k(t \mid x_{[0,t)})\mathrm{d}t\right),\ \lambda_1(t \mid x_{[0,t)})\mathrm{d}t,\ \ldots\ ,\ \lambda_K(t \mid x_{[0,t)})\mathrm{d}t\right] \tag{6.13b}$$

The (infinitesimal) negative cross-entropy between them is always smaller than or equal to the negative entropy of the distribution in equation (6.13a): it will be strictly smaller if these two distributions are distinct, and equal when they are identical.

It is then obvious that any $\boldsymbol{\theta}^* \in \Theta^*$ maximizes $J_{\mathrm{LL}}(\boldsymbol{\theta})$ because it maximizes the negative cross-entropy for any history $x_{[0,t)}$ at any time t.

To check if any other $\bar{\boldsymbol{\theta}} \notin \Theta^*$ maximizes $J_{\mathrm{LL}}(\boldsymbol{\theta})$ as well, we analyze

$$J_{\mathrm{LL}}(\bar{\boldsymbol{\theta}}) - J_{\mathrm{LL}}(\boldsymbol{\theta}^*) = \int_{t=0}^T \sum_{x_{[0,t)}} p^*(x_{[0,t)}) \underbrace{\left(H_{\bar{\boldsymbol{\theta}}}(t, x_{[0,t)}) - H_{\boldsymbol{\theta}^*}(t, x_{[0,t)})\right)}_{\text{denote it as } G_{\bar{\boldsymbol{\theta}}}(t, x_{[0,t)})\mathrm{d}t} \tag{6.14}$$

where $\boldsymbol{\theta}^*$ can be any member in Θ^*. Note that we can denote $H_{\bar{\boldsymbol{\theta}}} - H_{\boldsymbol{\theta}^*}$ as $G_{\bar{\boldsymbol{\theta}}}\mathrm{d}t$ because the probabilities in H and thus the entropy changes (if any) are all infinitesimal.

According to the definition of $\bar{\boldsymbol{\theta}}$ and $\boldsymbol{\theta}^*$, there must exist a sequence $\bar{x}_{[0,T)}$, a time $\bar{t} \in (0, T)$ and a type $\bar{k} \in \{1, \ldots, K\}$ such that $\lambda_{\bar{k}}(\bar{t} \mid \bar{x}_{[0,\bar{t})}) \neq \lambda_{\bar{k}}^*(\bar{t} \mid \bar{x}_{[0,\bar{t})})$. Therefore, we have $G_{\bar{\boldsymbol{\theta}}}(\bar{t}, \bar{x}_{[0,\bar{t})}) < 0$ since the distributions in equation (6.13) are distinct for the given history $\bar{x}_{[0,\bar{t})}$. Does this difference lead to any overall change of the entire objective?

Actually, according to Lemma 1 (that we will prove shortly), the existence of such $\bar{x}_{[0,T)}$, $\bar{t}$ and $\bar{k}$ implies that there exists an interval $(t', t'') \subset [0, T)$ such that, for any $t \in (t', t'')$, there exists a set $\mathcal{X}(t)$ of histories with non-zero measure such that any $x_{[0,t)} \in \mathcal{X}(t)$ satisfies $\lambda_{\bar{k}}(t \mid x_{[0,t)}) \neq \lambda^*_{\bar{k}}(t \mid x_{[0,t)})$. That is to say, the fraction of the integral over (t', t'') is a non-infinitesimal negative number:

$$\int_{t=t'}^{t''} \sum_{x_{[0,t)}} p^*(x_{[0,t)}) G_{\bar{\boldsymbol{\theta}}}(t, x_{[0,t)}) \mathrm{d}t \tag{6.15a}$$

$$= \underbrace{\int_{t=t'}^{t''} \sum_{x_{[0,t)} \in \mathcal{X}(t)} p^*(x_{[0,t)}) G_{\bar{\boldsymbol{\theta}}}(t, x_{[0,t)}) \mathrm{d}t}_{<0} + \underbrace{\int_{t=t'}^{t''} \sum_{x_{[0,t)} \notin \mathcal{X}(t)} p^*(x_{[0,t)}) G_{\bar{\boldsymbol{\theta}}}(t, x_{[0,t)}) \mathrm{d}t}_{\leq 0} \tag{6.15b}$$

where the second integral ≤ 0 because $G_{\boldsymbol{\theta}}$ always ≤ 0. For the same reason, we also have $\int_{t=0}^{t'} \sum_{x_{[0,t)}} p^*(x_{[0,t)}) G_{\bar{\boldsymbol{\theta}}}(t, x_{[0,t)}) \mathrm{d}t \leq 0$ and $\int_{t=t''}^{T} \sum_{x_{[0,t)}} p^*(x_{[0,t)}) G_{\bar{\boldsymbol{\theta}}}(t, x_{[0,t)}) \mathrm{d}t \leq 0$. Then the overall difference must be strictly negative, i.e.,

$$J_{\mathrm{LL}}(\bar{\boldsymbol{\theta}}) - J_{\mathrm{LL}}(\boldsymbol{\theta}^*) < 0 \tag{6.16}$$

Note that this inequality holds for any $\bar{\boldsymbol{\theta}} \notin \Theta^*$ and any $\boldsymbol{\theta}^* \in \Theta^*$, meaning that $\boldsymbol{\theta}^* \in \Theta^*$ is necessary to maximize the objective.

Now the proof of $\arg\max_{\boldsymbol{\theta}} J_{\mathrm{LL}}(\boldsymbol{\theta}) = \Theta^*$ is complete.

□

Lemma 1. *Suppose that we have two intensity functions that meet assumption 1: they have different parameters $\boldsymbol{\theta}$ and $\boldsymbol{\theta}^*$ and are denoted as $\lambda_k(t \mid x_{[0,t)})$ and $\lambda^*_k(t \mid x_{[0,t)})$ respectively. If there exists a sequence $\bar{x}_{[0,T)}$, a time $\bar{t} \in (0, T)$ and a type $\bar{k} \in \{1, \ldots, K\}$ such that $\lambda_{\bar{k}}(\bar{t} \mid \bar{x}_{[0,\bar{t})}) \neq \lambda^*_{\bar{k}}(\bar{t} \mid \bar{x}_{[0,\bar{t})})$, then there exists an open*

interval $(t', t'') \subset [0, T)$ *such that, for any* $t \in (t', t'')$*, there exists a set* $\mathcal{X}$ *of histories with non-zero measure such that any* $x_{[0,t)} \in \mathcal{X}$ *satisfies* $\lambda_{\bar{k}}(t \mid x_{[0,t)}) \neq \lambda^*_{\bar{k}}(t \mid x_{[0,t)})$.

This lemma says: if $\boldsymbol{\theta}$ and $\boldsymbol{\theta}^*$ are meaningfully different in that they predict different intensities at time t for *some history*, then they actually do so for a *set of histories of non-zero measure*, making this difference visible in the objective functions like $J_{\mathrm{LL}}(\boldsymbol{\theta})$ (see above) and $J_{\mathrm{NC}}(\boldsymbol{\theta})$ (see section 6.B). Note that previous work did not encounter this since they only worked on either non-sequential data (e.g., Gutmann and Hyvärinen (2010) and Gutmann and Hyvärinen (2012)) or discrete-time sequential data (e.g., Ma and Collins (2018)).

Proof. We first prove the existence of an interval (t', t'') such that $\lambda_{\bar{k}}(t \mid \bar{x}_{[0,t)}) \neq \lambda^*_{\bar{k}}(t \mid \bar{x}_{[0,t)})$ for the given sequence $\bar{x}_{[0,T)}$ and any time $t \in (t', t'')$. It turns out to be straightforward under assumption 1: since the intensity functions are continuous between events, we can construct this interval by expanding from the given time $\bar{t}$ until $\lambda_{\bar{k}}(t \mid \bar{x}_{[0,t)}) = \lambda^*_{\bar{k}}(t \mid \bar{x}_{[0,t)})$.

We use d to denote the maximal difference between the intensities over (t', t''), i.e., $d \stackrel{\text{def}}{=} \max_{t \in (t', t'')} |\lambda_{\bar{k}}(t \mid \bar{x}_{[0,t)}) - \lambda^*_{\bar{k}}(t \mid \bar{x}_{[0,t)})|$. Then, to facilitate the rest of the proof, we shrink the interval (t', t'') such that $|\lambda_{\bar{k}}(t \mid \bar{x}_{[0,t)}) - \lambda^*_{\bar{k}}(t \mid \bar{x}_{[0,t)})| > d/2$ for any time $t \in (t', t'')$.

Now, for any time $t \in (t', t'')$, we prove the existence of the set described in Lemma 1 by constructing it.

We initialize this set as $\{\bar{x}_{[0,t)}\}$. If $\bar{x}_{[0,t)}$ doesn't have any event, then its probability $p(\bar{x}_{[0,t)}) = \exp(-\int_{s=0}^{t} \sum_{k=1}^{K} \lambda_k(s \mid \bar{x}_{[0,s)}) \mathrm{d}s)$ is not infinitesimal and this set already has non-zero measure.

What if $\bar{x}_{[0,t)}$ has $I \geq 1$ events at times $0 < t_1 < \ldots < t_I < t$? Intuitively, we can construct many other histories satisfying the intensity inequality by slightly shifting the time of each event: as long as they aren't shifted by too far, the $d/2$ difference between intensities won't vanish (even if it decreases). See the formal proof as below.

In the case of $I \geq 1$, the probability $p(\bar{x}_{[0,t)})$ is infinitesimal in the order of $(\mathrm{d}t)^I$: $p(\bar{x}_{[0,t)}) = \prod_{i=1}^{I}(\lambda_{\bar{x}_{t_i}}(t_i \mid \bar{x}_{[0,t_i)})\mathrm{d}t) \exp(-\int_{s=0}^{t}\sum_{k=1}^{K}\lambda_k(s \mid \bar{x}_{[0,s)})\mathrm{d}s)$. Therefore, to construct a set with non-zero measure, the number of histories satisfying the inequality has to be in the order of $(\frac{1}{\mathrm{d}t})^I$.

We define an open interval (t_1', t_1'') that covers t_1 but not any other event time. Now we can construct uncountably many—in the order of $\frac{1}{\mathrm{d}t}$—histories $x_{[0,t)}$ by freely shifting the event time t_1 inside (t_1', t_1''). Suppose that t_1 has been shifted by $\Delta \in \mathbb{R}$. Under assumption 1, there is a continuous function $c(\Delta)$ such that $c(0) = 0$ and

$$\lambda_{\bar{k}}(t \mid x_{[0,t)}) - \lambda_{\bar{k}}^*(t \mid x_{[0,t)}) = \lambda_{\bar{k}}(t \mid \bar{x}_{[0,t)}) - \lambda_{\bar{k}}^*(t \mid \bar{x}_{[0,t)}) + c(\Delta) \tag{6.17}$$

meaning that the intensity difference will change by $c(\Delta)$. By triangle inequality, we have

$$\left|\lambda_{\bar{k}}(t \mid x_{[0,t)}) - \lambda_{\bar{k}}^*(t \mid x_{[0,t)})\right| \geq \left|\left|\lambda_{\bar{k}}(t \mid \bar{x}_{[0,t)}) - \lambda_{\bar{k}}^*(t \mid \bar{x}_{[0,t)})\right| - |c(\Delta)|\right| \tag{6.18}$$

Since $c(\Delta)$ is continuous, as long as we make $|\Delta|$ small enough, we'll have $|c(\Delta)| \leq d/2$ and then the following inequality holds:

$$\left|\lambda_{\bar{k}}(t \mid x_{[0,t)}) - \lambda_{\bar{k}}^*(t \mid x_{[0,t)})\right| \geq \left|\lambda_{\bar{k}}(t \mid \bar{x}_{[0,t)}) - \lambda_{\bar{k}}^*(t \mid \bar{x}_{[0,t)})\right| - |c(\Delta)| \tag{6.19a}$$

$$> d/2 - d/2 = 0 \tag{6.19b}$$

meaning that the intensities given the new history are still different. Therefore, as long

as we keep the interval (t_1', t_1'') small enough, we'll have order-$\frac{1}{\mathrm{d}t}$ many histories and the inequality in equation (6.19) holds given any of them.

Recall that we need order-$(\frac{1}{\mathrm{d}t})^I$ many such histories. We can obtain them by simply defining I *disjoint* open intervals $(t_1', t_1''), \ldots, (t_I', t_I'')$ such that $t_i \in (t_i', t_i'')$ and freely shifting each event time t_i inside (t_i', t_i''). Suppose that t_i has been shifted by $\Delta_i \in \mathbb{R}$, Under assumption 1, there is a continuous function $c(\Delta_1, \ldots, \Delta_I)$ such that $c(0, \ldots, 0) = 0$ and

$$\lambda_{\bar{k}}(t \mid x_{[0,t)}) - \lambda_{\bar{k}}^*(t \mid x_{[0,t)}) = \lambda_{\bar{k}}(t \mid \bar{x}_{[0,t)}) - \lambda_{\bar{k}}^*(t \mid \bar{x}_{[0,t)}) + c(\Delta_1, \ldots, \Delta_I) \quad (6.20)$$

Since c is a continuous function, there exist I positive real numbers $\bar{\Delta}_1, \ldots, \bar{\Delta}_I$ such that $|c(\Delta_1, \ldots, \Delta_I)| \leq d/2$ as long as $|\Delta_i| \leq \bar{\Delta}_i$ holds for all $i = 1, \ldots, I$. In this case, by triangle inequality, we still have

$$\left|\lambda_{\bar{k}}(t \mid x_{[0,t)}) - \lambda_{\bar{k}}^*(t \mid x_{[0,t)})\right| \geq \left|\lambda_{\bar{k}}(t \mid \bar{x}_{[0,t)}) - \lambda_{\bar{k}}^*(t \mid \bar{x}_{[0,t)})\right| - |\Delta_i| > 0 \quad (6.21)$$

Now we have order-$(\frac{1}{\mathrm{d}t})^I$ many histories: each of them has order-$(\mathrm{d}t)^I$ probability and the inequality in equation (6.21) holds given any of them. That is to say, the set of these histories has non-zero measure and we have $\lambda_{\bar{k}}(t \mid x_{[0,t)}) \neq \lambda_{\bar{k}}^*(t \mid x_{[0,t)})$ given any $x_{[0,t)}$ in this set.

This completes the proof.

□

6.B NCE Details

In this section, we will discuss the theoretical guarantees of our NCE method in detail.

6.B.1 Derivation Details

In this section, we show how to get the rearranged NCE objective in section 6.3.5 from equation (6.6). First of all, we observe that the NCE objective in equation (6.6), which is restated here for convenient reference

$$\mathbb{E}_{x^0_{[0,T)}\sim p^*, x^{1:M}_{[0,T)}\sim q}\Big[\sum_{t:x^0_t\neq\varnothing}\log\frac{\lambda_{x^0_t}(t\mid x^0_{[0,t)})}{\underline{\lambda}_{x^0_t}(t\mid x^0_{[0,t)})}+\sum_{m=1}^{M}\sum_{t:x^m_t\neq\varnothing}\log\frac{\lambda^{\mathrm{q}}_{x^m_t}(t\mid x^0_{[0,t)})}{\underline{\lambda}_{x^m_t}(t\mid x^0_{[0,t)})}\Big] \tag{6.22}$$

can be rearranged to be

$$\int_{t=0}^{T}\mathbb{E}_{x^0_{[0,t)}\sim p^*}[H_{\boldsymbol{\theta}}(t,x^0_{[0,t)})] \tag{6.23}$$

where $H_{\boldsymbol{\theta}}(t,x^0_{[0,t)})$ is shorthand for

$$\sum_{k=1}^{K}\lambda^*_k(t\mid x^0_{[0,t)})\mathrm{d}t\log\frac{\lambda_{x^0_t}(t\mid x^0_{[0,t)})}{\underline{\lambda}_{x^0_t}(t\mid x^0_{[0,t)})}+\sum_{m=1}^{M}\sum_{k=1}^{K}\lambda^{\mathrm{q}}_k(t\mid x^0_{[0,t)})\mathrm{d}t\log\frac{\lambda^{\mathrm{q}}_{x^m_t}(t\mid x^0_{[0,t)})}{\underline{\lambda}_{x^m_t}(t\mid x^0_{[0,t)})} \tag{6.24}$$

This rearrangement is similar to that of equations (6.12a)–(6.12b). The intuition of equation (6.6) is that we sample M i.i.d. noise sequences $x^1_{[0,T)},\ldots,x^M_{[0,T)}$ for each possible real data $x^0_{[0,T)}$, sum up the log-ratio whenever $x^{0:M}_t$ has an event, and then take the expectation over all the possible real data $x^0_{[0,T)}$. The intuition of equations (6.23) and (6.24) is that we draw noise samples $x^1_t,\ldots,x^M_t$ for each real history $x^0_{[0,t)}$ at each time t, compute the log-ratio if $x^{0:M}_t$ has an event, take the expectation of the log-ratio over all the possible real histories and then sum over all the possible times. Therefore, these two expectations are equal.

We then rearrange equation (6.24) to be

$$\sum_{k=1}^{K}\left(\lambda^*_k(t\mid x^0_{[0,t)})\mathrm{d}t\log\frac{\lambda_k(t\mid x^0_{[0,t)})}{\underline{\lambda}_k(t\mid x^0_{[0,t)})}+\sum_{m=1}^{M}\lambda^{\mathrm{q}}_k(t\mid x^0_{[0,t)})\mathrm{d}t\log\frac{\lambda^{\mathrm{q}}_k(t\mid x^0_{[0,t)})}{\underline{\lambda}_k(t\mid x^0_{[0,t)})}\right) \tag{6.25}$$

where each k-specific term can be further rearranged to be

$$\lambda_k^*(t \mid x_{[0,t)}^0)\mathrm{d}t \log \frac{\lambda_k(t \mid x_{[0,t)}^0)}{\underline{\lambda}_k(t \mid x_{[0,t)}^0)} + M\lambda_k^{\mathrm{q}}(t \mid x_{[0,t)}^0)\mathrm{d}t \log \frac{\lambda_k^{\mathrm{q}}(t \mid x_{[0,t)}^0)}{\underline{\lambda}_k(t \mid x_{[0,t)}^0)} \tag{6.26a}$$

$$= \underline{\lambda}_k^*(t \mid x_{[0,t)}^0)\mathrm{d}t \underbrace{\left(\frac{\lambda_k^*(t \mid x_{[0,t)}^0)}{\underline{\lambda}_k^*(t \mid x_{[0,t)}^0)} \log \frac{\lambda_k(t \mid x_{[0,t)}^0)}{\underline{\lambda}_k(t \mid x_{[0,t)}^0)} + M \frac{\lambda_k^{\mathrm{q}}(t \mid x_{[0,t)}^0)}{\underline{\lambda}_k^*(t \mid x_{[0,t)}^0)} \log \frac{\lambda_k^{\mathrm{q}}(t \mid x_{[0,t)}^0)}{\underline{\lambda}_k(t \mid x_{[0,t)}^0)} \right)}_{\text{call it } H_{\boldsymbol{\theta}}(k,t,x_{[0,t)}^0)} \tag{6.26b}$$

where $\underline{\lambda}_k^*(t \mid x_{[0,t)}^0) \stackrel{\text{def}}{=} \lambda_k^*(t \mid x_{[0,t)}^0) + M\lambda_k^{\mathrm{q}}(t \mid x_{[0,t)}^0)$ can be thought of as the intensity of type k under the superposition of p^* and M copies of q.

That is, we have

$$H_{\boldsymbol{\theta}}(t, x_{[0,t)}^0) = \sum_{k=1}^{K} \underline{\lambda}_k^*(t \mid x_{[0,t)}^0) H_{\boldsymbol{\theta}}(k, t, x_{[0,t)}^0) \tag{6.27}$$

Plugging equation (6.27) into equation (6.23) gives us the final rearranged objective:

$$J_{\mathrm{NC}}(\boldsymbol{\theta}) = \int_{t=0}^{T} \sum_{x_{[0,t)}^0} p^*(x_{[0,t)}^0) \sum_{k=1}^{K} \underline{\lambda}_k^*(t \mid x_{[0,t)}^0) H_{\boldsymbol{\theta}}(k, t, x_{[0,t)}^0)\mathrm{d}t \tag{6.28}$$

6.B.2 Optimality Proof Details

In this section, we prove Theorem 1 that we stated in section 6.3.5. Recall the theorem:

Theorem 1 (Optimality)**.** *Under assumptions 1 and 2,* $\boldsymbol{\theta} \in \arg\max_{\boldsymbol{\theta}} J_{\mathrm{NC}}(\boldsymbol{\theta})$ *if and only if* $p_{\boldsymbol{\theta}} = p^*$.

We first need to highlight the key insight that $H_{\boldsymbol{\theta}}(k, t, x_{[0,t)}^0)$ in equation (6.28) is the negative cross-entropy between the following two discrete distributions over

$\{\varnothing, 1, \ldots, K\}$:

$$[\frac{\lambda_k^*(t \mid x_{[0,t)}^0)}{\underline{\lambda}_k^*(t \mid x_{[0,t)}^0)}, \frac{\lambda_k^q(t \mid x_{[0,t)}^0)}{\underline{\lambda}_k^*(t \mid x_{[0,t)}^0)}, \ldots, \frac{\lambda_k^q(t \mid x_{[0,t)}^0)}{\underline{\lambda}_k^*(t \mid x_{[0,t)}^0)}] \tag{6.29a}$$

$$[\frac{\lambda_k(t \mid x_{[0,t)}^0)}{\underline{\lambda}_k(t \mid x_{[0,t)}^0)}, \underbrace{\frac{\lambda_k^q(t \mid x_{[0,t)}^0)}{\underline{\lambda}_k(t \mid x_{[0,t)}^0)}, \ldots, \frac{\lambda_k^q(t \mid x_{[0,t)}^0)}{\underline{\lambda}_k(t \mid x_{[0,t)}^0)}}_{\text{length is } M}] \tag{6.29b}$$

This negative cross-entropy is always smaller than or equal to the negative entropy of the distribution in equation (6.29a): it will be strictly smaller if these two distributions are distinct and equal when they are identical. Notice that in contrast to the negative cross-entropy at equation (6.13), this negative cross-entropy here is not infinitesimal.

Proof. The "if" part is straightforward to prove. Any $\boldsymbol{\theta}$ for which $p_{\boldsymbol{\theta}} = p^*$ would make $\lambda_k(t \mid x_{[0,t)}^0) = \lambda_k^*(t \mid x_{[0,t)}^0)$, thus maximizing the negative cross-entropy between the two distributions in equation (6.29), for any type k and any real history $x_{[0,t)}^0$ at any time t. Then the NCE objective in equation (6.28) is obviously maximized.

To check if any other $\bar{\boldsymbol{\theta}} \notin \Theta^* \stackrel{\text{def}}{=} \{\boldsymbol{\theta}^* : p_{\boldsymbol{\theta}^*} = p^*\}$ maximizes $J_{\text{NC}}(\boldsymbol{\theta})$ as well, we analyze $J_{\text{NC}}(\bar{\boldsymbol{\theta}}) - J_{\text{NC}}(\boldsymbol{\theta}^*)$ which is equal to

$$= \int_{t=0}^{T} \sum_{x_{[0,t)}^0} p^*(x_{[0,t)}^0) \sum_{k=1}^{K} \underline{\lambda}_k^*(t \mid x_{[0,t)}^0) \underbrace{\Big(H_{\bar{\boldsymbol{\theta}}}(k, t, x_{[0,t)}^0) - H_{\boldsymbol{\theta}^*}(k, t, x_{[0,t)}^0)\Big)}_{\text{denote it as } G_{\bar{\boldsymbol{\theta}}}(k,t,x_{[0,t)}^0)} \mathrm{d}t$$

where $\boldsymbol{\theta}^*$ can be any member in Θ^*. Note that $G_{\bar{\boldsymbol{\theta}}}$ is not infinitesimal because the probabilities in H and thus the entropy changes (if any) are not infinitesimal.

According to the definition of $\bar{\boldsymbol{\theta}}$ and $\boldsymbol{\theta}^*$, there must exist a sequence $\bar{x}_{[0,T)}$, a time $\bar{t} \in (0, T)$ and a type $\bar{k} \in \{1, \ldots, K\}$ such that $\lambda_{\bar{k}}(\bar{t} \mid \bar{x}_{[0,\bar{t})}) \neq \lambda_{\bar{k}}^*(\bar{t} \mid \bar{x}_{[0,\bar{t})})$. Therefore, we have $G_{\bar{\boldsymbol{\theta}}}(\bar{k}, \bar{t}, \bar{x}_{[0,\bar{t})}) < 0$ since the distributions in equation (6.29) are distinct for the given history $\bar{x}_{[0,\bar{t})}$. Does this difference lead to any overall change of

the entire objective?

Actually, according to Lemma 1 in section 6.A, the existence of such $\bar{x}_{[0,T)}$, $\bar{t}$ and $\bar{k}$ implies that there exists an interval $(t', t'') \subset [0, T)$ such that, for any $t \in (t', t'')$, there exists a set $\mathcal{X}(t)$ of histories with non-zero measure such that any $x_{[0,t)} \in \mathcal{X}(t)$ satisfies $\lambda_{\bar{k}}(t \mid x_{[0,t)}) \neq \lambda^*_{\bar{k}}(t \mid x_{[0,t)})$. Then, given any of these histories, the entropy difference $G_{\bar{\boldsymbol{\theta}}}$ would be < 0. That is to say, the following integral must be a non-infinitesimal negative number:

$$\int_{t=0}^{T} \sum_{x^0_{[0,t)}} p^*(x^0_{[0,t)}) \underline{\lambda}^*_{\bar{k}}(t \mid x^0_{[0,t)}) G_{\bar{\boldsymbol{\theta}}}(\bar{k}, t, x^0_{[0,t)}) \mathrm{d}t \tag{6.30a}$$

$$= \int_{t=t'}^{t''} \sum_{x^0_{[0,t)} \in \mathcal{X}(t)} p^*(x^0_{[0,t)}) \underline{\lambda}^*_{\bar{k}}(t \mid x^0_{[0,t)}) G_{\bar{\boldsymbol{\theta}}}(\bar{k}, t, x^0_{[0,t)}) \mathrm{d}t \qquad (< 0) \tag{6.30b}$$

$$+ \int_{t=t'}^{t''} \sum_{x^0_{[0,t)} \notin \mathcal{X}(t)} p^*(x^0_{[0,t)}) \underline{\lambda}^*_{\bar{k}}(t \mid x^0_{[0,t)}) G_{\bar{\boldsymbol{\theta}}}(\bar{k}, t, x^0_{[0,t)}) \mathrm{d}t \qquad (\leq 0) \tag{6.30c}$$

$$+ \int_{t=0}^{t'} \sum_{x^0_{[0,t)}} p^*(x^0_{[0,t)}) \underline{\lambda}^*_{\bar{k}}(t \mid x^0_{[0,t)}) G_{\bar{\boldsymbol{\theta}}}(\bar{k}, t, x^0_{[0,t)}) \mathrm{d}t \qquad (\leq 0) \tag{6.30d}$$

$$+ \int_{t=t''}^{T} \sum_{x^0_{[0,t)}} p^*(x^0_{[0,t)}) \underline{\lambda}^*_{\bar{k}}(t \mid x^0_{[0,t)}) G_{\bar{\boldsymbol{\theta}}}(\bar{k}, t, x^0_{[0,t)}) \mathrm{d}t \qquad (\leq 0) \tag{6.30e}$$

Therefore, the overall difference must be < 0 as well:

$$J_{\mathrm{LL}}(\bar{\boldsymbol{\theta}}) - J_{\mathrm{LL}}(\boldsymbol{\theta}^*) \tag{6.31a}$$

$$= \int_{t=0}^{T} \sum_{x^0_{[0,t)}} p^*(x^0_{[0,t)}) \sum_{k=1}^{K} \underline{\lambda}^*_{k}(t \mid x^0_{[0,t)}) G_{\bar{\boldsymbol{\theta}}}(k, t, x^0_{[0,t)}) \mathrm{d}t \tag{6.31b}$$

$$= \int_{t=0}^{T} \sum_{x^0_{[0,t)}} p^*(x^0_{[0,t)}) \underline{\lambda}^*_{\bar{k}}(t \mid x^0_{[0,t)}) G_{\bar{\boldsymbol{\theta}}}(\bar{k}, t, x^0_{[0,t)}) \mathrm{d}t \qquad (< 0) \tag{6.31c}$$

$$+\int_{t=0}^{T}\sum_{x^0_{[0,t)}} p^*(x^0_{[0,t)})\sum_{k\neq\bar{k}}\underline{\lambda}^*_k(t\mid x^0_{[0,t)})G_{\bar{\boldsymbol{\theta}}}(k,t,x^0_{[0,t)})\mathrm{d}t \qquad (\leq 0\,) \qquad (6.31\mathrm{d})$$

Note that $J_{\mathrm{LL}}(\bar{\boldsymbol{\theta}}) - J_{\mathrm{LL}}(\boldsymbol{\theta}^*) < 0$ holds any $\bar{\boldsymbol{\theta}} \notin \Theta^*$ and any $\boldsymbol{\theta}^* \in \Theta^*$, meaning that $\boldsymbol{\theta}^* \in \Theta^*$ is necessary to maximize the objective. Then the proof of the "only if" part is complete.

Now we have proved both the "if" and "only if" parts so the proof is complete.

□

6.B.3 Consistency Proof Details

To discuss the statistical consistency (in this section) and efficiency (in section 6.B.4), we first need to spell out the empirical version of the objective

$$J^N_{\mathrm{NC}}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N}\left(\sum_{t:x^0_{t,n}\neq\varnothing}\log\frac{\lambda_{x^0_{t,n}}(t\mid x^0_{[0,t),n})}{\underline{\lambda}_{x^0_{t,n}}(t\mid x^0_{[0,t),n})} + \sum_{m=1}^{M}\sum_{t:x^m_{t,n}\neq\varnothing}\log\frac{\lambda^{\mathrm{q}}_{x^m_{t,n}}(t\mid x^0_{[0,t),n})}{\underline{\lambda}_{x^m_{t,n}}(t\mid x^0_{[0,t),n})}\right) \qquad (6.32)$$

where the subscript $_n$ denotes the n^{th} i.i.d. draw of the observed sequence and the M noise samples for this sequence. It is obvious that $\lim_{N\to\infty} J^N_{\mathrm{NC}}(\boldsymbol{\theta}) \to J_{\mathrm{NC}}(\boldsymbol{\theta})$.

To analyze the consistency, we make the following assumptions:

Assumption 3 (Continuity wrt. $\boldsymbol{\theta}$)**.** *For any history $x_{[0,t)}$ and event type $k \in \{1,\ldots,K\}$, $\lambda_k(t \mid x_{[0,t)})$ is continuous with respect to $\boldsymbol{\theta}$.*

Assumption 4 (Compactness)**.** *The set of optimal parameters Θ^* is contained in the interior of a compact set $\Theta \subset \mathbb{R}^{|\boldsymbol{\theta}|}$.*

They are analogous to assumptions 4.2 and 4.3 of Ma and Collins (2018) respectively.

Our NCE method turns out to be strongly consistent in the sense that:

Theorem 2 (Consistency). *Under assumptions 2–4, for any* $\boldsymbol{\theta} \in \Theta_{\text{NC}}^N \stackrel{def}{=} \arg\max_{\boldsymbol{\theta}} J_{\text{NC}}^N(\boldsymbol{\theta})$ *and* $M \geq 1$*, with probability 1, we have* $\lim_{N\to\infty} \min_{\boldsymbol{\theta}^* \in \Theta^*} \|\boldsymbol{\theta} - \boldsymbol{\theta}^*\| = 0$ *where* $\|\cdot\|$ *is the* L_2 *norm.*

The intuition of this theorem is that: since the two functions $J_{\text{NC}}^N(\boldsymbol{\theta})$ and $J_{\text{NC}}(\boldsymbol{\theta})$ will become the same as $N \to \infty$ and they are continuous with respect to $\boldsymbol{\theta}$, then any $\boldsymbol{\theta} \in \arg\max_{\boldsymbol{\theta}} J_{\text{NC}}^N(\boldsymbol{\theta})$ has to be close to some member of the set $\arg\max_{\boldsymbol{\theta}} J_{\text{NC}}(\boldsymbol{\theta})$. The full proof is almost identical to the proof of Theorem 4.2 in Ma and Collins (2018). But we will still spell it out in our notation for completeness.

Proof. Under the assumption in Theorem 2, by classical large sample theory (Ferguson, 1996), we have

$$\mathbb{P}\left[\lim_{N\to\infty} \sup_{\boldsymbol{\theta}\in\Theta'} |J_{\text{NC}}^N(\boldsymbol{\theta}) - J_{\text{NC}}(\boldsymbol{\theta})| = 0\right] = 1 \text{ for any compact set } \Theta' \subset \Theta \tag{6.33}$$

where $\mathbb{P}$ stands for "probability". Since $|J_{\text{NC}}^N(\boldsymbol{\theta}) - J_{\text{NC}}(\boldsymbol{\theta})| \geq J_{\text{NC}}^N(\boldsymbol{\theta}) - J_{\text{NC}}(\boldsymbol{\theta})$, we have

$$\mathbb{P}\left[\limsup_{N\to\infty} \sup_{\boldsymbol{\theta}\in\Theta'} (J_{\text{NC}}^N(\boldsymbol{\theta}) - J_{\text{NC}}(\boldsymbol{\theta})) \leq 0\right] = 1 \tag{6.34}$$

Moreover, for any $\boldsymbol{\theta}'^N \in \arg\max_{\boldsymbol{\theta}\in\Theta'} J_{\text{NC}}^N(\boldsymbol{\theta})$, we have

$$\sup_{\boldsymbol{\theta}\in\Theta'} (J_{\text{NC}}^N(\boldsymbol{\theta}) - J_{\text{NC}}(\boldsymbol{\theta})) \geq J_{\text{NC}}^N(\boldsymbol{\theta}'^N) - J_{\text{NC}}(\boldsymbol{\theta}'^N) \geq \sup_{\boldsymbol{\theta}\in\Theta'} J_{\text{NC}}^N(\boldsymbol{\theta}) - \sup_{\boldsymbol{\theta}\in\Theta'} J_{\text{NC}}(\boldsymbol{\theta}) \tag{6.35}$$

Plugging equation (6.35) into equation (6.34) gives

$$\mathbb{P}\left[\limsup_{N\to\infty} \sup_{\boldsymbol{\theta}\in\Theta'} J_{\text{NC}}^N(\boldsymbol{\theta}) - \sup_{\boldsymbol{\theta}\in\Theta'} J_{\text{NC}}(\boldsymbol{\theta}) \leq 0\right] \tag{6.36a}$$

$$= \mathbb{P}\left[\limsup_{N\to\infty} \sup_{\boldsymbol{\theta}\in\Theta'} J_{\text{NC}}^N(\boldsymbol{\theta}) \leq \sup_{\boldsymbol{\theta}\in\Theta'} J_{\text{NC}}(\boldsymbol{\theta})\right] \tag{6.36b}$$

$$=1 \tag{6.36c}$$

For any $\delta > 0$, we define $\Theta_\delta \stackrel{\text{def}}{=} \{\boldsymbol{\theta} : \min_{\boldsymbol{\theta}^* \in \Theta^*} \|\boldsymbol{\theta} - \boldsymbol{\theta}^*\| > \delta\}$ and have

$$\mathbb{P}\left[\limsup_{N\to\infty} \sup_{\boldsymbol{\theta}\in\Theta_\delta} J_{\mathrm{NC}}^N(\boldsymbol{\theta}) \leq \sup_{\boldsymbol{\theta}\in\Theta_\delta} J_{\mathrm{NC}}(\boldsymbol{\theta}) < \sup_{\boldsymbol{\theta}\in\Theta} J_{\mathrm{NC}}(\boldsymbol{\theta})\right] = 1 \tag{6.37}$$

On the other hand, we also have $|J_{\mathrm{NC}}^N(\boldsymbol{\theta}) - J_{\mathrm{NC}}(\boldsymbol{\theta})| \geq J_{\mathrm{NC}}(\boldsymbol{\theta}) - J_{\mathrm{NC}}^N(\boldsymbol{\theta})$, which gives

$$\mathbb{P}\left[\limsup_{N\to\infty} \sup_{\boldsymbol{\theta}\in\Theta'} (J_{\mathrm{NC}}(\boldsymbol{\theta}) - J_{\mathrm{NC}}^N(\boldsymbol{\theta})) \leq 0\right] = 1 \tag{6.38}$$

For any $\boldsymbol{\theta}' \in \arg\max_{\boldsymbol{\theta}\in\Theta'} J_{\mathrm{NC}}(\boldsymbol{\theta})$, we have

$$\sup_{\boldsymbol{\theta}\in\Theta'} (J_{\mathrm{NC}}(\boldsymbol{\theta}) - J_{\mathrm{NC}}^N(\boldsymbol{\theta})) \geq J_{\mathrm{NC}}(\boldsymbol{\theta}') - J_{\mathrm{NC}}^N(\boldsymbol{\theta}') \geq \sup_{\boldsymbol{\theta}\in\Theta'} J_{\mathrm{NC}}(\boldsymbol{\theta}) - \sup_{\boldsymbol{\theta}\in\Theta'} J_{\mathrm{NC}}^N(\boldsymbol{\theta}) \tag{6.39}$$

Plugging equation (6.39) into equation (6.38) gives

$$\mathbb{P}\left[\sup_{\boldsymbol{\theta}\in\Theta'} J_{\mathrm{NC}}(\boldsymbol{\theta}) + \limsup_{N\to\infty} (-\sup_{\boldsymbol{\theta}\in\Theta'} J_{\mathrm{NC}}^N(\boldsymbol{\theta})) \leq 0\right] \tag{6.40a}$$

$$=\mathbb{P}\left[\liminf_{N\to\infty} \sup_{\boldsymbol{\theta}\in\Theta'} J_{\mathrm{NC}}^N(\boldsymbol{\theta}) \geq \sup_{\boldsymbol{\theta}\in\Theta'} J_{\mathrm{NC}}(\boldsymbol{\theta})\right] \tag{6.40b}$$

$$=1 \tag{6.40c}$$

which, when we let $\Theta' = \Theta$, gives

$$\mathbb{P}\left[\liminf_{N\to\infty} \sup_{\boldsymbol{\theta}\in\Theta} J_{\mathrm{NC}}^N(\boldsymbol{\theta}) \geq \sup_{\boldsymbol{\theta}\in\Theta} J_{\mathrm{NC}}(\boldsymbol{\theta})\right] = 1 \tag{6.41}$$

Combining equation (6.37) and equation (6.41), we have that, for any $\boldsymbol{\theta}^N \in \Theta^N \stackrel{\text{def}}{=} \arg\max_{\boldsymbol{\theta}} J_{\mathrm{NC}}^N(\boldsymbol{\theta})$ (defined in Theorem 2), there exists an integer N' such that for any

$N \geq N'$

$$\mathbb{P}\left[\boldsymbol{\theta}^N \notin \Theta_\delta\right] = 1 \tag{6.42}$$

which holds for any $\delta > 0$ and thus gives

$$\mathbb{P}\left[\lim_{N\to\infty} \min_{\boldsymbol{\theta}^* \in \Theta^*} \|\boldsymbol{\theta}^N - \boldsymbol{\theta}^*\| = 0\right] = 1 \tag{6.43}$$

which completes the proof of Theorem 2.

□

6.B.4 Efficiency Proof Details

To quantify the statistical efficiency of our method, we make the following assumptions:

Assumption 5 (Identifiability)**.** *There is only one parameter vector $\boldsymbol{\theta}^*$ such that $p_{\boldsymbol{\theta}^*} = p^*$.*

Assumption 6 (Differentiability)**.** *For any history $x_{[0,t)}$ and event type $k \in \{1, \ldots, K\}$, $\lambda_k(t \mid x_{[0,t)})$ is twice continuously differentiable with respect to $\boldsymbol{\theta}$.*

Assumption 7 (Singularity)**.** *The Fisher information matrix $\mathbf{I}_*$ under the model $p_{\boldsymbol{\theta}}$ is non-singular.*

They are analogous to assumptions 4.4, 4.6 and 4.7 of Ma and Collins (2018) respectively.

Before we show the efficiency of our method, we first spell out the definition of $\mathbf{I}_*$:

$$\mathbf{I}_* \stackrel{\text{def}}{=} \mathbb{E}_{x_{[0,T)} \sim p^*}[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}^*}(x_{[0,T)}) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}^*}(x_{[0,T)})^\top] \tag{6.44}$$

where $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}^*}$ stands for "the gradient of $\log p_{\boldsymbol{\theta}}$ with respect to $\boldsymbol{\theta}$ at $\boldsymbol{\theta} = \boldsymbol{\theta}^*$." This formula can be rearranged as

$$\int_{t=0}^{T} \mathbb{E}_{x_{[0,t)} \sim p^*} [\mathbb{E}_{x_{[t,t+\mathrm{d}t)} \sim p^*} [\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}^*}(x_{[t,t+\mathrm{d}t)} \mid x_{[0,t)}) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}^*}(x_{[t,t+\mathrm{d}t)} \mid x_{[0,t)})^\top]] \tag{6.45a}$$

$$= \int_{t=0}^{T} \mathbb{E}_{x_{[0,t)} \sim p^*} [\mathbb{E}_{x_{[t,t+\mathrm{d}t)} \sim p^*} [\frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}^*}(x_{[t,t+\mathrm{d}t)} \mid x_{[0,t)})}{p_{\boldsymbol{\theta}^*}(x_{[t,t+\mathrm{d}t)} \mid x_{[0,t)})} \frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}^*}(x_{[t,t+\mathrm{d}t)} \mid x_{[0,t)})^\top}{p_{\boldsymbol{\theta}^*}(x_{[t,t+\mathrm{d}t)} \mid x_{[0,t)})}]] \tag{6.45b}$$

$$= \int_{t=0}^{T} \mathbb{E}_{x_{[0,t)} \sim p^*} [\sum_{x_{[t,t+\mathrm{d}t)}} \frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}^*}(x_{[t,t+\mathrm{d}t)} \mid x_{[0,t)}) \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}^*}(x_{[t,t+\mathrm{d}t)} \mid x_{[0,t)})^\top}{p_{\boldsymbol{\theta}^*}(x_{[t,t+\mathrm{d}t)} \mid x_{[0,t)})}] \tag{6.45c}$$

Technically, $x_{[t,t+\mathrm{d}t)}$ will have an event of type k with probability $\lambda_k^*(t)\mathrm{d}t$ under p^* ($\lambda_k(t)\mathrm{d}t$ under $p_{\boldsymbol{\theta}}$) or has no event at all with probability $1 - \sum_{k=1}^K \lambda_k^*(t)\mathrm{d}t$ under p^* ($1 - \sum_{k=1}^K \lambda_k(t)\mathrm{d}t$ under $p_{\boldsymbol{\theta}}$). In the former case, we have $\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}^*} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}^*}^\top / p_{\boldsymbol{\theta}^*} = \nabla_{\boldsymbol{\theta}} \lambda_k^*(t) \nabla_{\boldsymbol{\theta}} \lambda_k^*(t)^\top \mathrm{d}t / \lambda_k^*(t)$; in the latter case, we have $\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}^*} = -\sum_{k=1}^K \nabla_{\boldsymbol{\theta}} \lambda_k^*(t)\mathrm{d}t$ but $p_{\boldsymbol{\theta}^*} \approx 1$, so $\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}^*} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}^*}^\top / p_{\boldsymbol{\theta}^*} = o(\mathrm{d}t)$ can be ignored. Plugging these quantities into equation (6.45) gives us

$$\mathbf{I}_* = \int_{t=0}^{T} \mathbb{E}_{x_{[0,t)} \sim p^*} [\sum_{k=1}^{K} \frac{\nabla_{\boldsymbol{\theta}} \lambda_k^*(t \mid x_{[0,t)}) \nabla_{\boldsymbol{\theta}} \lambda_k^*(t \mid x_{[0,t)})^\top}{\lambda_k^*(t \mid x_{[0,t)})} \mathrm{d}t] \tag{6.46a}$$

$$= \int_{t=0}^{T} \sum_{x_{[0,t)}} p^*(x_{[0,t)}) \sum_{k=1}^{K} \frac{\nabla_{\boldsymbol{\theta}} \lambda_k^*(t \mid x_{[0,t)}) \nabla_{\boldsymbol{\theta}} \lambda_k^*(t \mid x_{[0,t)})^\top}{\lambda_k^*(t \mid x_{[0,t)})} \mathrm{d}t \tag{6.46b}$$

Note that $\nabla_{\boldsymbol{\theta}} \lambda_k^*(t)$ stands for "the gradient of $\lambda_k(t)$ with respect to $\boldsymbol{\theta}$ at $\boldsymbol{\theta} = \boldsymbol{\theta}^*$."

Now we proceed to our efficiency theorem. We denote the unique optimal parameter vector as $\boldsymbol{\theta}^*$ and use $\hat{\boldsymbol{\theta}}$ for the estimate given by maximizing $J_{\mathrm{NC}}^N(\boldsymbol{\theta})$. It turns out that our method approaches *Fisher efficiency* as M grows.

Theorem 3 (Efficiency)**.** *Under assumptions 2 and 4–7, there exists an integer* $\bar{M}$

such that for all $M > \bar{M}$

$$\sqrt{N}(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}^*) \to \text{Normal}(0, \mathbf{I}_M^{-1}) \text{ as } N \to \infty \tag{6.47}$$

for some non-singular matrix $\mathbf{I}_M^{-1}$*. Moreover, there exist a constant* $C > 0$ *such that for all* $M > \bar{M}$

$$\|\mathbf{I}_M^{-1} - \mathbf{I}_*^{-1}\| \leq C/M \tag{6.48}$$

where $\|\mathbf{I}\|$ *is the spectral norm of matrix* $\mathbf{I}$*.*

Proof. We first prove that $\sqrt{N}(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}^*)$ is asymptotically normal. By the Mean-Value Theorem, we have

$$\nabla_{\boldsymbol{\theta}} J_{\text{NC}}^N(\hat{\boldsymbol{\theta}}) = \nabla_{\boldsymbol{\theta}} J_{\text{NC}}^N(\boldsymbol{\theta}^*) + (\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}^*) \int_{u=0}^1 \nabla_{\boldsymbol{\theta}}^2 J_{\text{NC}}^N(\boldsymbol{\theta}^* + u(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}^*))\mathrm{d}t \tag{6.49}$$

Since $\hat{\boldsymbol{\theta}}$ maximizes J_{NC}^N, we have

$$\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}^* = \left[- \int_{u=0}^1 \nabla_{\boldsymbol{\theta}}^2 J_{\text{NC}}^N(\boldsymbol{\theta}^* + u(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}^*))\mathrm{d}t \right]^{-1} \nabla_{\boldsymbol{\theta}} J_{\text{NC}}^N(\boldsymbol{\theta}^*) \tag{6.50}$$

By Law of Large Numbers and Theorem 2, we have

$$\int_{u=0}^1 \nabla_{\boldsymbol{\theta}}^2 J_{\text{NC}}^N(\boldsymbol{\theta}^* + u(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}^*))\mathrm{d}t \to \underbrace{\mathbb{E}_{x_{[0,T)}^0 \sim p^*, x_{[0,T)}^{1:M} \sim q}[\nabla_{\boldsymbol{\theta}}^2 L(\boldsymbol{\theta}^*)]}_{\text{short as } \mathbb{E}[\nabla_{\boldsymbol{\theta}}^2 L(\boldsymbol{\theta}^*)]} \text{ as } N \to \infty \tag{6.51}$$

where $L(\boldsymbol{\theta})$ is defined as the objective for a random draw of $x_{[0,T)}^{0:M}$ and thus is just the term inside the expectation of equation (6.6):

$$L(\boldsymbol{\theta}) \stackrel{\text{def}}{=} \sum_{t: x_t^0 \neq \varnothing} \log \frac{\lambda_{x_t^0}(t \mid x_{[0,t)}^0)}{\underline{\lambda}_{x_t^0}(t \mid x_{[0,t)}^0)} + \sum_{m=1}^M \sum_{t: x_t^m \neq \varnothing} \log \frac{\lambda_{x_t^m}^{\text{q}}(t \mid x_{[0,t)}^0)}{\underline{\lambda}_{x_t^m}(t \mid x_{[0,t)}^0)} \tag{6.52}$$

The term $\nabla_{\boldsymbol{\theta}}^2 L(\boldsymbol{\theta}^*)$ stands for "the Hessian matrix of $L(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$ at $\boldsymbol{\theta} = \boldsymbol{\theta}^*$."

As for $\nabla_{\boldsymbol{\theta}} J_{\mathrm{NC}}^N(\boldsymbol{\theta}^*)$, by Central Limit Theorem, we have

$$\sqrt{N}\nabla_{\boldsymbol{\theta}} J_{\mathrm{NC}}^N(\boldsymbol{\theta}^*) \to \mathrm{Normal}(0, \underbrace{\mathbb{E}_{x_{[0,T)}^0 \sim p^*, x_{[0,T)}^{1:M} \sim q}[\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^*) \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^*)^\top]}_{\text{short as } \mathbb{V}[\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^*)]}) \tag{6.53}$$

Combining equations (6.50), (6.51) and (6.53), we obtain the asymptotic normality

$$\sqrt{N}(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}^*) \to \mathrm{Normal}(0, \mathbb{E}[\nabla_{\boldsymbol{\theta}}^2 L(\boldsymbol{\theta}^*)]^{-1} \mathbb{V}[\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^*)] \mathbb{E}[\nabla_{\boldsymbol{\theta}}^2 L(\boldsymbol{\theta}^*)]^{-1}) \tag{6.54}$$

Now we compute the covariance matrix of the asymptotic normal distribution. Following steps similar to equations (6.22), (6.23), (6.25) and (6.26), we rearrange $\mathbb{E}[\nabla_{\boldsymbol{\theta}}^2 L(\boldsymbol{\theta}^*)]$ to be

$$\int_{t=0}^{T} \mathbb{E}_{x_{[0,t)}^0 \sim p^*}[\sum_{k=1}^{K} \left(\lambda_k^*(t) \mathrm{d}t \nabla_{\boldsymbol{\theta}}^2 \log \frac{\lambda_k^*(t)}{\underline{\lambda}_k^*(t)} + M \lambda_k^{\mathrm{q}}(t) \mathrm{d}t \nabla_{\boldsymbol{\theta}}^2 \log \frac{\lambda_k^{\mathrm{q}}(t)}{\underline{\lambda}_k^*(t)} \right)] \tag{6.55a}$$

$$= \int_{t=0}^{T} \mathbb{E}_{x_{[0,t)}^0 \sim p^*}[\sum_{k=1}^{K} (\frac{1}{\underline{\lambda}_k^*(t)} - \frac{1}{\lambda_k^*(t)}) \nabla_{\boldsymbol{\theta}} \lambda_k^*(t) \nabla_{\boldsymbol{\theta}} \lambda_k^*(t)^\top \mathrm{d}t] \tag{6.55b}$$

$$= \int_{t=0}^{T} p^*(x_{[0,t)}^0) \sum_{k=1}^{K} (\frac{1}{\underline{\lambda}_k^*(t)} - \frac{1}{\lambda_k^*(t)}) \nabla_{\boldsymbol{\theta}} \lambda_k^*(t) \nabla_{\boldsymbol{\theta}} \lambda_k^*(t)^\top \mathrm{d}t \tag{6.55c}$$

where we omit the condition $x_{[0,t)}^0$ in the probabilities and intensities for presentation simplicity. We also omit the tedious arithmetic manipulation that spells $\nabla_{\boldsymbol{\theta}}^2 \log(\lambda/\underline{\lambda})$ out.

Following similar steps, we then rearrange $\mathbb{V}[\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^*)]$ to be

$$\int_{t=0}^{T} \mathbb{E}_{x_{[0,t)}^0 \sim p^*}[\sum_{k=1}^{K} \left(\lambda_k^*(t) \mathrm{d}t \nabla_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}}^\top \log \frac{\lambda_k^*(t)}{\underline{\lambda}_k^*(t)} + M \lambda_k^{\mathrm{q}}(t) \mathrm{d}t \nabla_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}}^\top \log \frac{\lambda_k^q(t)}{\underline{\lambda}_k^*(t)}) \right)] \tag{6.56a}$$

$$= \int_{t=0}^{T} \mathbb{E}_{x_{[0,t)}^0 \sim p^*}[\sum_{k=1}^{K} (\frac{1}{\lambda_k^*(t)} - \frac{1}{\underline{\lambda}_k^*(t)}) \nabla_{\boldsymbol{\theta}} \lambda_k^*(t) \nabla_{\boldsymbol{\theta}} \lambda_k^*(t)^\top \mathrm{d}t] \tag{6.56b}$$

$$=\mathbb{E}[-\nabla^2_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^*)] \tag{6.56c}$$

where we use $\nabla_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}}^\top f(\boldsymbol{\theta})$ to denote $(\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}))(\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}))^\top$. For presentation simplicity, we omit the arithmetic manipulation that spells $\nabla_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}}^\top \log(\lambda/\underline{\lambda})$ out.

Then we can simplify the asymptotic normality to be

$$\sqrt{N}(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}^*) \to \mathrm{Normal}(0, \mathbb{E}[-\nabla^2_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^*)]^{-1}) \tag{6.57}$$

We can think of $\mathbf{I}_M \stackrel{\text{def}}{=} \mathbb{E}[-\nabla^2_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^*)]$ as the "information matrix" of our objective $J_{\mathrm{NC}}(\boldsymbol{\theta})$. And its relation with the Fisher information matrix $\mathbf{I}_*$ is:

$$\mathbf{I}_M = \mathbf{I}_* - \underbrace{\int_{t=0}^{T} \sum_{x^0_{[0,t)}} p^*(x^0_{[0,t)}) \sum_{k=1}^{K} \frac{1}{\lambda_k^*(t) + M\lambda_k^{\mathrm{q}}(t)} \nabla_{\boldsymbol{\theta}} \lambda_k^*(t) \nabla_{\boldsymbol{\theta}} \lambda_k^*(t)^\top \mathrm{d}t}_{\text{call it } \Delta\mathbf{I}} \tag{6.58}$$

Apparently, when M is large enough, $\mathbf{I}_M$ will be non-singular. Precisely, since $\mathbf{I}_*$ is non-singular, there must exist $\bar{M} > 0$ such that, for any $M > \bar{M}$, $0 < \|\Delta\mathbf{I}\| \leq \sigma(\mathbf{I}_*)/2$ where $\sigma(\mathbf{I})$ is the *smallest* singular value of matrix $\mathbf{I}$ and $\|\mathbf{I}\|$ is the *spectral norm*, i.e., the *largest* singular value, of matrix $\mathbf{I}$. By Weyl's inequality, we have $\sigma(\mathbf{I}_M) \geq \sigma(\mathbf{I}_*) - \|\Delta\mathbf{I}\| \geq \sigma(\mathbf{I}_*)/2$, meaning that $\mathbf{I}_M$ is non-singular.

Now we can start analyzing $\|\mathbf{I}_M^{-1} - \mathbf{I}_*^{-1}\|$. By the definition of the spectral norm, we have:

$$\|\mathbf{I}_M^{-1} - \mathbf{I}_*^{-1}\| = \|\mathbf{I}_*^{-1}(\mathbf{I}_* - \mathbf{I}_M)\mathbf{I}_M^{-1}\| \leq \|\mathbf{I}_*^{-1}\| \|\Delta\mathbf{I}\| \|\mathbf{I}_M^{-1}\| \leq \frac{1}{\sigma(\mathbf{I}_*)} \|\Delta\mathbf{I}\| \frac{2}{\sigma(\mathbf{I}_*)} \tag{6.59}$$

Since the intensity functions are all bounded, continuous and twice continuously differentiable, $\|\nabla_{\boldsymbol{\theta}} \lambda_k^*(t) \nabla_{\boldsymbol{\theta}} \lambda_k^*(t)^\top\|$ will be bounded, meaning that $\|\Delta\mathbf{I}\|$ will be bounded as well. Moreover, the ratio $\lambda_k^*(t)/\lambda_k^{\mathrm{q}}(t)$ is also bounded. We define $r = \sup_{x^0_{[0,t)}, k} \frac{\lambda_k^*(t|x^0_{[0,t)})}{\lambda_k^{\mathrm{q}}(t|x^0_{[0,t)})}$ and have $M\lambda_k^{\mathrm{q}}(t) \geq M\lambda_k^*(t)/r$. Then there must exist $B > 0$ such

that we have:

$$\|(1+\frac{M}{r})\Delta\mathbf{I}\| \leq B\|\mathbf{I}_*\| \Rightarrow \|\Delta\mathbf{I}\| \leq \frac{rB}{r+M}\|\mathbf{I}_*\| < \frac{1}{M}rB\|\mathbf{I}_*\| \tag{6.60}$$

Combining equations (6.59) and (6.60), we have

$$\|\mathbf{I}_M^{-1} - \mathbf{I}_*^{-1}\| \leq \frac{1}{M}\underbrace{\frac{2}{\sigma(\mathbf{I}_*)^2}rB\|\mathbf{I}_*\|}_{\text{call it } C} \tag{6.61}$$

meaning that there exists $C > 0$ such that, for any $M > \bar{M}$, $\|\mathbf{I}_M^{-1} - \mathbf{I}_*^{-1}\| \leq C/M$.

Note that the ratio r reflects the effect of $\lambda_k^{\mathrm{q}}(t)$ on the efficiency. In the special case of $q = p^*$, we have $r = 1$ and $\Delta\mathbf{I} = \frac{1}{M+1}\mathbf{I}_*$ and the asymptotic covariance matrix becomes $(1+\frac{1}{M})\mathbf{I}_*^{-1}$.

This completes our proof.

□

6.C Algorithm Details

6.C.1 NCE Objective Computation Details

Our main algorithm is presented as Algorithm 6.1. It covers the recipe for computing our NCE objective, as well as the algorithm to sample from q.

6.C.2 Training the Noise Distribution q by NCE

Before we optimize our $J_{\mathrm{NC}}(\boldsymbol{\theta})$, we first fit the noise distribution q to the training data. As discussed in section 6.3.2, we expect that fitting the data well will give a good training signal to learn $\boldsymbol{\theta}$.

In the experiments of this chapter, we used MLE to estimate the parameters ϕ of q, which involves taking approximate integrals as in the previous chapters. (After

Algorithm 6.1 Training Objective Computation for Noise-Contrastive Estimation.

Input: event sequence $x_{[0,T)}$ with I events at times $0 = t_0 < t_1 < \dots t_I < t_{I+1} = T$; model $p_{\boldsymbol{\theta}}$; noise distribution q; number of noise samples M

Output: training objective J_{NC} evaluated on $x_{[0,T)}$ and the noise samples

procedure COMPUTEOBJECTIVE($x_{[0,T)}, p_{\boldsymbol{\theta}}, q, M$)
  ▷ *algorithm input $p_{\boldsymbol{\theta}}$ gives info to define intensity function $\lambda_k(t)$*
  initialize the states s and s^{q} of $p_{\boldsymbol{\theta}}$ and q respectively ▷ *i.e., their LSTM states*
  $J_{\mathrm{NC}} \leftarrow 0, i \leftarrow 0$
  while $i \leq I$:
    $i \mathrel{+}= 1$
    ▷ *use noise samples in the current interval*
    for $(t, k, \lambda^{\mathrm{q}}, \mu)$ **in** DRAWNOISESAMPLES(t_{i-1}, t_i) :
      compute the model intensity $\lambda_k(t \mid s)$ under $p_{\boldsymbol{\theta}}$
      $J_{\mathrm{NC}} \mathrel{+}= \mu \log \frac{\lambda^{\mathrm{q}}}{\lambda_k(t|s) + M\lambda^{\mathrm{q}}}$
    if $i > I$: **break**
    ▷ *use the real event at time t_i*
    $t \leftarrow t_i$, $k \leftarrow x_{t_i}$
    compute $\lambda_k(t \mid s)$ and $\lambda_k^{\mathrm{q}}(t \mid s^{\mathrm{q}})$ under $p_{\boldsymbol{\theta}}$ and q respectively
    $J_{\mathrm{NC}} \mathrel{+}= \log \frac{\lambda_k(t|s)}{\lambda_k(t|s) + M\lambda_k^{\mathrm{q}}(t|s^{\mathrm{q}})}$
    update the neural states s and s^{q} of $p_{\boldsymbol{\theta}}$ and q respectively with this real event
  return J_{NC}
procedure DRAWNOISESAMPLES($t_{\text{beg}}, t_{\text{end}}$) ▷ *draw noise samples over (t_{beg}, t_{end})*
  ▷ *has access to q, M; define the total intensity function $\lambda^{\mathrm{q}}(t \mid s^{\mathrm{q}}) \stackrel{def}{=} \sum_{c=1}^{C} \lambda_c^{\mathrm{q}}(t \mid s^{\mathrm{q}})$*
  $\mathcal{Q} \leftarrow$ empty collection ▷ *collection of noise samples*
  $t \leftarrow t_{\text{beg}}$; find any $\overline{\lambda} \geq \sup \{\lambda^{\mathrm{q}}(t \mid s^{\mathrm{q}}) : t \in (t_{\text{beg}}, t_{\text{end}})\}$
  repeat
    draw $\Delta \sim \mathrm{Exp}(M\overline{\lambda})$; $t \mathrel{+}= \Delta$ ▷ *propose a noise time*
    if $t < t_{\text{end}}$:
      $\mu \leftarrow \lambda^{\mathrm{q}}(t \mid s^{\mathrm{q}})/\overline{\lambda}$ ▷ *compute probability to accept the proposed time*
      if $\mu < 0.05$: ▷ *stochastically accept t with prob μ if $\mu < 0.05$*
        $u \sim \mathrm{Unif}(0, 1)$; **if** $u < \mu$: $\mu \leftarrow 1$
      if $\mu \geq 0.05$: ▷ *otherwise fractionally accept t with weight μ*
        draw $c \in \{1, \dots, C\}$ where prob of c is $\propto \lambda_c^{\mathrm{q}}(t \mid s^{\mathrm{q}})$ ▷ *choose coarse type*
        draw $k \in \{1, \dots, K\}$ where prob of k is $q(k \mid c)$ ▷ *choose refinement*
        compute the noise intensity $\lambda_k^{\mathrm{q}}(t \mid s^{\mathrm{q}})$ under q
        add $(t, k, \lambda_k^{\mathrm{q}}(t \mid s^{\mathrm{q}}), \mu)$ to $\mathcal{Q}$
  until $t \geq t_{\text{end}}$
  return $\mathcal{Q}$

all, we did not yet know whether NCE would work well.) To avoid the approximate integrals, however, one could instead estimate ϕ using NCE. When evaluating this NCE objective during training of ϕ, one can take the noise distribution to be $q_{\phi_{\text{old}}}$ where ϕ_{old} is any snapshot of ϕ from a recent iteration of training (even the current iteration). The same ϕ_{old} must be used for both drawing noise events via the thinning algorithm, and for scoring these noise events and their contrasting observed events.

Regardless of whether we use MLE or NCE, it is faster to train q than to train p because q only has C event types instead of K.

The idea of using as the noise distribution a model previously trained with NCE was also considered in the original NCE paper (Gutmann and Hyvärinen, 2010).

6.D Experimental Details and Additional Results

6.D.1 Dataset Details

Besides the datasets we have introduced in section 6.5, we also run experiments on the following real-world social interaction datasets:

CollegeMsg (Panzarasa, Opsahl, and Carley, 2009)**.** This dataset contains anonymized private messages sent on an online social network at an university. Each record (u, v, t) means that user u sent a private message to user v at time t and each u, v pair is an event type. We consider the top 100 users sorted by the number of messages they sent and received: the total number of possible event types is then $K = 9900$ since self-messaging is not allowed.

WikiTalk (Leskovec, Huttenlocher, and Kleinberg, 2010)**.** This dataset contains the records of anonymized Wikipedia users editing each other's Talk page. Each record (u, v, t) means that user u edited user v's talk page at time t and each u, v pair is an

Dataset	K	# of Event Tokens			Sequence Length		
		Train	Dev	Test	Min	Mean	Max
Synthetic-1	10000	100000	10000	10000	100	100	100
Synthetic-2	10000	100000	10000	10000	100	100	100
EuroEmail	10000	50000	10000	10000	100	100	100
BitcoinOTC	19800	1000	500	500	100	100	100
CollegeMsg	9900	8000	1000	1000	100	100	100
WikiTalk	10000	100000	20000	20000	100	100	100
RoboCup	528	2195	817	780	780	948	1336
IPTV	49000	27355	4409	4838	36602	36602	36602

Table 6.1: Statistics of each dataset. For IPTV, we have a single long sequence of 36602 tokens: we use the first 27355 as training data, the next 4409 as dev data and the remaining 4838 as test data. For other datasets, training, dev and test sequences are separate sequences.

event type. We consider the top 100 users sorted by the number of edits they made and received and the total number of possible event types is $K = 10000$.

Table 6.1 shows statistics about each dataset that we use in this chapter.

6.D.2 Training Details

For each of the chosen models in section 6.5, the only hyperparameter to tune is the hidden dimension D of the neural network. On each dataset, we searched for D that achieves the best performance on the dev set. Our search space is $\{4, 8, 16, 32, 64, 128\}$.

For learning, we used the Adam algorithm (Kingma and Ba, 2015) with its default settings. For each ρ or M, we run training long enough so that the log-likelihood on the held-out data can converge.

6.D.3 More Results on Real-World Social Interaction Datasets

The learning curves on CollegeMsg and WikiTalk datasets are shown in Figure 6.5: they look similar to those in Figure 6.2 and lead to the same conclusions.

6.D.4 Ablation Study I: Always or Never Redraw Noise samples

In Figure 6.6, we show the learning curves for the "always redraw" and "never redraw" strategies on the first synthetic dataset. As shown in Figure 6.6a, with the "always redraw" strategy, NCE (—) needs considerably fewer intensity evaluations to reach the highest log-likelihood (- - -) that MLE (—) can achieve on the held-out data. However, the curve with $M = 1000$ increases more slowly than MLE in terms of wall-clock time since it spends too much time on drawing new noise samples.

As shown in Figure 6.6b, with the "never redraw" strategy, $M = 1000$ overtakes MLE: a single draw of $M = 1000$ noise sequences is able to give very good training signals and the saved computation can be spent on training $p_{\boldsymbol{\theta}}$ repeatedly on the same samples. However, the curve of $M = 1$ only achieves log-likelihood ≈ -200 and thus falls out of the zoomed-in view.

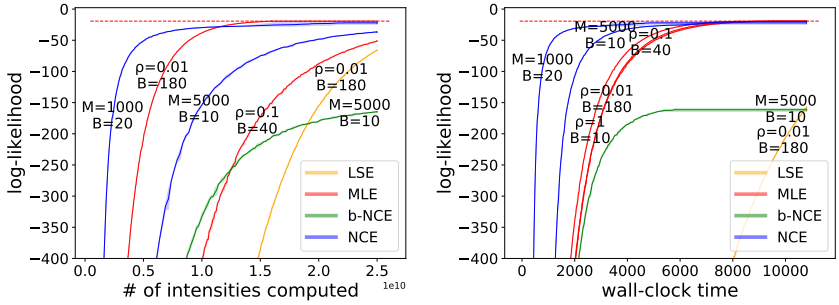
6.D.5 Ablation Study II: NCE with Untrained Noise Distribution

In Figure 6.7, we show the learning curves of NCE with untrained noise distributions on the real-world social interaction datasets. As we can see, NCE in this setting tends to end up with worse generalization (interestingly except on WikiTalk) and suffers slow convergence (on BitcoinOTC and CollegeMsg) and large variance (on BitcoinOTC).

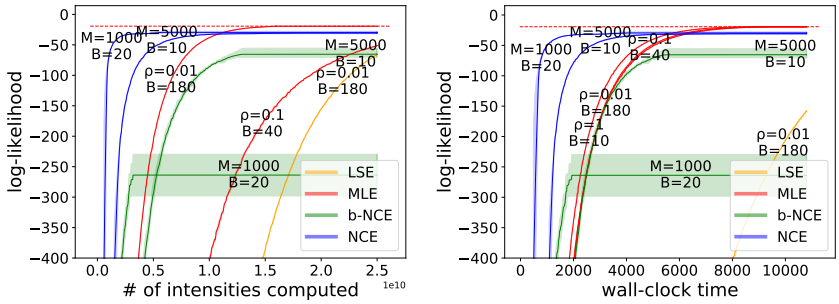
6.D.6 Ablation Study III: Effect of C

In Figure 6.8, we show learning curves of NCE using the neural q with $C = 1$. Taking $C = 1$ means that the same number of noise samples can be drawn faster (with fewer intensity evaluations). However, more training epochs may be needed because the noise looks less like true observations and so NCE's discrimination tasks are less challenging (see section 6.3.2).

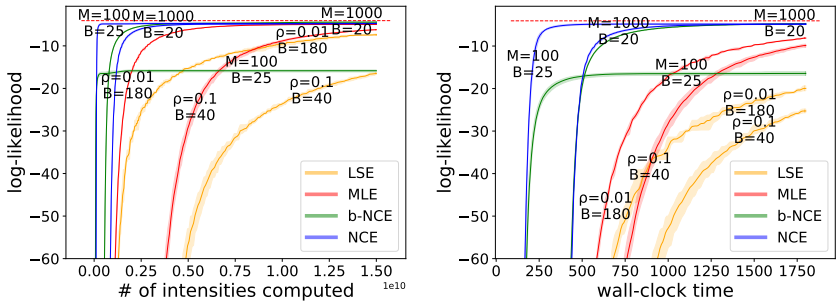
On the RoboCup dataset, $C = 1$ exhibits similar learning speed to $C = 5$ but has slightly worse generalization. On the IPTV dataset, $C = 1$ gives a considerable speedup over $C = 49$ without harming the final generalization. The NCE curves for $M = 5$ and $M = 10$ shift substantially to the left, since $C = 1$ requires *many* fewer intensity evaluations.



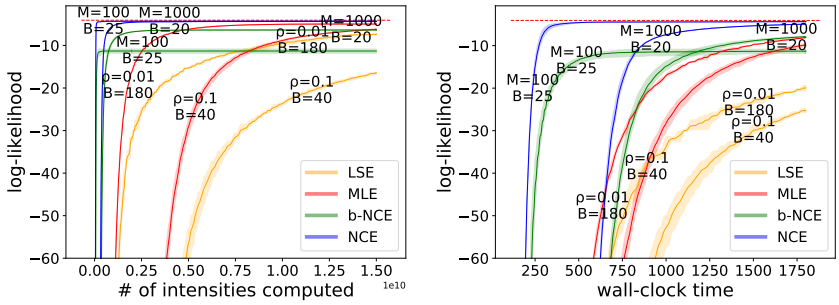
(a1) CollegeMsg: neural q



(a2) CollegeMsg: Poisson q

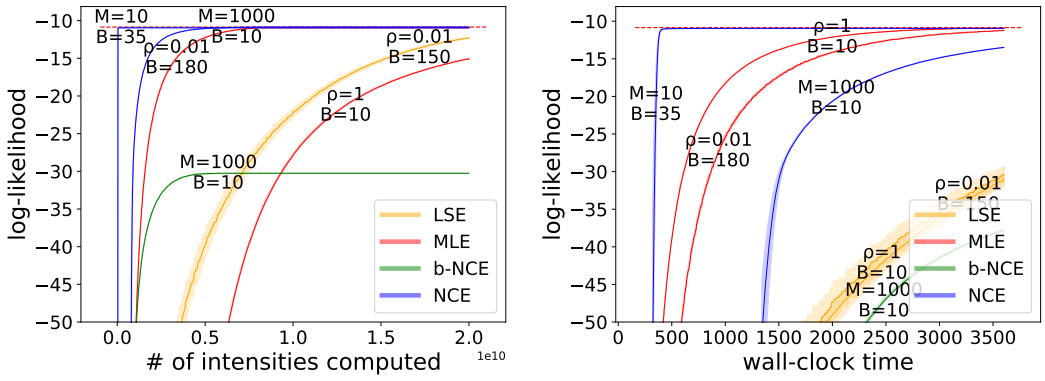


(b1) WikiTalk: neural q

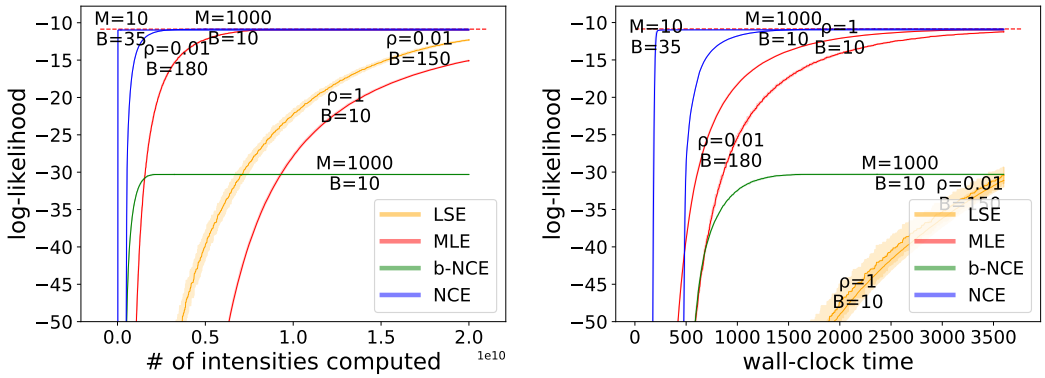


(b2) WikiTalk: Poisson q

Figure 6.5: Learning curves of MLE and NCE on the other real-world social interaction datasets.

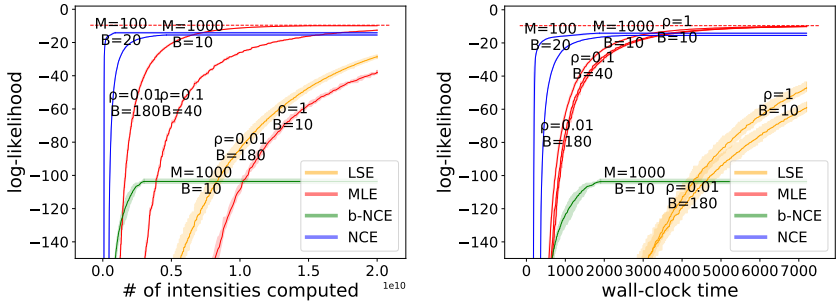


(a) Always redraw new noise samples

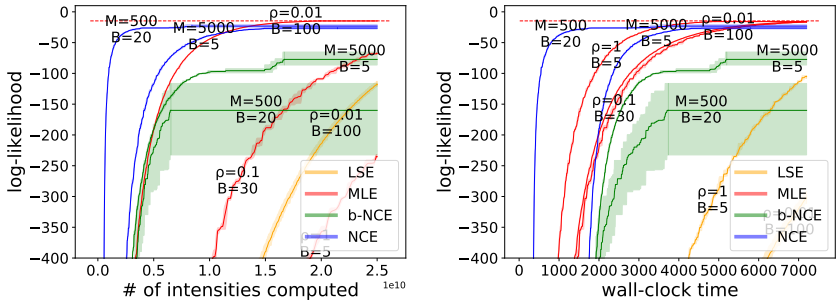


(b) Never redraw new noise samples

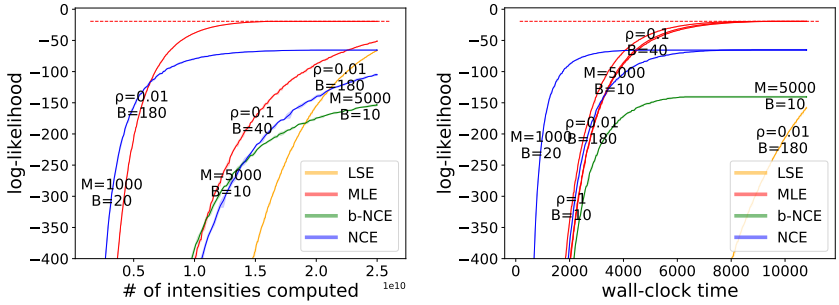
Figure 6.6: Ablation study I. Learning curves of MLE and NCE with $q = p^*$ and different "redraw" strategies.



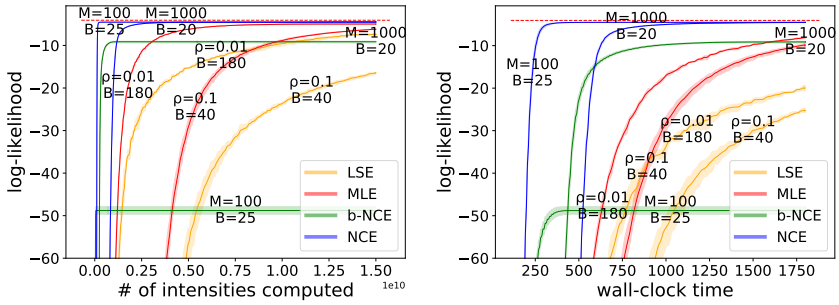
(a) EuroEmail



(b) BitcoinOTC

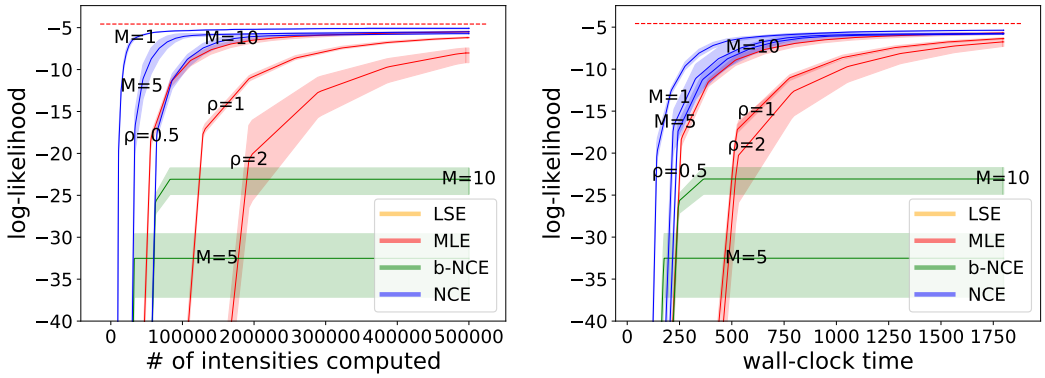


(c) CollegeMsg

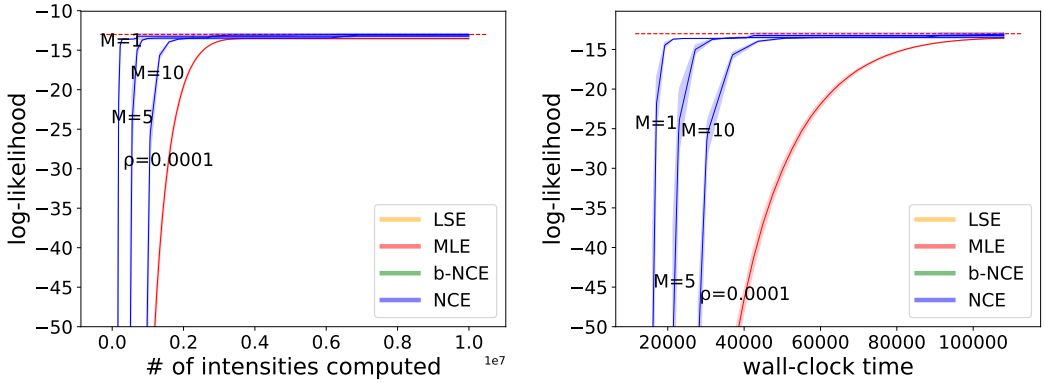


(d) WikiTalk

Figure 6.7: Ablation study II. Learning curves of MLE and NCE with untrained q on social interaction datasets.



(a) RoboCup



(b) IPTV

Figure 6.8: Ablation study III. Learning curves of MLE and NCE using neural q with $C = 1$.

Chapter 7

Efficient Imputation of Missing Events: Particle Smoothing

In Chapter 3 and Chapter 4, we focused on the task of predicting future events given a *complete* sequence. In this chapter, we consider a new scenario where the given sequence is *incomplete* and we would like to impute the *missing* events. Given a probability model of complete sequences, we propose particle smoothing—a form of sequential importance sampling—for this task. We develop a trainable family of proposal distributions based on a type of bidirectional continuous-time LSTM: bidirectionality lets the proposals condition on future observations, not just on the past as in particle filtering. Our method can sample an ensemble of possible complete sequences (particles), from which we form a single consensus prediction that has low Bayes risk under our chosen loss metric. We experiment in multiple synthetic and real domains, using different missingness mechanisms, and modeling the complete sequences in each domain with a neural Hawkes process. On held-out incomplete sequences, our method is effective at inferring the ground-truth unobserved events, with particle smoothing consistently improving upon particle filtering.

7.1 Motivation

Event sequences are often *partially* observed. We would like to impute the missing events $\mathbf{z}$. Suppose we know the prior distribution p_{model} of complete event sequences, as well as the "missingness mechanism" $p_{\text{miss}}(\mathbf{z} \mid \text{complete sequence})$, which stochastically determines which of the events will not be observed. One can then use use Bayes' Theorem, as spelled out in equation (7.1) below, to define the posterior distribution $p(\mathbf{z} \mid \mathbf{x})$ given just the observed events $\mathbf{x}$.[1]

7.1.1 Why is this important?

The ability to impute $\mathbf{z}$ is useful in many applied domains, for example:

Medical records. Some patients record detailed symptoms, self-administered medications, diet, and sleep. Imputing these events for other patients would produce an augmented medical record that could improve diagnosis, prognosis, treatment, and counseling.

Similar remarks apply to users of life-tracking apps (e.g., MyFitnessPal) who forget to log some of their daily activities (e.g., meals, sleep and exercise).

Competitive games. In poker or StarCraft, a player lacks full information about what her opponents have acquired (cards) or done (build mines and train soldiers). Accurately imputing hidden actions from "what I did" and "what I observed others doing" can help the player make good decisions. Similar remarks apply to practical scenarios (e.g., military) where multiple actors compete and/or cooperate.

User interface interactions. Cognitive events are usually unobserved. For example,

[1] Bayes' Theorem can be applied even if p_{miss} is a missing-not-at-random (MNAR) mechanism, as is common in this setting. MNAR is only tricky if we know *neither* p_{model} *nor* p_{miss}.

users of an online news provider (e.g., Bloomberg Terminal) may have read and remembered a displayed headline whether or not they clicked on it. Such events are expensive to observe (e.g., via gaze tracking or asking the user). Imputing them given the observed events (e.g., other clicks) would facilitate personalization.

Other partially observed event sequences arise in *online shopping*, *social media*, etc.

7.1.2 Why is it challenging?

It is computationally difficult to reason about the posterior distribution $p(\mathbf{z} \mid \mathbf{x})$. Even for a simple p_{model} like a Hawkes process (Hawkes, 1971), Markov chain Monte Carlo (MCMC) methods are needed, and these methods obtain an efficient transition kernel only by exploiting special properties of the process (Shelton, Qin, and Shetty, 2018). Unfortunately, such properties no longer hold for the more flexible neural models including our neural Hawkes process.

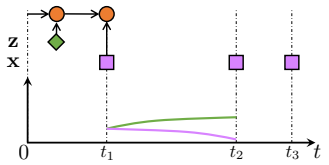
7.2 An Overview of Our Approach

We are, to the best of our knowledge, the first to develop general sequential Monte Carlo (SMC) methods to approximate the posterior distribution over incompletely observed draws from a neural point process. We begin by sketching the approach.

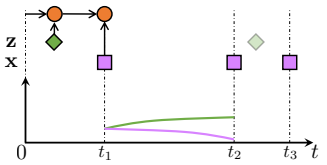
7.2.1 A Naive but Inefficient Method

Section 2.3 gives an algorithm to sample a complete sequence from a point process. Each event in turn is sampled given the complete history of previous events. However, this algorithm only samples from the prior over complete sequences. We first adapt it into a **particle filtering** algorithm that samples from the posterior given all the observed events. The basic idea (Figure 7.1a) is to draw the events in sequence

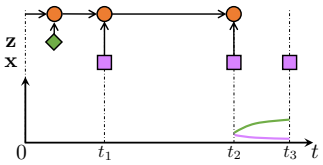
Figure 7.1: Stochastically imputing a taxi's pick-up events (◆) given its observed drop-off events (■). At this stage, we are trying to determine the next event after the ■ at time t_1—either an unobserved event at $t_{1,1} \in (t_1, t_2)$ or the next observed event at t_2.



(a1) Both intensities are low (i.e., passengers are scarce at this time of day), so no event happens to be proposed in (t_1, t_2).

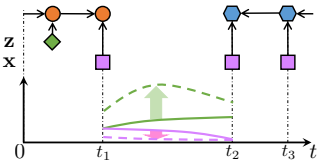


(a2) Specifically, the next proposed event (◆) would be *somewhere* after t_2, without bothering to determine its time precisely.

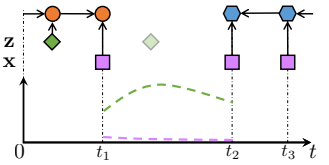


(a3) Thus, the next event is ■@t_2; we feed it into the LSTM, preempting ◆, which is discarded (line 46 of Algorithm 7.1).

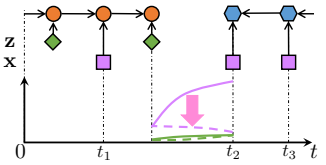
(a) Particle filtering (section 7.4.1). We show part of the process of drawing one particle. Above left, the neural Hawkes process's LSTM has already read the proposed and observed events at times $\leq t_1$. Its resulting state ● determines the model intensities ▬ and ▬ of the two event types ◆ and ■, from which the sampler (Algorithm 7.1 in section 7.C) determines that there is no unobserved event in (t_1, t_2). Above right, we continue to extend the particle by feeding ■@t_2 into the LSTM and proposing subsequent events based on the new intensities after t_2. But because ▬ was low at t_2, the ■@t_2 was unexpected, and that results in downweighting the particle (line 57 of Algorithm 7.1). Downweighting recognizes belatedly that proposing no event in (t_1, t_2) has committed us to a particle that will be improbable under the posterior, because its complete sequence includes consecutive drop-offs (■@t_1, ■@t_2) far apart in time.



(b1) Since a drop-off at t_2 strongly suggests a pick-up before t_2, considering the future increases the intensity of pick-up on (t_1, t_2) from ▬ to ▬ (while decreasing that of drop-off from ▬ to ▬).



(b2) Consequently, the next proposed event is more likely to be a pick-up in (t_1, t_2) than it was in Figure 7.1a. If we stochastically generate such an event ◆@$t_{1,1}$, it is fed into the original ● LSTM.



(b3) The updated state ● determines the new *model* intensities ▬ and ▬, and also combines with ⬢ to determine the new *proposal* intensities ▬ and ▬, which are used to sample the next event.

(b) Particle smoothing (section 7.4.2) samples from a better-informed proposal distribution: a second LSTM (section 7.D) reads the future observations from right to left, and its state ⬢ is used *together* with ● to determine the proposal intensities ▬ and ▬.

as before, but now we *force* any observed events to be "drawn" at the appropriate times. That is, we add the observed events to the sequence as they happen (and they duly affect the distribution of subsequent events). There is an associated cost: if we

are forced to draw an observed event that is *improbable* given its past history, we must downweight the resulting complete sequence accordingly, because evidently the particular past history that we sampled was inconsistent with the observed event, and hence cannot be part of a high-likelihood complete sequence. Using this method, we sample many sequences (or **particles**) of different *relative* weights. This method applies to any temporal point process.[2] Linderman, Wang, and Blei (2017) apply it to the classical Hawkes process.

Alas, this approach is computationally inefficient. Sampling a complete sequence that is actually probable under the posterior requires great luck, as the proposal distribution must have the good fortune to draw only events that happen to be consistent with future observations. Such lucky particles would appropriately get a high weight relative to other particles. The problem is that we will rarely get such particles at all (unless we sample very many).

7.2.2 A Smart and Efficient Method

To get a more accurate picture of the posterior, we draw each event from a smarter distribution that is conditioned on the future observations (rather than drawing the event in ignorance of the future and then downweighting the particle if the future does not turn out as hoped).

This idea is called **particle smoothing** (Doucet and Johansen, 2009). How does it work in our setting? The neural Hawkes process defines the distribution of the next event using the state of a continuous-time LSTM that has read the past history from left to right. When sampling a proposed event, we now use a modified distribution

[2] As long as the number of events is finite with probability 1, and it is tractable to compute the log-likelihood of a complete sequence and to estimate the log-likelihoods of its prefixes.

(Figure 7.1b) that *also* considers the state of a second continuous-time LSTM that has read the future observations from right to left. As this modified distribution is still imperfect—merely a proposal distribution—we still have to reweight our particles to match the actual posterior under the model. But this reweighting is not as drastic as for particle filtering, because the new proposal distribution was constructed and trained to resemble the actual posterior. Our proposal distribution could also be used with other point process models by replacing the left-to-right LSTM state with other informative statistics of the past history.

What other contributions? We introduce an appropriate evaluation loss metric for event sequence reconstruction, and then design a **consensus decoder** that outputs a single low-risk prediction of the missing events by combining the sampled particles (instead of picking one of them).

7.3 Problem Formulation

7.3.1 Partially Observed Event Sequences

We are given a fixed time interval $[0, T)$. In the previous chapters, we were assumed to observe the *entire* event sequence $x_{[0,T)}$ over the interval. But in this chapter, we consider a missing-data setting (Little and Rubin, 1987): each event of type $k \in \{1, \ldots, K\}$ is designated as either "observed" or "missing", and we observe only the "observed" events.

For presentation simplicity, throughout this chapter, we denote each event mnemonically as $k@t$. We denote the **observed events** by $\mathbf{x} = \{k_1@t_1, k_2@t_2, \ldots, k_I@t_I\}$, where $0 = t_0 < t_1 < t_2 < \ldots < t_I < t_{I+1} = T$. We are given the observation interval $[0, T)$ in the form of two **boundary events** $k_0@t_0$ and $k_{I+1}@t_{I+1}$ at its endpoints,

where $k_0 = 0$ and $k_{I+1} = K + 1$.

Let $k_{i,0}@t_{i,0}$ be an alternative notation for the observed event $k_i@t_i$. Following this observed event (for any $0 \leq i \leq I$), there are $J_i \geq 0$ **unobserved events** $\mathbf{z} = \{k_{i,1}@t_{i,1}, k_{i,2}@t_{i,2}, \ldots, k_{i,J_i}@t_{i,J_i}\}$, where $t_{i,0} < t_{i,1} < \ldots < t_{i,J_i} < t_{i+1}$. We must guess this unobserved sequence including its length J_i. Let $\sqcup$ denote disjoint union. Our hypothesized **complete event sequence**[3] $\mathbf{x} \sqcup \mathbf{z}$ is thus $\{k_{i,j}@t_{i,j} : 0 \leq i \leq I+1, 0 \leq j \leq J_i\}$, where $t_{i,j}$ increases strictly with the pair $\langle i, j \rangle$ in lexicographic order.[4]

In this chapter, we will attempt to guess all of $\mathbf{z}$ jointly by sampling it from the posterior distribution

$$p(Z = \mathbf{z} \mid X = \mathbf{x}) \propto p_{\text{model}}(Y = \mathbf{x} \sqcup \mathbf{z}) \cdot p_{\text{miss}}(Z = \mathbf{z} \mid Y = \mathbf{x} \sqcup \mathbf{z})$$

of a process that *first* generates the complete sequence $\mathbf{x} \sqcup \mathbf{z}$ from a complete data model p_{model} (given $[0, T)$), and *then* determines which events to censor with the possibly stochastic **missingness mechanism** p_{miss}. The random variables X, Z, and Y refer respectively to the sets of observed events, missing events, and all events over $[0, T)$. Thus $Y = X \sqcup Z$. Under the distributions we will consider, $|Y|$ is almost surely finite. Notice that $\mathbf{z}$ denotes the set of missing events in Y and $Z = \mathbf{z}$ denotes the fact that they are missing. That said, we will abbreviate our notation above in the standard way:

$$p(\mathbf{z} \mid \mathbf{x}) \;\propto\; p_{\text{model}}(\mathbf{x} \sqcup \mathbf{z}) \cdot p_{\text{miss}}(\mathbf{z} \mid \mathbf{x} \sqcup \mathbf{z}) \tag{7.1}$$

[3] Denoted as $x_{[0,T)}$ in the previous chapters.

[4] In general we should allow $t_{i,j}$ to increase *non*-strictly with $\langle i, j \rangle$. But equality happens to have probability 0 under the neural Hawkes model. So it is convenient to exclude it here, simplifying notation by allowing $\mathbf{x}, \mathbf{z}, \mathcal{H}(t)$ to be sets, not sequences.

Note that $\mathbf{x} \sqcup \mathbf{z}$ is simply an undifferentiated sequence of $k@t$ pairs; the subscripts $\langle i, j \rangle$ are in effect assigned by p_{miss}, which partitions $\mathbf{x} \sqcup \mathbf{z}$ into $\mathbf{x}$ and $\mathbf{z}$. To explain a sequence of 50 observed events, one hypothesis is that p_{model} generated 73 events and then p_{miss} selected 23 of them to be missing (as $\mathbf{z}$), leaving the 50 observed events (as $\mathbf{x}$).

In many missing data settings, the second factor of equation (7.1) can be ignored because (for the given $\mathbf{x}$) it is known to be a constant function of $\mathbf{z}$. Then the missing data are said to be **missing at random (MAR)**. For event sequences, however, the second factor is generally not constant in $\mathbf{z}$ but varies with the *number* of missing events $|\mathbf{z}|$. Thus, our unobserved events are normally **missing not at random (MNAR)**. See discussion in section 7.6.1 and section 7.A.

7.3.2 Choice of p_{model}

We take $p_{\text{model}}(\mathbf{x} \sqcup \mathbf{z})$ to be our neural Hawkes process (Chapter 3). Whether an event happens at time $t \in [0, T)$ depends on the **history** $\mathcal{H}(t) \stackrel{\text{def}}{=} \{k'@t' \in \mathbf{x} \sqcup \mathbf{z} : t' < t\}$—the set of all *observed* and *unobserved* events before t. Given this history, the neural Hawkes process defines an intensity $\lambda_k(t \mid \mathcal{H}(t)) \in \mathbb{R}_{\geq 0}$, which may be thought of as the instantaneous rate at time t of events of type k:

$$\lambda_k(t \mid \mathcal{H}(t)) = f_k(\mathbf{v}_k^\top \mathbf{h}(t)) \tag{7.2}$$

The vector $\mathbf{h}(t) \in (-1, 1)^D$ summarizes $(\mathcal{H}(t), t)$. It is the hidden state at time t of a continuous-time LSTM that previously read the events in $\mathcal{H}(t)$ *as they happened*. The state of such an LSTM evolves endogenously as it waits between events, so the state $\mathbf{h}(t)$ reflects not only the sequence of past events but also their *timing*, including the gap between the last event in $\mathcal{H}(t)$ and t.

Remark. In the previous chapters, we write the intensities as $\lambda_k(t \mid x_{[0,t)})$, where the sequence $x_{[0,t)}$ over $[0, t)$ includes all the events and non-events before time t. In this chapter, we choose to define the new notation $\mathcal{H}$ to facilitate the presentation involving the *previous* events that *actually happened*. Shortly, a similar notation $\mathcal{F}$ will be defined in section 7.4.2, to facilitate the presentation involving the *future* events that are *observed*. Previous notations like $x_{[0,t)}$ and $x_{(t,T)}$ are not sufficient to facilitate such presentation since they also include *non-events* and *missing* events.

7.4 Particle Methods

It is often intractable to sample *exactly* from $p(\mathbf{z} \mid \mathbf{x})$, because $\mathbf{x}$ and $\mathbf{z}$ can be interleaved with each other. As an alternative, we can use normalized importance sampling, drawing many $\mathbf{z}$ values from a **proposal distribution** $q(\mathbf{z} \mid \mathbf{x})$ and weighting them in proportion to $\frac{p(\mathbf{z}|\mathbf{x})}{q(\mathbf{z}|\mathbf{x})}$. Figure 7.1 shows the key ideas in terms of an example. Full details are spelled out in Algorithm 7.1 in section 7.C.

Algorithm 7.1 is a **Sequential Monte Carlo (SMC)** approach (Moral, 1997; Liu and Chen, 1998; Doucet, Godsill, and Andrieu, 2000; Doucet and Johansen, 2009). It returns an **ensemble of weighted particles** $\mathcal{Z}_M = \{(\mathbf{z}_m, w_m)\}_{m=1}^M$. Each particle $\mathbf{z}_m$ is sampled from the **proposal distribution** $q(\mathbf{z} \mid \mathbf{x})$, which is defined to support sampling via a *sequential* procedure that draws one unobserved event at a time. The corresponding w_m are **importance weights**, which are defined as follows (and built up factor-by-factor in Algorithm 7.1):

$$w_m \propto \frac{p_{\text{model}}(\mathbf{x} \sqcup \mathbf{z}_m)\; p_{\text{miss}}(\mathbf{z}_m \mid \mathbf{x} \sqcup \mathbf{z}_m)}{q(\mathbf{z}_m \mid \mathbf{x})} \geq 0 \tag{7.3}$$

where the normalizing constant is chosen to make $\sum_{m=1}^M w_m = 1$. Equations (7.1) and (7.3) imply that we would have $w_m = 1/M$ if we could set $q(\mathbf{z} \mid \mathbf{x})$ equal to

$p(\mathbf{z} \mid \mathbf{x})$, so that the particles were IID samples from the desired posterior distribution. In practice, q will not equal p, but will be easier than p to sample from. To correct for the mismatch, the importance weights w_m are higher for particles that q proposes less often than p would have proposed them.

The distribution implicitly formed by the ensemble, $\hat{p}(\mathbf{z})$, approaches $p(\mathbf{z} \mid \mathbf{x})$ as $M \to \infty$ (Doucet and Johansen, 2009). Thus, for large M, the ensemble may be used to estimate the expectation of *any* function $f(\mathbf{z})$, via

$$\mathbb{E}_{p(\mathbf{z}|\mathbf{x})}[f(\mathbf{z})] \approx \mathbb{E}_{\hat{p}}[f(\mathbf{z})] = \textstyle\sum_{m=1}^{M} w_m f(\mathbf{z}_m) \tag{7.4}$$

$f(\mathbf{z})$ may be a function that summarizes properties of the complete sequence $\mathbf{x} \sqcup \mathbf{z}$ on $[0, T)$, or predicts *future* events on $[T, \infty)$ using the sufficient statistic $\mathcal{H}(T) = \mathbf{x} \sqcup \mathbf{z}$.

In the subsections below, we will describe two specific proposal distributions q that are appropriate for the neural Hawkes process, as we sketched in section 7.1. These distributions define intensity functions λ^q over time intervals.

The trickiest part of Algorithm 7.1 (at line 38) is to sample the next unobserved event from the proposal distribution q. Here we use the thinning algorithm (section 2.3). Briefly, this is a rejection sampling algorithm whose own proposal distribution uses a *constant* intensity λ^*, making it a homogeneous Poisson process (which is easy to sample from). A event proposed by the Poisson process at time t is accepted with probability $\lambda^q(t)/\lambda^* \leq 1$. If it is rejected, we move on to the next event proposed by the Poisson process, continuing until we either accept such an unobserved event or are preempted by the arrival of the next observed event.

After each step, one may optionally *resample* a new set of particles from $\{\mathbf{z}_m\}_{m=1}^M$ (the RESAMPLE procedure in Algorithm 7.1). This trick tends to discard low-weight

particles and clone high-weight particles, so that the algorithm can explore multiple continuations of the high-weight particles.

7.4.1 Particle Filtering

We already have a neural Hawkes process p_{model} that was trained on complete data. This model uses a neural net to define an intensity function $\lambda_k^p(t \mid \mathcal{H}(t))$ for *any* history $\mathcal{H}(t)$ of events before t and each event type k.

The simplest proposal distribution uses this intensity function to draw the unobserved events. More precisely, for each $i = 0, 1, \ldots, I$, for each $j = 0, 1, 2, \ldots$, let the next event $k_{i,j+1}@t_{i,j+1}$ be the first event generated by any of the K intensity functions $\lambda_k(t \mid \mathcal{H}(t))$ over the interval $t \in (t_{i,j}, t_{i+1})$, where $\mathcal{H}(t)$ consists of all observed and unobserved events up through $k_{i,j}@t_{i,j}$. If no event is generated on this interval, then the next event is $k_{i+1}@t_{i+1}$. This is implemented by Algorithm 7.1 with $smooth =$ **false**.

7.4.2 Particle Smoothing

As motivated in section 7.1, we would rather draw each unobserved event according to $\lambda_k(t \mid \mathcal{H}(t), \mathcal{F}(t))$ where the **future** $\mathcal{F}(t) \stackrel{\text{def}}{=} \{k_i@t_i : t < t_i \leq T\}$ consists of all *observed* events that happen after t. Note the asymmetry with $\mathcal{H}(t)$, which includes observed but also unobserved events.

We use a **right-to-left continuous-time LSTM** to summarize the future $\mathcal{F}(t)$ for any time t into another hidden state vector $\bar{\mathbf{h}}(t) \in (-1, 1)^{D'}$. Then we parameterize the proposal intensity using an extended variant of equation (7.2):

$$\lambda_k^q(t \mid \mathcal{H}(t), \mathcal{F}(t)) = f_k(\mathbf{v}_k^\top(\mathbf{h}(t) + \mathbf{B}\bar{\mathbf{h}}(t))) \tag{7.5}$$

This extra machinery is used by Algorithm 7.1 when $smooth = \textbf{true}$. Intuitively, the left-to-right $\mathbf{h}(t)$, as explained in Chapter 3, reads the history $\mathcal{H}(t)$ and computes sufficient statistics for predicting events at times $\geq t$ given $\mathcal{H}(t)$. But we wish to predict these events given $\mathcal{H}(t)$ *and* $\mathcal{F}(t)$. Equation (7.5) approximates this Bayesian update using the right-to-left $\bar{\mathbf{h}}(t)$, which is trained to carry back relevant information about future observations $\mathcal{F}(t)$.

This is a kind of neuralized forward-backward algorithm. Lin and Eisner (2018) treat the discrete-time analogue, explaining why a neural forward p_{model} no longer admits tractable exact proposals as does a hidden Markov model (Rabiner, 1989) or linear dynamical system (Rauch, Striebel, and Tung, 1965). Like them, we fall back on training an approximate proposal distribution. Regardless of p_{model}, particle smoothing is to particle filtering as Kalman smoothing is to Kalman filtering (Kalman, 1960; Kalman and Bucy, 1961).

Our right-to-left LSTM has the same architecture as the left-to-right LSTM used in our p_{model} (section 7.3.2), but a separate parameter vector. For any time $t \in [0, T)$, it arrives at $\bar{\mathbf{h}}(t)$ by reading *only* the *observed* events $\{k_i @ t_i : t < t_i \leq T\}$, i.e., $\mathcal{F}(t)$, in *reverse* chronological order. Formulas are given in section 7.D. This architecture seemed promising for reading an *incomplete* sequence of events from right to left, as section 3.3.4 of Chapter 3 had already shown that this architecture is predictive when used to read incomplete sequences from left to right.

7.4.3 Training the Particle Smoother

The particle smoothing proposer q can be trained to approximate $p(\mathbf{z} \mid \mathbf{x})$ by minimizing a **Kullback-Leibler (KL) divergence**. Its left-to-right LSTM is fixed at p_{model}, so its trainable parameters ϕ are just the parameters of the right-to-left LSTM together

with the matrix $\mathbf{B}$ from equation (7.5). Though $p(\mathbf{z} \mid \mathbf{x})$ is unknown, the gradient of **inclusive KL divergence** between $q(\mathbf{z} \mid \mathbf{x})$ and $p(\mathbf{z} \mid \mathbf{x})$ is

$$\nabla_\phi \mathrm{KL}(p \mid\mid q) = \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z}|\mathbf{x})}[-\nabla_\phi \log q(\mathbf{z} \mid \mathbf{x})] \tag{7.6}$$

and the gradient of **exclusive KL divergence** is:

$$\nabla_\phi \mathrm{KL}(q \mid\mid p) = \mathbb{E}_{\mathbf{z} \sim q}[\nabla_\phi \left(\tfrac{1}{2} \left(\log q(\mathbf{z} \mid \mathbf{x}) - b\right)^2\right)] \tag{7.7a}$$

$$b = \log p_{\text{model}}(\mathbf{x} \sqcup \mathbf{z}) + \log p_{\text{miss}}(\mathbf{z} \mid \mathbf{x} \sqcup \mathbf{z}) \tag{7.7b}$$

where $\log p_{\text{model}}(\mathbf{x} \sqcup \mathbf{z})$ is given in **??**, $\log q(\mathbf{z} \mid \mathbf{x})$ is given in section 7.C.1, and $p_{\text{miss}}(\mathbf{z} \mid \mathbf{x} \sqcup \mathbf{z})$ is assumed to be known to us for any given pair of $\mathbf{x}$ and $\mathbf{z}$.

Minimizing inclusive KL divergence aims at high recall—$q(\mathbf{z} \mid \mathbf{x})$ is adjusted to assign high probabilities to all of the good hypotheses (according to $p(\mathbf{z} \mid \mathbf{x})$). Conversely, minimizing exclusive KL divergence aims at high precision—$q(\mathbf{z} \mid \mathbf{x})$ is adjusted to assign low probabilities to poor reconstructions, so that they will not be proposed. We seek to minimize the linearly combined divergence

$$\text{Div} = \beta\, \mathrm{KL}(p \| q) + (1 - \beta)\mathrm{KL}(q \| p) \text{ with } \beta \in [0, 1] \tag{7.8}$$

and training is early-stopped when the divergence stops decreasing on the held-out development set.

But how do we measure these divergences between $q(\mathbf{z} \mid \mathbf{x})$ and $p(\mathbf{z} \mid \mathbf{x})$? Of course, we actually want the *expected* divergence when the observed sequence $\mathbf{x} \sim$ the true distribution. Thus, we sample $\mathbf{x}$ by starting with a *fully observed* sequence from our training examples and then sampling a partition $\mathbf{x}, \mathbf{z}$ from the

known missingness mechanism p_{miss}.[5] The inclusive expectation in (7.6) uses this $\mathbf{x}$ and $\mathbf{z}$. For the exclusive expectation in (7.7), we keep this $\mathbf{x}$ but sample a new $\mathbf{z}$ from our proposal distribution $q(\cdot \mid \mathbf{x})$.

Notice that minimizing exclusive divergence here is essentially the REINFORCE algorithm (Williams, 1992), which is known to have large variance. In practice, when tuning our hyperparameters (section 7.G.2), $\beta = 1$ in (7.8) gave the best results. That is—perhaps unsurprisingly—our experiments effectively avoided REINFORCE altogether and placed *all* the weight on the inclusive KL, which has no variance issue. More training details including a bias and variance discussion can be found in section 7.G.2.

Section 7.H discusses situations where training on incomplete data by EM is possible.

7.5 A Loss Function and Decoding Method

It is often useful to find a *single* hypothesis $\hat{\mathbf{z}}$ that minimizes the *Bayes risk*, i.e., the expected loss with respect to the *unknown* ground truth $\mathbf{z}^*$. This procedure is called **minimum Bayes risk (MBR) decoding** and can be approximated with our ensemble of weighted particles:

$$\hat{\mathbf{z}} = \arg\min_{\mathbf{z}\in\mathcal{Z}} \sum_{\mathbf{z}^*\in\mathcal{Z}} p(\mathbf{z}^* \mid \mathbf{x}) L(\mathbf{z}, \mathbf{z}^*) \tag{7.9a}$$

$$\approx \arg\min_{\mathbf{z}\in\mathcal{Z}} \sum_{m=1}^{M} w_m L(\mathbf{z}, \mathbf{z}_m) \tag{7.9b}$$

[5] To get more data for training q, we could sample more partitions of the fully observed sequence. In this chapter, we only sample one partition. Note that the fully observed sequence is a real observation from the true complete data distribution (not the model).

where $L(\mathbf{z}, \mathbf{z}^*)$ is the **loss** of $\mathbf{z}$ with respect to $\mathbf{z}^*$. This procedure for combining the particles into a single prediction is sometimes called **consensus decoding**. We now propose a specific loss function L and an approximate decoder.

7.5.1 Optimal Transport Distance

The loss of $\mathbf{z}$ is defined as the minimum cost of editing $\mathbf{z}$ into the ground truth $\mathbf{z}^*$. To accomplish this edit, we must identify the best **alignment**—a one-to-one partial matching $\mathbf{a}$—of the events in the two sequences. We require any two aligned events to have the same type k. We define $\mathbf{a}$ as a collection of alignment edges (t, t^*) where t and t^* are the times of the aligned events in $\mathbf{z}$ and $\mathbf{z}^*$ respectively. An alignment edge between a predicted event at time t (in $\mathbf{z}$) and a true event at time t^* (in $\mathbf{z}^*$) incurs a cost of $|t - t^*|$ to move the former to the correct time. Each unaligned event in $\mathbf{z}$ incurs a deletion cost of C_{delete}, and each unaligned event in $\mathbf{z}^*$ incurs an insertion cost of C_{insert}. Now

$$L(\mathbf{z}, \mathbf{z}^*) = \min_{\mathbf{a} \in \mathcal{A}(\mathbf{z}, \mathbf{z}^*)} D(\mathbf{z}, \mathbf{z}^*, \mathbf{a}) \tag{7.10}$$

where $\mathcal{A}(\mathbf{z}, \mathbf{z}^*)$ is the set of all possible alignments between $\mathbf{z}$ and $\mathbf{z}^*$, and $D(\mathbf{z}, \mathbf{z}^*, \mathbf{a})$ is the total cost given the alignment $\mathbf{a}$. Notice that if $|\mathbf{z}| \neq |\mathbf{z}^*|$, *any* alignment leaves some events unaligned; also, rather than align two faraway events, it is cheaper to leave them unaligned if $C_{\text{delete}} + C_{\text{insert}} < |t - t^*|$. Algorithm 7.2 in section 7.E uses dynamic programming to compute the loss (7.10) and its corresponding alignment $\mathbf{a}$, similar to edit distance (Levenshtein, 1965) or dynamic time warping (Sakoe and Chiba, 1971; Listgarten et al., 2005). In practice we symmetrize the loss by specifying equal costs $C_{\text{insert}} = C_{\text{delete}} = C$.

7.5.2 Consensus Decoding

Since aligned events must have the same type, consensus decoding (7.9b) decomposes into *separately* choosing a set $\hat{\mathbf{z}}^{(k)}$ of type-k events for *each* $k = 1, 2, \ldots, K$, based on the particles' sets $\mathbf{z}_m^{(k)}$ of type-k events. Thus, we simplify the presentation by omitting $^{(k)}$ throughout this section. The loss function L defined in section 7.5.1 warrants:

Theorem 4. *Given* $\{\mathbf{z}_m\}_{m=1}^M$*, if we define* $\mathbf{z}_\sqcup = \bigsqcup_{m=1}^M \mathbf{z}_m$*, then* $\exists \hat{\mathbf{z}} \subseteq \mathbf{z}_\sqcup$ *such that*

$$\sum_{m=1}^M w_m L(\hat{\mathbf{z}}, \mathbf{z}_m) = \min_{\mathbf{z} \in \mathcal{Z}} \sum_{m=1}^M w_m L(\mathbf{z}, \mathbf{z}_m)$$

That is to say, there exists one subsequence of $\mathbf{z}_\sqcup$ *that achieves the minimum Bayes risk.*

The proof is given in section 7.F: it shows that if $\hat{\mathbf{z}}$ minimizes the Bayes risk but is *not* a subsequence of $\mathbf{z}_\sqcup$, then we can modify it to either improve its Bayes risk (a contradiction) or keep the same Bayes risk while making it a subsequence of $\mathbf{z}_\sqcup$ as desired.

Now we have reduced this decoding problem to a combinatorial optimization problem:

$$\hat{\mathbf{z}} = \arg\min_{\mathbf{z} \subseteq \mathbf{z}_\sqcup} \sum_{m=1}^M w_m L(\mathbf{z}, \mathbf{z}_m) \tag{7.11}$$

which is probably NP-hard, by analogy with the Steiner string problem (Gusfield, 1997).

Our heuristic (Algorithm 7.3 of section 7.F) seeks to iteratively improve $\hat{\mathbf{z}}$ by (1) using Algorithm 7.2 to find the optimal alignment $\mathbf{a}_m$ of $\hat{\mathbf{z}}$ with each $\mathbf{z}_m$, and then (2) repeating the following sequence of 3 phases until $\hat{\mathbf{z}}$ does not change. Each phase tries to update $\hat{\mathbf{z}}$ to decrease the weighted distance $\sum_{m=1}^M w_m D(\hat{\mathbf{z}}, \mathbf{z}_m, \mathbf{a}_m)$ which by

Theorem 4 is an upper bound of the Bayes risk $\sum_{m=1}^{M} w_m L(\hat{\mathbf{z}}, \mathbf{z}_m)$:[6]

Move Phase For each event in $\hat{\mathbf{z}}$, move its time to the weighted median (using weights w_m) of the times of all $\leq M$ events that $\mathbf{a}_m$ aligns it to (if any), while keeping the alignment edges. This selects the new time that minimizes $\sum_{m=1}^{M} w_m D(\hat{\mathbf{z}}, \mathbf{z}_m, \mathbf{a}_m)$.

Delete Phase For each event in $\hat{\mathbf{z}}$, delete it (together with any related edges in each $\mathbf{a}_m$) if this decreases $\sum_{m=1}^{M} w_m D(\hat{\mathbf{z}}, \mathbf{z}_m, \mathbf{a}_m)$.

Insert Phase If we inserted t into $\hat{\mathbf{z}}$, we would also modify each $\mathbf{a}_m$ to align t to the closest unaligned event in $\mathbf{z}_m$ (if any) provided that this decreased $D(\hat{\mathbf{z}}, \mathbf{z}_m, \mathbf{a}_m)$. Let $\Delta(t)$ be the resulting reduction in $\sum_{m=1}^{M} w_m D(\hat{\mathbf{z}}, \mathbf{z}_m, \mathbf{a}_m)$. Let $t^* = \arg\max_{t \in \mathbf{z}_\sqcup, t \notin \hat{\mathbf{z}}} \Delta(t)$. While $\Delta(t^*) > 0$, insert t^*.

The move or delete phase can consider events in any order, or in parallel; this does not change the result.

7.6 Experiments

We compare our particle smoothing method with the strong particle filtering baseline—our neural version of Linderman, Wang, and Blei (2017)'s Hawkes process particle filter—on multiple real-world and synthetic datasets. See section 7.G for training details (e.g., hyperparameter selection). PyTorch code can be found at `https://github.com/HMEIatJHU/neural-hawkes-particle-smoothing`.

[6] Note these phases compute $D(\hat{\mathbf{z}}, \mathbf{z}_m, \mathbf{a}_m)$ but not $L(\hat{\mathbf{z}}, \mathbf{z}_m)$, so they need not call the dynamic programming algorithm.

7.6.1 Missing-Data Mechanisms

We experiment with missingness mechanisms of the form

$$p_{\text{miss}}(\mathbf{z} \mid \mathbf{x} \sqcup \mathbf{z}) = \prod_{k_i @ t_i \in \mathbf{z}} \rho_{k_i} \prod_{k_i @ t_i \in \mathbf{x}} (1 - \rho_{k_i}) \tag{7.12}$$

meaning that each event in the complete sequence $\mathbf{x} \sqcup \mathbf{z}$ is independently censored with probability ρ_k that only depends on its event type k.[7] We consider both deterministic and stochastic missingness mechanisms. For the deterministic experiments, we set ρ_k for each k to be either 0 or 1, so that some event types are always observed while others are always missing. Then $p_{\text{miss}}(\mathbf{z} \mid \mathbf{x} \sqcup \mathbf{z}) = 1$ if $\mathbf{z}$ consists of precisely the events in $\mathbf{x} \sqcup \mathbf{z}$ that ought to go missing, and 0 otherwise. For our stochastic experiments, we simply set $\rho_k = \rho$ regardless of the event type k and experiment with $\rho = 0.1, 0.3, 0.5, 0.7, 0.9$. Then equation (7.12) can be written as $p_{\text{miss}}(\mathbf{z} \mid \mathbf{x} \sqcup \mathbf{z}) = (1-\rho)^{|\mathbf{x}|}\rho^{|\mathbf{z}|}$, whose value decreases exponentially in the number of missing events $|\mathbf{z}|$. As this depends on $\mathbf{z}$, the stochastic setting is definitely MNAR (not MCAR as one might have imagined).

7.6.2 Datasets

The datasets that we use in this chapter range from short sequences with mean length 15 to long ones with mean length > 300. For each of the datasets, we possess fully observed data that we use to train the model and the proposal distribution.[8] For each dev and test example, we censored out some events from the fully observed sequence, so we present the $\mathbf{x}$ part as input to the proposal distribution but we also know the $\mathbf{z}$

[7] Section 7.H discusses how $\boldsymbol{\rho}$ could be imputed when complete and incomplete data are both available.

[8] The focus of this chapter is on inference (imputation) under a given model, so training the model is simply a preparatory step. However, inference could be used to help train on incomplete data via the EM algorithm, provided that the missingness mechanism is known; see section 7.H for discussion.

part for evaluation purposes. Fully replicable details of the dataset preparation can be found in section 7.G, including how event types are defined and which event types are missing in the deterministic settings.

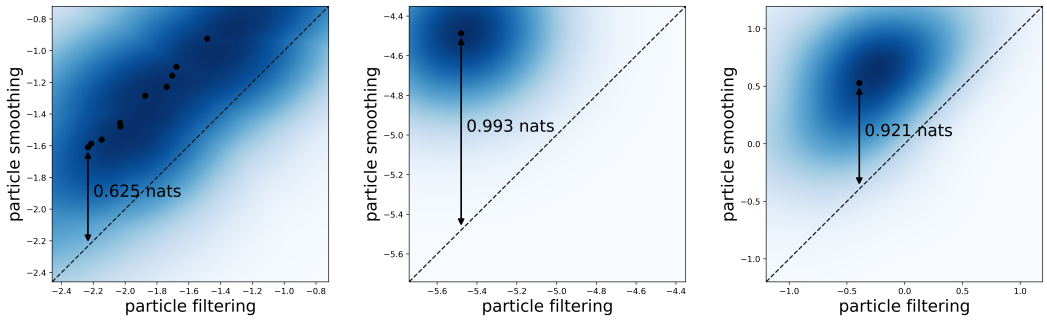
Synthetic Datasets We first checked that we could successfully impute unobserved events that are generated from *known* distributions. That is, when the generating distribution actually is a neural Hawkes process, could our method outperform particle filtering in practice? Is the performance consistent over multiple datasets drawn from different processes? To investigate this, we synthesized 10 datasets, each of which was drawn from a different neural Hawkes process with randomly sampled parameters.

Elevator System Dataset (Crites and Barto, 1996)**.** A multi-floor building is often equipped with multiple elevator cars that follow *cooperative* strategies to transport passengers between floors (Lewis, 1991; Bao et al., 1994; Crites and Barto, 1996). In this dataset, the events are which elevator car stops at which floor. The deterministic case of this domain is representative of many real-world cooperative (or competitive) scenarios—observing the activities of some players and imputing those of the others.

New York City Taxi Dataset (Whong, 2014)**.** Each medallion taxi in New York City has a sequence of time-stamped pick-up and drop-off events, where different locations have different event types. Figure 7.1 shows how we impute the pick-up events given the drop-off events (the deterministic missingness case).

7.6.3 Data Fitting Results

First, as an internal check, we measure *how probable* each ground truth reference $\mathbf{z}^*$ is under the proposal distribution constructed by each method, i.e., $\log q(\mathbf{z}^* \mid \mathbf{x})$. As shown in Figure 7.2, the improvement from particle smoothing is remarkably robust



(a) Synthetic Data **(b)** Elevator System **(c)** NYC Taxi

Figure 7.2: Scatterplots of neural Hawkes particle smoothing (y-axis) vs. particle filtering (x-axis) with a stochastic missingness mechanism ($\rho = 0.5$). Each point represents a single test sequence, and compares the values of $\log q(\mathbf{z}^* \mid \mathbf{x}) \;/\; |\mathbf{z}^*|$. Larger values mean that the proposal distribution is better at proposing the ground truth $\mathbf{z}^*$. Each dataset's scatterplot is converted to a cloud using kernel density estimation, with the centroid denoted by a black dot. A double-arrowed line indicates the improvement of particle smoothing over filtering. For the synthetic datasets, we draw ten clouds on the same figure and show the line for the dataset where smoothing improves the most. As we can see, the density is always well concentrated above $y = x$. That is, this is not merely an average improvement: *nearly every* ground truth $\mathbf{z}^*$ gets higher proposal probability! Particle smoothing performs well even on datasets where particle filtering performs badly.

across 12 datasets, improving *nearly every* sequence in each dataset. The plots for the deterministic missingness mechanisms are so boringly similar that we only show them in section 7.G.6 (Figure 7.4).

7.6.4 Decoding Results

For each $\mathbf{x}$, we now make a prediction by sampling an ensemble of $M = 50$ particles (section 7.4)[9] and constructing their consensus sequence $\hat{\mathbf{z}}$ (section 7.5.2). We use multinomial resampling since otherwise the effective sample size is very low (only 1–2 on some datasets).[10] We evaluate $\hat{\mathbf{z}}$ by its optimal transport distance (section 7.5.1)

[9] Increasing M would increase both effective sample size (ESS) and runtime.

[10] Any multinomial resampling step drives the ESS metric to M. This cannot guarantee better samples in general, but resampling did improve our decoding performance on all datasets.

to the ground truth $\mathbf{z}^*$. Note that $\forall \mathbf{a}$, we can decompose $D(\hat{\mathbf{z}}, \mathbf{z}^*, \mathbf{a})$ as

$$C \cdot (\underbrace{|\hat{\mathbf{z}}| + |\mathbf{z}^*| - 2|\mathbf{a}|}_{\text{total insertions+deletions}}) + \underbrace{\textstyle\sum_{(t,t^*)\in\mathbf{a}} |t - t^*|}_{\text{total distance moved}} \tag{7.13}$$

Letting $\mathbf{a}$ be the alignment that minimizes $D(\hat{\mathbf{z}}, \mathbf{z}^*, \mathbf{a})$, the former term measures how well $\hat{\mathbf{z}}$ predicts *which* events happened, and the latter measures how well $\hat{\mathbf{z}}$ predicts *when* those events happened. Different choices of C yield different $\hat{\mathbf{z}}$ with different trade-offs between these two terms. Intuitively, when $C \approx 0$, the decoder is free to insert and delete event tokens; as C increases, $\hat{\mathbf{z}}$ will tend to insert/delete fewer event tokens and move more of them.

Figure 7.3 plots the performance of particle smoothing (▲) vs. particle filtering (●) for the stochastic missingness mechanisms, showing the two terms above as the x and y coordinates. The very similar plots for the deterministic missingness mechanisms are in section 7.G.6 (Figure 7.5).[11]

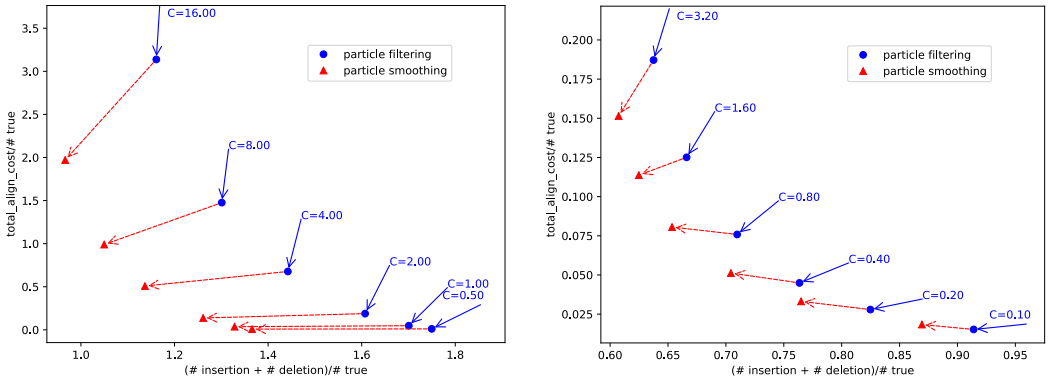
7.6.5 Sensitivity to Missingness Mechanism

For the stochastic missingness mechanisms, we also did experiments with different values of missing rate $\rho = 0.1, 0.3, 0.7, 0.9$. Our particle smoothing method consistently outperforms the filtering baseline in all the experiments (Figures 7.6 and 7.7 in section 7.G.7), similar to Figure 7.3.

7.6.6 Runtime

The theoretical runtime complexity is $O(MI)$ where M is the number of particles and I is the number of observed events. In practice, we generate the particles in parallel, leading to acceptable speeds of 300-400 milliseconds per event for the final method.

[11] We show the 2 real datasets only. The figures for the 10 synthetic datasets are boringly similar to these.



(a) Elevator System **(b)** NYC Taxi

Figure 7.3: Optimal transport distance of particle smoothing (▲) vs. particle filtering (●) on test data with a stochastic missingness mechanism ($\rho = 0.5$). In each figure, the x-axis is the total number of deletions and insertions in the test dataset, $\sum_{n=1}^{N}(|\hat{\mathbf{z}}_n| + |\mathbf{z}^*{}_n| - 2|\mathbf{a}_n|)$, and the y-axis is the total movement cost, $\sum_{n=1}^{N}\sum_{(t,t^*)\in\mathbf{a}_n}|t-t^*|$. Both axes are normalized by the true total number of missing events $\sum_{n=1}^{N}|\mathbf{z}^*{}_n|$, so the x-axis shows a fraction and the y-axis shows an average time difference. On each dataset, we show one ● per C. According to equation (7.13), $(C, 1)$, denoted by ←, turns out to be the *gradient* of $\sum_{n=1}^{N} D(\hat{\mathbf{z}}_n, \mathbf{z}^*{}_n, \mathbf{a}_n)$ at this ●. The <- - shows the actual improvement obtained by switching to particle smoothing (which is, indeed, an improvement because it has positive dot product with the gradient ←). The Pareto frontier (convex hull) of the ▲ symbols dominates the Pareto frontier of the ● symbols—lying everywhere to its left—which means that our particle smoothing method outperforms the filtering baseline.

More details about the wall-clock runtime can be found in section 7.G.8.

7.7 Discussion and Related Work

To our knowledge, this is the first time a bidirectional recurrent neural network has been extended to predict events in continuous time. Bidirectional architectures have proven effective at predicting linguistic words and their properties given their left *and right* contexts (Graves, Jaitly, and Mohamed, 2013; Bahdanau, Cho, and Bengio, 2015; Peters et al., 2018; Devlin et al., 2018): in particular, Lin and Eisner (2018) recently applied them to particle smoothing for discrete-time sequence tagging.

Previous work that infers unobserved events in continuous time exploits special properties of simpler models, including Markov jump processes (Rao and Teh, 2012; Rao and Teh, 2013), continuous-time Bayesian networks (Fan, Xu, and Shelton, 2010) and Hawkes processes (Shelton, Qin, and Shetty, 2018). Such properties no longer hold for our more expressive neural model, necessitating our approximate inference method.

Metropolis-Hastings would be an alternative to our particle smoothing method. The transition kernel could propose a single-event change to $\mathbf{z}$ (insert, delete, or move). Unfortunately, this would be quite slow for a neural model like ours, because any proposed change early in the sequence would affect the LSTM state and hence the probability of all subsequent events. Thus, a single move takes $O(|\mathbf{x} \sqcup \mathbf{z}|)$ time to evaluate. Furthermore, the Markov chain may mix slowly because a move that changes only one event may often lead to an incoherent sequence that will be rejected. The point of our particle smoothing is essentially to avoid such rejection by proposing a *coherent sequence of events*, left to right but considering future $\mathbf{x}$ events, from an approximation $q(\mathbf{z} \mid \mathbf{x})$ to the true posterior. (One might build a better Metropolis-Hastings algorithm by designing a transition kernel that makes use of our current proposal distribution, e.g., via particle Gibbs (Chopin and Singh, 2015).)

We also introduced an optimal transport distance between event sequences, which is a valid metric. It essentially regards each event sequence as a 0-1 function over times, and applies a variant of Wasserstein distance (Villani, 2008) or Earth Mover's distance (Kantorovitch, 1958; Levina and Bickel, 2001). Such variants are under active investigation (Benamou, 2003; Chizat et al., 2015; Frogner et al., 2015; Chizat et al., 2018). Our version allows event insertion and deletion during alignment, where

these operations can only apply to an entire event—we cannot align half of an event and delete the other half. Due to these constraints, dynamic programming rather than a linear programming relaxation is needed to find the optimal transport. Xiao et al. (2017a) also proposed an optimal transport distance between event sequences that allows event insertion and deletion; however, their insertion and deletion costs turn out to depend on the timing of the events in (we feel) a peculiar way.

We also gave a method to find a single "consensus" reconstruction with small average distance to our particles. This problem is related to Steiner string (Gusfield, 1997), which is usually reduced to multiple sequence alignment (MSA) (Mount, 2004) and heuristically solved by progressive alignment construction using a guide tree (Feng and Doolittle, 1987; Larkin et al., 2007; Notredame, Higgins, and Heringa, 2000) and iterative realignment of the initial sequences with addition of new sequences to the growing MSA (Hirosawa et al., 1995; Gotoh, 1996). These methods might also be tried in our setting. For us, however, the i^{th} event of type k is not simply a character in a finite alphabet such as $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ but a time that falls in the infinite set $[0, T)$. The substitution cost between two events of type k is then their time difference.

On multiple synthetic and real-world datasets, our method turns out to be effective at inferring the ground truth sequence of unobserved events. The improvement of particle smoothing upon particle filtering is substantial and consistent, showing the benefit of training a proposal distribution.

Appendices

7.A Little and Rubin (1987)'s Missing-Data Taxonomy

Little and Rubin (1987)'s classical taxonomy of MNAR, MAR, and MCAR mechanisms[12] was meant for graphical models. A graphical model has a *fixed set* of random variables. The missingness mechanisms envisioned by Little and Rubin (1987) simply decide which of those variables are suppressed in a joint observation. For them, an observed sample always reveals *which* variables were observed, and thus it reveals *how many* variables are missing.

In contrast, our incomplete event sequence is most simply described as a single random variable Y that is *partly* missing. If we tried to describe it using $|\mathbf{x} \sqcup \mathbf{z}|$ random variables with values like $k@t$, then the observed sample $\mathbf{x}$ would *not* reveal the number of missing variables $|\mathbf{z}|$ nor the total number of variables $|\mathbf{x} \sqcup \mathbf{z}|$. There would not be a fixed set of random variables.

To formulate our model in Little and Rubin's terms, we would need a fixed set of uncountably many random variables K_t where t ranges over the set of times. $K_t = k$ if there is an event of type k at time t, and otherwise $K_t = 0$. For some finite set of

[12] **Missing not at random (MNAR)** makes no assumptions about the missingness mechanism. **Missing at random (MAR)** is a modeling assumption: determining from data whether the MAR property holds is "almost impossible" (Mohan and Pearl, 2018). **Missing completely at random (MCAR)** is a simple special case of MAR.

times t, we observe a specific value $K_t > 0$, corresponding to some observed event. For all other times t, the value of K_t is missing, meaning that we do not know whether or not there is any event at time t, let alone the type of such an event. A crucial point is that 0 values are never observed in our setting, because we are never told that an event did *not* happen at time t. In contrast, a value > 0 (corresponding to an event) may be either observed or unobserved. Thus, the probability that K_t is missing depends on whether $K_t > 0$, meaning that this setting is MNAR.

We preferred to present our model (section 7.3.1) in terms of the finite sequences that are generated or read by our LSTMs. This simplified the notation later in the chapter. But it does not cure the MNAR property: see section 7.6.1.

Again, our presentation does not allow a Little and Rubin (1987) style formulation in terms of a finite fixed set of random variables, some of which have missing values. That formulation would work if we knew the total number of events I, and were simply missing the value k_i and/or t_i for some indices i. But in our setting, the number of events is itself missing: after each observed event i, we are missing J_i events where J_i is itself unknown. In other words, we need to impute even the number of variables in the complete sequence $\mathbf{x} \sqcup \mathbf{z}$, not just their values.

Our definition of MAR in section 7.3.1 is the correct generalization of Little and Rubin (1987)'s definition: namely, it is the case in which the second factor of equation (7.1) can be ignored. The ability to ignore that factor is precisely why anyone cares about the MAR case. This was mentioned at equation (7.1), and is discussed in conjunction with the EM algorithm in section 7.H.

Since missing-event settings tend to violate this desirable MAR property, all our experiments address MNAR problems. As Little and Rubin (1987) explained, the

more general case of MNAR data cannot be treated without additional knowledge. The difficulty is that identifying p *jointly* with p_{miss} becomes impossible. If you observe few 50-year-olds on your survey, you cannot know (beyond your prior) whether that's because there are few 50-year-olds, or because 50-year-olds are very likely to omit their age.

Fortunately, we do have additional knowledge in our setting. Joint identification of p and p_{miss} is unnecessary if either (1) one has separate knowledge of the missingness distribution p_{miss}, or (2) one has separate knowledge of the complete-data distribution p. In fact, both (1) and (2) hold in the MNAR experiments of this chapter (sections 7.6.1–7.6.2). But in general, if we know at least one of the distributions, then we can still infer the other (section 7.H).

7.A.1 Obtaining Complete Data

Readers might wonder why (2) above would hold in a missing-data situation. In practice, where would we obtain a dataset of complete event sequences (as in section 7.6.2) for supervised training of $p(\mathbf{x} \sqcup \mathbf{z})$?

In some event sequence scenarios, a training dataset of complete event sequences can be collected at extra expense. This is the hope in the medical and user-interface scenarios in section 7.1. For our imputation method to work on partially observed sequences $\mathbf{x} \sqcup \mathbf{z}$, their complete sequences should be distributed like the ones in the training dataset.

Other scenarios could be described as having **eventually complete** sequences. Complete information about each event sequence *eventually* arrives, at no extra expense, and that event sequence can then be used for training. For example, in the competitive game scenario in section 7.1), perhaps including wars and political

campaigns, each game's true complete history is revealed after the game is over and the need for secrecy has passed. While a game is underway, however, some events are still missing, and imputing them is valuable. Both (1) and (2) hold in these settings.

An interesting subclass of eventual completeness arises in monitoring settings such as epidemiology, journalism, and sensor networks. These often have **reporting delays**, so that one observes each event only some time after it happens. Yet one must make decisions at time $t < T$ based on the events that have been observed so far. This may involve imputing the past and predicting the future. The missingness mechanism for these reporting delays says that more recent events (soon before the current time t) are more likely to be missing. The probability that such an event would be missing depends on the specific distribution of delays, which can be learned with supervision once all the data have arrved.

We point out that in all these cases, the "complete" sequences $\mathbf{x} \sqcup \mathbf{z}$ that are used to train p do not actually have to be *causally* complete. It may be that in the real world, there are additional latent events $\mathbf{w}$ that cause the events in $\mathbf{x} \sqcup \mathbf{z}$ or mediate their interactions. Section 3.3.4 of Chapter 3 found that the neural Hawkes process was expressive enough in practice to ignore this causal structure and simply use $\mathbf{x} \sqcup \mathbf{z}$ sequences to directly train a neural Hawkes process model $p(\mathbf{x} \sqcup \mathbf{z})$ of the *marginal* distribution of $\mathbf{x} \sqcup \mathbf{z}$, without explicitly considering $\mathbf{w}$ in the model or attempting to sum over $\mathbf{w}$ values. The assumption here is the usual assumption that $\mathbf{x} \sqcup \mathbf{z}$ will have the same distribution in training and test data, and thus $\mathbf{w}$ will be missing in both, with the same missingness mechanism in both. By contrast, $\mathbf{z}$ is missing only in test data. It is not possible to impute $\mathbf{w}$ because it was not modeled explicitly, nor observed even in training data. However, it remains possible to impute $\mathbf{z}$ in test data based on

its distribution in training data.

7.B Complete Data Model Details

Our complete data model, such as a neural Hawkes process, gives the probability $p(\mathbf{x} \sqcup \mathbf{z})$ that $\mathbf{x} \sqcup \mathbf{z}$ will be the complete set of events on a given interval $[0, T)$. this probability can always be written in the factored form

$$\left(\prod_{i=0}^{I} \prod_{j=0}^{J_i} p(k_{i,j} @ t_{i,j} \mid \mathcal{H}(t_{i,j})) \right) \cdot p(@ \geq T \mid \mathcal{H}(T)) \tag{7.14}$$

where $p(k @ t \mid \mathcal{H}(t))$ denotes the probability density that the first event following $\mathcal{H}(t)$ (which is the set of events occurring *strictly* before t) will be $k @ t$, and $p(@ \geq t' \mid \mathcal{H}(t))$ denotes the probability that this event will fall at some time $\geq t'$.

Thus, the final factor of (7.14) is the probability that there are no more events on $[0, T)$ following the last event of $\mathbf{x} \sqcup \mathbf{z}$. The initial factor $p(k_0 @ t_0 \mid \mathcal{H}(t_0))$ is defined to be 1, since the boundary event $k_0 @ t_0$ is given (see section 7.3.1).

Chapter 3 has all the details about the neural Hawkes process. The intensity functions $\lambda_k(t \mid \mathcal{H}(t))$ are continuous on intervals during which no event occurs (note that $\mathcal{H}(t)$ is constant on such intervals). They jointly determine a distribution over the time of the next event after $\mathcal{H}(t)$, as used in every factor of equation (7.14). As it turns out in Chapter 3, $\log p(\, Y = \mathbf{x} \sqcup \mathbf{z})$ becomes

$$\sum_{\ell} \log \lambda_{k_\ell}(t_\ell \mid \mathcal{H}(t_\ell)) - \int_{t=0}^{T} \sum_{k=1}^{K} \lambda_k(t \mid \mathcal{H}(t)) dt \tag{7.15}$$

where the first sum ranges over all events $k_\ell @ t_\ell$ in $\mathbf{x} \sqcup \mathbf{z}$. The model can be trained by maximizing this log-likelihood; a full recipe can be found in section 2.2.

7.C Sequential Monte Carlo Details

Our main algorithm is presented as Algorithm 7.1. It covers both particle filtering and particle smoothing, with optional multinomial resampling.

In this section, we provide some additional details and notes on the design and operation of the pseudocode.

7.C.1 Explicit Formula for the Proposal Distribution

The proposal distribution $q(\mathbf{z} \mid \mathbf{x})$ factors as follows, and the pseudocode uses this factorization to construct $\mathbf{z}$ by sampling its individual events from left to right:

$$\prod_{i=0}^{I} \left(\prod_{j=1}^{J_i} \left(q(k_{i,j} @ t_{i,j} \mid \mathcal{H}(t_{i,j}), \mathcal{F}(t_{i,j})) \right) \cdot q(@ \geq t_{i+1} \mid \mathcal{H}(t_{i+1}), \mathcal{F}(t_{i,J_i})) \right) \tag{7.16}$$

Here the notation for $q(\cdot \mid \cdot)$ is the same as that for $p(\cdot \mid \cdot)$ in section 7.B. However, the $q(\cdot \mid \cdot)$ terms are proposal probabilities that condition on different evidence—not only the set $\mathcal{H}(t)$ of all events (observed and unobserved) at times $< t$, but also the set $\mathcal{F}(t)$ of events at times $> t$.[13] All of the proposal probabilities $q(\cdot \mid \cdot)$ are determined by the intensity functions in (7.5).

We can sample $\mathbf{z}$ from $q(\mathbf{z} \mid \mathbf{x})$ in chronological order: for each $0 \leq i \leq I$ in turn, draw a sequence of J_i unobserved events that follow the observed event $k_i @ t_i$. The probabilities of these J_i events are the inner factors in equation (7.16). This sequence ends (thereby determining J_i) if the next proposed event would have fallen after t_{i+1} and thus is preempted by the observed event $k_{i+1} @ t_{i+1}$. The probability of so ending the sequence corresponds to the $q(@ \geq t_{i+1} \mid \cdots)$ factor in equation (7.16).

[13] In particular, the second q factor above is the probability that the event at time t_{i,J_i} is the last one before t_{i+1}, given knowledge of all past events up through and including the one at t_{i,J_i}, and all future observed events starting with the one at t_{i+1}.

Algorithm 7.1 Sequential Monte Carlo — Particle Filtering/Smoothing

Input: observed sequence $\mathbf{x} = k_0@t_0, \ldots, k_{I+1}@t_{I+1}$ with $t_0 = 0, t_{I+1} = T$; model p; missingness mechanism p_{miss}; proposal distribution q; number of particles M; boolean flags $smooth$ and $resample$

Output: collection $\{(\mathbf{z}_1, w_1), \ldots, (\mathbf{z}_M, w_M)\}$ of weighted particles

procedure SEQUENTIALMONTECARLO($\mathbf{x}, p, p_{\text{miss}}, q, M, smooth, resample$)
  for $m = 1$ **to** M : ▷ *initialize the M weighted particles $(\mathbf{z}_m, w_m)$*
    $\mathbf{z}_m \leftarrow$ empty seq; $w_m \leftarrow 1$
    $\mathcal{H}_m \leftarrow$ empty stack ▷ *history $\mathcal{H}_m$ will be a stack of the past events*
    push $k_0@t_0$ onto $\mathcal{H}_m$ ▷ *boundary event 0 not generated by p*
  $\mathcal{F} \leftarrow$ empty stack ▷ *$\mathcal{F}$ is a stack of future observed events, with next event on top*
  for $i = I$ **downto** 0 :
    ▷ *init $\mathcal{F}$; later, as we reach each event, we'll pop it from $\mathcal{F}$ and push it onto $\mathcal{H}_m$ ($\forall m$)*
    push $k_{i+1}@t_{i+1}$ onto $\mathcal{F}$
  for $i = 0$ **to** I :
    ▷ *propose unobserved events over (t_i, t_{i+1}), then observe next event $k_{i+1}@t_{i+1}$*
    for $m = 1$ **to** M : ▷ *destructively extend $\mathbf{z}_m, w_m, \mathcal{H}_m$ with events on $(t_i, t_{i+1}]$*
      DRAWSEGMENT(i, m)
    if $resample$ & LOWESS() : RESAMPLE() ▷ *optional multinomial resampling*
  return $\{(\mathbf{z}_m, w_m / \sum_{m=1}^{M} w_m)\}_{m=1}^{M}$ ▷ *M particles with normalized weights*
procedure LOWESS ▷ *check if effective sample size is low*
  ESS $\leftarrow (\sum_{m=1}^{M} w_m)^2 / \sum_{m=1}^{M} (w_m)^2$
  if ESS $< M/2$: **return true**
  return false
procedure RESAMPLE ▷ *has access to global variables*
  ▷ *often draws multiple copies of good (high-weight) particles, 0 copies of bad ones*
  for $m = 1$ **to** M :
    draw $\tilde{m} \in \{1, \ldots, M\}$ where probability of choosing any $\tilde{m}$ is $\propto w_{\tilde{m}}$
    $\tilde{\mathbf{z}}_m \leftarrow \mathbf{z}_{\tilde{m}}$
  for $m = 1$ **to** M :
    $\mathbf{z}_m \leftarrow \tilde{\mathbf{z}}_m$; $w_m \leftarrow 1$ ▷ *update particles and their weights*

Algorithm 7.1 (Continued) Sequential Monte Carlo — Particle Filtering/Smoothing

procedure DRAWSEGMENT(i, m) ▷ *has access to global variables*
  ▷ *algorithm input p gives info to define intensity function $\lambda_k^p(t) \stackrel{def}{=} \lambda_k(t \mid \mathcal{H}_m)$*
  ▷ *algorithm input q gives info to define intensity function $\lambda_k^q(t) \stackrel{def}{=} \lambda_k(t \mid \mathcal{H}_m, \mathcal{F})$*
  ▷ *if $smooth =$ **false**, then $\lambda_k^q(t) = \lambda_k^p(t)$*
  ▷ *these functions consult the state of a left-to-right LSTM that's read $\mathcal{H}_m$ and possibly a right-to-left LSTM that's read $\mathcal{F}$*
  ▷ *define the total intensity functions $\lambda^p(t) \stackrel{def}{=} \sum_{k=1}^K \lambda_k^p(t)$ and $\lambda^q(t) \stackrel{def}{=} \sum_{k=1}^K \lambda_k^q(t)$*
  $i' \leftarrow i$; $j \leftarrow 0$; $t \leftarrow t_i$
  ▷ *t_i can be found as the time of the top element of $\mathcal{H}_m$ (currently an observed event)*
  repeat
    ▷ *add a new event to $\mathcal{H}_m$ with index $\langle i', j \rangle = \langle i, 1 \rangle, \ldots, \langle i, J_i \rangle, \langle i+1, 0 \rangle$*
    $j \leftarrow j + 1$
    repeat ▷ *thinning algorithm (see section 2.3)*
      ▷ *see section 7.C.4 for how to find λ^*; e.g., old λ^* still works if i unchanged*
      find any $\lambda^* \geq \sup \{\lambda^q(t') : t' \in (t, t_{i+1}]\}$
      draw $\Delta \sim \text{Exp}(\lambda^*)$, $u \sim \text{Unif}(0, 1)$
      $t \mathrel{+}= \Delta$ ▷ *time of next proposed event (before thinning)*
      if $t > t_{i+1}$: ▷ *t_{i+1} is the time of the top element of $\mathcal{F}$ (always an observed event)*
        ▷ *preempt proposed event by $k_{i+1}@t_{i+1}$ (popped from future into present)*
        $k@t \leftarrow$ pop $\mathcal{F}$; $i' \leftarrow i + 1$; $j \leftarrow 0$;
        break
    until $u\lambda^* \leq \lambda^q(t)$ ▷ *thinning: accept proposed time t only with prob $\frac{\lambda^q(t)}{\lambda^*} \leq 1$*
    ▷ *we've now chosen next event time $t_{i',j}$ to be t*
    ▷ *let t_{prev} denote the time of the top element of $\mathcal{H}_m$*
    if $i' = i$: ▷ *it's a missing event*
      draw $k \in \{1, \ldots, K\}$ where prob of k is $\propto \lambda_k^q(t)$ ▷ *choose event type*
      append $k@t$ to $\mathbf{z}_m$ ▷ *add our proposed event $k_{i',j}@t_{i',j}$*
      ▷ *new factor within q in denominator of (7.3); see section 7.C.3*
      $w_m \leftarrow w_m \,/\, (\exp(-\int_{t'=t_{\text{prev}}}^{t} \lambda^q(t')\, dt') \cdot \lambda_k^q(t))$
    if $i' \leq I$: ▷ *skip final boundary event $I+1$ (not generated by p)*
      ▷ *new factor within p in numerator of (7.3); see section 7.C.3*
      $w_m \leftarrow w_m \cdot (\exp(-\int_{t'=t_{\text{prev}}}^{t} \lambda^p(t')\, dt') \cdot \lambda_k^p(t))$
      push $k@t$ onto $\mathcal{H}_m$ ▷ *event $\langle i, j \rangle$ just generated now becomes part of the past*
      ▷ *new factor within p_{miss} in numerator of (7.3): missing or obs*
      $w_m \leftarrow w_m \cdot p_{\text{miss}}((k@t \in Z) = (i' = i) \mid \mathcal{H}_m)$
  until $i' = i + 1$

Equation (7.16) resembles equation (7.14), but it conditions each proposed unobserved event not only on the history but also on the future. Section 7.4.3 tries to train $q(\mathbf{z} \mid \mathbf{x})$ to approximate the target distribution $p(\mathbf{z} \mid \mathbf{x})$, by making $q(\cdot \mid \mathcal{H}, \mathcal{F}) \approx p(\cdot \mid \mathcal{H}, \mathcal{F})$. In other words, at each step, q should draw the next proposed event approximately from the posterior of the model p, even though we have no closed form computation for that posterior.

Just as equation (7.14) yields the formula (7.15) for $\log p$ when we use a neural Hawkes process model, equation (7.16) yields the following formula for $\log q(\mathbf{z} \mid \mathbf{x})$ when we use the proposal intensities from equation (7.5):

$$\sum_{\ell} \log \lambda^q_{k_\ell}(t_\ell \mid \mathcal{H}(t_\ell), \mathcal{F}(t_\ell)) - \int_{t=0}^{T} \sum_{k=1}^{K} \lambda^q_k(t \mid \mathcal{H}(t), \mathcal{F}(t))\, dt \tag{7.17}$$

where the first sum ranges over all events $k_\ell @ t_\ell$ in $\mathbf{z}$ only.

7.C.2 Managing LSTM State Information

The push and pop operations shown in the pseudocode must be implemented so that they also have the effect of updating LSTM configurations.

Our p uses a left-to-right LSTM to construct its state after reading all events so far from left to right (section 7.3.2). Since each particle posits a different event sequence, we maintain a separate LSTM configuration for each particle $m = 1, 2, \ldots, M$. If *smooth* = **true**, our q additionally uses a right-to-left LSTM whose state has read all future observed events from right to left (section 7.4.2). We maintain the configuration of this LSTM as well.

Specifically, in Algorithm 7.1, when we push an event to the stack $\mathcal{H}_m$ (lines 5 and 58), we update the configuration of particle m's left-to-right LSTM (including gates, cell memories and states).

If *smooth* $=$ **true**, then when we push an event to the stack $\mathcal{F}$ (line 9), we update the configuration of the right-to-left LSTM. Moreover, before updating that configuration, we push it onto an parallel stack, so that we can revert the update when we later *pop* the event from $\mathcal{F}$ (line 45).

These LSTM configurations provide the $\mathbf{h}(t)$ and $\bar{\mathbf{h}}(t)$ vectors for the computation of intensities $\lambda_k^p(t)$ and $\lambda_k^q(t)$ in equations (7.2) and (7.5). These intensities are needed in lines 54 and 57 of the algorithm.

7.C.3 Integral Computation

Section 2.2 construct a Monte Carlo estimator of the $\int_0^T$ integral in equation (7.15), by evaluating $\sum_k \lambda_k(t) \cdot T$ at a random $t \sim \mathrm{Unif}(0, 1)$. While even one such sample would provide an unbiased estimate, they draw $N = O(I)$ such samples, where I is the number of events, and average over these samples. This reduces the variance of the estimator, which decreases as $O(1/N)$. Notice that because they sample the N time points uniformly on $[0, T)$, longer intervals between observed events will tend to contain more points, which is appropriate.

We found that rather few samples could be used to estimate the integral. Indeed, even sampling at only I time points gave a standard deviation of log-likelihood—for the whole sequence—that was on the order of 0.1% of absolute.

Our particle methods in the present chapter involve *comparing probabilities*. For each observed sequence $\mathbf{x}$, we use (7.3) to reweight the M particles according to their probability under the model divided by their probability under the proposal distribution. This means contrasting *two* probabilities for each particle (the p and q probabilities). It also means *comparing* the resulting probability ratios across all M particles, resulting in the normalized weights of equation (7.3).

For each of the M particles, the p factor in equation (7.3) is obtained by exponentiating equation (7.15), while the q factor is obtained by exponentiating equation (7.17). This means that each of these $2M$ factors contains the $\exp$ of an integral. To make all of these integral estimates more comparable and thus reduce the variance in the importance weights w_m (equation (7.3)), we evaluate all $2M$ integrals at the same set of N time points (see section 7.G.8). This practice ensures a "paired comparison" among particles: w_m and $w_{m'}$ differ only because they have different intensities at the sampled points, and not also because they sample at different points.

In Algorithm 7.1, these integral estimates are accumulated gradually at lines 54 and 57. The idea is that particle $\mathbf{z}_m$ partitions $[0, T)$ into the intervals between successive events of $\mathbf{x} \sqcup \mathbf{z}_m$. Thus, the (estimated) integral over $[0, T)$ can be expressed by summing the (estimated) integrals over these intervals. The estimate over an interval uses only the small subset of the N time points that fall into the interval. When we exponentiate the integrals to convert from log-probabilities into probabilities, this sum turns into a product, as shown at lines 54 and 57.

This gradual accumulation method gives the same result as if we had computed each integral "all at once" before exponentiating. However, it is useful to begin weighting the particles before they are complete. After each event $k_i @ t_i$ (for $0 \leq i \leq I + 1$), the partial particles up through this event already have partial weights w_m. It is these partial weights that are used by the RESAMPLE procedure (when $resample =$ **true**).

In all experiments in this chapter, we first sampled $I + 1$ points uniformly on $[0, T)$, for an average of only 1 time point per interval. In addition, for each interval (t_i, t_{i+1}), we sampled 1 point uniformly on that interval if it did not yet contain any

points. Thus, $N \in [I+1, 2I+1]$.

Sampling at more points might be wise in settings where there are many missing events per interval (e.g., large ρ in section 7.6.1). This is especially true when $resample =$ **true**. Resampling allows us to try multiple extensions of a high-weight particle; at the next resampling step, we prefer to keep the extensions that fared best. The danger is that if only a few sampling points happen to fall between resampling steps, then we may make a poor (high-variance) estimate of which extensions fared best.

For our setting, however, we found only negligible changes in the results by increasing to 5 time points per interval (i.e., sampling $5I + 5$ points at the first step). Our evaluation metric (the minimum of equation (7.13) over all alignments $\mathbf{a}$) became slightly better for some values of C and slightly worse for others, but never by more than 2% relative. This is about the same variance that we get across different runs (with different random seeds) that have 1 time point per interval.

Thus, we report only the results of the faster scheme. We caution that other settings might be more sensitive to this hyperparameter settings. Thus, it might be wise to eliminate the hyperparameter altogether by estimating the integrals at lines 54 and 57 with a more sophisticated Monte Carlo integration method, such as the adaptive partitioning method of Baran, Demaine, and Katz (2008), which can bound the additive error of the estimate with high probability. This approach no longer provides a "paired comparison" across particles, nor does it need one.

7.C.4 Choice of λ^*

How do we construct the upper bound λ^* (line 40 of Algorithm 7.1)? For particle filtering, we follow the recipe in Chapter 3: we can express $\lambda^* = f_k(\max_t g_1(t) +$

$\ldots + \max_t g_n(t))$ where each summand $v_{kd} h_d(t) = v_{kd} \cdot o_{id} \cdot (2\sigma(2c_d(t)) - 1)$ is upper-bounded by $\max_{c \in \{c_{id}, \underline{c}_{id}\}} v_{kd} \cdot o_{id} \cdot (2\sigma(2c) - 1)$. Note that the coefficients v_{kd} may be either positive or negative.

For particle smoothing, we simply have more summands inside f_k so $\lambda^* = f_k(\max_t g_1(t) + \ldots + \max_t g_n(t) + \max_t \bar{g}_1(t) + \ldots + \max_t \bar{g}_{\bar{n}}(t))$ where each extra summand $u_{kd} \bar{h}_d(t) = u_{kd} \cdot \bar{o}_{id} \cdot (2\sigma(2\bar{c}_d(t)) - 1)$ is upper-bounded by $\max_{c \in \{\bar{c}_{id}, \underline{\bar{c}}_{id}\}} u_{kd} \cdot \bar{o}_{id} \cdot (2\sigma(2c) - 1)$ and each u_{kd} is the d^{th} element of vector $\mathbf{v}_k^\top \mathbf{B}$ (equation (7.5)). Note that the $\bar{o}_{id}, \bar{c}_{id}, \underline{\bar{c}}_{id}$ of newly added summands $\bar{g}$ are actually from the right-to-left LSTM while those of g are from the left-to-right LSTM.

7.C.5 Missing Data Factors in p

Recall that the joint model (7.1) includes a factor $p_{\text{miss}}(\mathbf{z} \mid \mathbf{x} \sqcup \mathbf{z})$, which appears in the numerator of the unnormalized importance weight (7.3). Regardless of the form of this factor, it could be multiplied into the particle's weight $\tilde{w}_m$ at the *end* of sampling (line 15 of Algorithm 7.1).

However, for some p_{miss} distributions, there is a better way. Algorithm 7.1 assumes that the missingness of each event $k@t$ depends only on that event and preceding events,[14] so that $p_{\text{miss}}(\mathbf{z} \mid \mathbf{x} \sqcup \mathbf{z})$ factors as

$$\prod_{\ell \in \text{indices}(\mathbf{z})} p_{\text{miss}}(k_\ell @ t_\ell \in Z \mid \{k_{\ell'} @ t_{\ell'} : \ell' \leq \ell\}) \cdot \prod_{\ell \in \text{indices}(\mathbf{x})} p_{\text{miss}}(k_\ell @ t_\ell \notin Z \mid \{k_{\ell'} @ t_{\ell'} : \ell' \leq \ell\}) \tag{7.18}$$

Algorithm 7.1 can thus *incrementally* incorporate the subfactors of equation (7.18), and does so at line 60 of Algorithm 7.1. For example, with the missingness mechanism

[14] This assumption could trivially be relaxed to allow it to also depend on the missingness of the preceding events, and/or on the future observed events $\mathcal{F}(t)$.

in our experiments, equation (7.12), the p_{miss} factor in line 60 is ρ_k if the event is unobserved (that is, $i' = i$) or $1 - \rho_k$ if it is observed.

These subfactors are therefore taken into account as the particles are constructed, and thus play a role in resampling.

7.C.6 Optional Missing Data Factors in q

We can optionally improve the particle filtering proposal intensities to incorporate the p_{miss} factor discussed in section 7.C.5 (in which case that factor will be multiplied into the denominator of (7.3) and not just the numerator). This makes $q(\mathbf{z} \mid \mathbf{x})$ better match $p(\mathbf{z} \mid \mathbf{x})$: it means we will rarely posit an unobserved event that would rarely have gone missing.

Specifically, if a completed-data event $k@t$ would have probability $\rho_k(t \mid \mathcal{H}(t))$ of going missing given the preceding events $\mathcal{H}(t)$, it is wise to define $\lambda_k^q(t \mid \mathcal{H}(t)) = \lambda_k^p(t \mid \mathcal{H}(t)) \cdot \rho_k(t \mid \mathcal{H}(t))$.

We do include this extra ρ_k factor when defining λ_k^q for our experiments (section 7.6); that is, we modify the definition of λ_k^q at line 29. The factor is particularly simple in our experiments, where ρ_k is constant for each k.

Was this factor really necessary in the case of particle smoothing? One might say no: particle smoothing already tries to ensure through training that the proposal distribution will incorporate p_{miss}. That is because section 7.4.3 aims to train $\lambda_k^q(t \mid \mathcal{H}(t), \mathcal{F}(t))$ so that the resulting $q(\mathbf{z} \mid \mathbf{x}) \approx p(\mathbf{z} \mid \mathbf{x})$, and the posterior distribution $p(\mathbf{z} \mid \mathbf{x})$ does condition on the missingness of $\mathbf{z}$.

Still, if the ρ_k factor is known, why not include it explicitly in the proposal distribution, instead of having to train the BiLSTM to mimic it? Thus, in effect, we

have modified the right-hand side of equation (7.5) to include a factor of ρ_k. This yields a more expressive and better-factored family of proposal distributions: missingness is now handled by the known ρ_k factor and the BiLSTM does not have to explain it. Additionally, our proposal distribution becomes more conservative about proposing missing events, because having a lot of missing events is *a posteriori* improbable. In other words, p_{miss} as given in equation (7.12) falls off with the number of missing events $|\mathbf{z}|$.

Modifying equation (7.5) in this way is particularly useful in the special case $\rho_k = 0$ (i.e., event type k is never missing and should not be proposed). There, it enforces the hard constraint that $\lambda_k^q = 0$ (something that the BiLSTM by itself could not achieve); and since this constraint is enforced regardless of the BiLSTM parameters, the events of type k appropriately become irrelevant to the training of the BiLSTM, which can focus on predicting other event types.

7.C.7 Events with Equal Times

In contrast to the notation in the main chapter, our pseudocode is written in terms of sequence of events, rather than sets of events. As a result, it can handle the generalization noted in footnote 4, where a 0 delay is allowed between an event and the preceding event in the complete sequence. If this occurs, it means that multiple events have fallen at the same time—yet they still have a well-defined order in which they are generated and read by the LSTM.

An unobserved event may have a 0 delay, if line 41 proposes $\Delta = 0$ and the proposal is accepted. The neural Hawkes model can in principle make such a proposal, but it has zero probability. However, it might have positive probability under a slightly different model.

An observed event may also have a 0 delay, if $t = t_{i+1}$ at line 43 and the proposal is accepted.[15] In this way, it is possible for the proposal distribution to propose any number of unobserved events at time t_{i+1} and immediately before the actual observed event $k_{i+1}@t_{i+1}$. However, once the proposal distribution happens to propose $\Delta > 0$, the actual observed event $k_{i+1}@t_{i+1}$ will preempt the proposal, ending this sequence of J_i unobserved events.

7.D Right-to-Left Continuous-Time LSTM

Here we give details of the right-to-left LSTM from section 7.4.2. Note that this set of formulas is nearly the same as that of section 3.1—after all, it is a continuous-time LSTM that has the same architecture as the one in the neural Hawkes process. The difference is that it reads only the observed events, and does so from right to left. The two LSTMs do not share parameters.

The hidden state $\bar{\mathbf{h}}(t)$ is continually obtained from the memory cells $\mathbf{c}(t)$ as the cells decay (in reversed time):

$$\bar{\mathbf{h}}(t) = \mathbf{o}_i \odot (2\sigma(2\bar{\mathbf{c}}(t)) - 1) \text{ for } t \in [t_{i-1}, t_i) \tag{7.19}$$

where the interval $[t_{i-1}, t_i)$ has consecutive observations $k_{i-1}@t_{i-1}$ and $k_i@t_i$ as endpoints. At t_i, the continuous-time LSTM reads $k_i@t_i$ and updates the current (decayed) hidden cells $\bar{\mathbf{c}}(t)$ to new initial values $\bar{\mathbf{c}}_{i-1}$, based on the current (decayed) hidden state $\bar{\mathbf{h}}(t_i)$, as follows:

$$\bar{\mathbf{i}}_{i-1} \leftarrow \sigma\left(\mathbf{W}_{\mathrm{i}}\mathbf{k}_i + \mathbf{U}_{\mathrm{i}}\bar{\mathbf{h}}(t_i) + \mathbf{d}_{\mathrm{i}}\right) \tag{7.20a}$$

[15] It may seem improbable to propose $t = t_{i+1}$ exactly, but if $t_i = t_{i+1}$, then proposing an unobserved event between these two observed events is just a case of proposing with 0 delay, as in the previous paragraph.

$$\bar{\mathbf{f}}_{i-1} \leftarrow \sigma\left(\mathbf{W}_{\mathrm{f}}\mathbf{k}_i + \mathbf{U}_{\mathrm{f}}\bar{\mathbf{h}}(t_i) + \mathbf{d}_{\mathrm{f}}\right) \tag{7.20b}$$

$$\bar{\mathbf{z}}_{i-1} \leftarrow 2\sigma\left(\mathbf{W}_{\mathrm{z}}\mathbf{k}_i + \mathbf{U}_{\mathrm{z}}\bar{\mathbf{h}}(t_i) + \mathbf{d}_{\mathrm{z}}\right) - 1 \tag{7.20c}$$

$$\bar{\mathbf{o}}_{i-1} \leftarrow \sigma\left(\mathbf{W}_{\mathrm{o}}\mathbf{k}_i + \mathbf{U}_{\mathrm{o}}\bar{\mathbf{h}}(t_i) + \mathbf{d}_{\mathrm{o}}\right) \tag{7.20d}$$

$$\bar{\mathbf{c}}_{i-1} \leftarrow \bar{\mathbf{f}}_{i-1} \odot \bar{\mathbf{c}}(t_i) + \bar{\mathbf{i}}_{i-1} \odot \bar{\mathbf{z}}_{i-1} \tag{7.20e}$$

$$\underline{\bar{\mathbf{c}}}_{i-1} \leftarrow \underline{\bar{\mathbf{f}}}_{i-1} \odot \underline{\bar{\mathbf{c}}}_{i} + \underline{\bar{\mathbf{i}}}_{i-1} \odot \bar{\mathbf{z}}_{i-1} \tag{7.20f}$$

$$\bar{\boldsymbol{\delta}}_{i-1} \leftarrow f\left(\mathbf{W}_{\mathrm{d}}\mathbf{k}_i + \mathbf{U}_{\mathrm{d}}\bar{\mathbf{h}}(t_i) + \mathbf{d}_{\mathrm{d}}\right) \tag{7.20g}$$

The vector $\mathbf{k}_i \in \{0,1\}^K$ is the i^{th} input: a one-hot encoding of the new event k_i, with non-zero value only at the entry indexed by k_i. Then, $\bar{\mathbf{c}}(t)$ for $t \in [t_{i-1}, t_i)$ is given by (7.21), which continues to control $\bar{\mathbf{h}}(t)$ except that i has now decreased by 1.

$$\bar{\mathbf{c}}(t) \stackrel{\text{def}}{=} \underline{\bar{\mathbf{c}}}_{i-1} + (\bar{\mathbf{c}}_{i-1} - \underline{\bar{\mathbf{c}}}_{i-1}) \exp\left(-\bar{\boldsymbol{\delta}}_{i-1}(t_i - t)\right) \tag{7.21}$$

On the interval $[t_{i-1}, t_i)$, $\bar{\mathbf{c}}(t)$ follows an exponential curve that begins at $\bar{\mathbf{c}}_{i-1}$ (in the sense that $\lim_{t \to t_i^-} \bar{\mathbf{c}}(t) = \bar{\mathbf{c}}_{i-1}$) and decays, as time t decreases, toward $\underline{\bar{\mathbf{c}}}_{i-1}$.

7.E Optimal Transport Distance Details

Pseudocode is presented in Algorithm 7.2 for finding optimal transport distance and the corresponding alignment. In the remainder of this section, we prove that optimal transport distance is a valid metric.

It is trivial that OTD is non-negative, since movement, deletion and insertion costs are all positive.

It is also trivial to prove that the following statement is true:

$$L(\mathbf{z}_1, \mathbf{z}_2) = 0 \Leftrightarrow \mathbf{z}_1 = \mathbf{z}_2, \tag{7.22}$$

where $\mathbf{z}_1$ and $\mathbf{z}_2$ are two sequences. If $\mathbf{z}_1$ is not identical to $\mathbf{z}_2$, the distance of them must be larger than 0 since we have to do some movement, insertion or deletion to make them exactly matched, so the right direction of equation (7.22) holds. If the distance between $\mathbf{z}_1$ and $\mathbf{z}_2$ is zero, which means they are already matched without any operations, $\mathbf{z}_1$ and $\mathbf{z}_2$ must be identical, thus the left direction of equation (7.22) holds.

OTD is symmetric, that is, $L(\mathbf{z}_1, \mathbf{z}_2) = L(\mathbf{z}_2, \mathbf{z}_1)$, if we set $C_{\text{insert}} = C_{\text{delete}}$. Suppose that $\mathbf{a}$ is an alignment between $\mathbf{z}_1$ and $\mathbf{z}_2$. It's easy to see that the only difference between $D(\mathbf{z}_1, \mathbf{z}_2, \mathbf{a})$ and $D(\mathbf{z}_2, \mathbf{z}_1, \mathbf{a})$ [16] is that the insertion and deletion operations are exchanged. For example, if we delete a token $t_i \in \mathbf{z}_1$ when calculating $D(\mathbf{z}_1, \mathbf{z}_2, \mathbf{a})$, we should insert a token at t_i to $\mathbf{z}_2$ when calculating $D(\mathbf{z}_2, \mathbf{z}_1, \mathbf{a})$. If we set $C_{\text{insert}} = C_{\text{delete}}$, we have

$$D(\mathbf{z}_1, \mathbf{z}_2, \mathbf{a}) = D(\mathbf{z}_2, \mathbf{z}_1, \mathbf{a}), \quad \forall \mathbf{a} \in \mathcal{A}(\mathbf{z}_1, \mathbf{z}_2). \tag{7.23}$$

Therefore, we could obtain

$$\begin{aligned} L(\mathbf{z}_1, \mathbf{z}_2) &= \min_{\mathbf{a}^* \in \mathcal{A}(\mathbf{z}_1, \mathbf{z}_2)} D(\mathbf{z}_1, \mathbf{z}_2, \mathbf{a}^*) \\ &= \min_{\mathbf{a}^* \in \mathcal{A}(\mathbf{z}_1, \mathbf{z}_2)} D(\mathbf{z}_2, \mathbf{z}_1, \mathbf{a}^*) = L(\mathbf{z}_2, \mathbf{z}_1) \end{aligned}$$

[16] We abuse the notation $\mathbf{a}$, which we think could represent both the movement from $\mathbf{z}_1$ to $\mathbf{z}_2$ and from $\mathbf{z}_2$ to $\mathbf{z}_1$.

Finally let's prove that OTD satisfies triangle inequality, that is:

$$L(\mathbf{z}_1, \mathbf{z}_2) + L(\mathbf{z}_2, \mathbf{z}_3) \geq L(\mathbf{z}_1, \mathbf{z}_3), \tag{7.25}$$

where $\mathbf{z}_1$, $\mathbf{z}_2$ and $\mathbf{z}_3$ are three sequences. This property could be proved intuitively. Suppose that the operations on $\mathbf{z}_1$ with minimal costs to make $\mathbf{z}_1$ matched to $\mathbf{z}_2$ are denoted by $o_1, o_2, \ldots, o_{n_1}$, and those on $\mathbf{z}_2$ to make $\mathbf{z}_2$ matched to $\mathbf{z}_3$ are denoted by $o'_1, o'_2, \ldots, o'_{n_2}$. o_i could be a deletion, insertion or movement on a token. To make $\mathbf{z}_1$ matched to $\mathbf{z}_3$, one possible way, which is not necessarily the optimal, is to do $o_1, o_2, \ldots, o_{n_1}, o'_1, o'_2, \ldots, o'_{n_2}$ on $\mathbf{z}_1$. Since the total cost is the accumulation of the cost of each operation, and the operations on $\mathbf{z}_1$ above to make $\mathbf{z}_1$ matched to $\mathbf{z}_3$ might not be optimal, the triangle inequality equation (7.25) holds.

7.F Approximate MBR Details

Our approximate consensus decoding algorithm is given as Algorithm 7.3. In the remainder of this section, we prove Theorem 4 from section 7.5.2, namely:

Theorem 1. *Given* $\{\mathbf{z}_m\}_{m=1}^M$*, if we define* $\mathbf{z}_\sqcup = \bigsqcup_{m=1}^M \mathbf{z}_m$*, then* $\exists \hat{\mathbf{z}} \subseteq \mathbf{z}_\sqcup$ *such that*

$$\textstyle\sum_{m=1}^M w_m L(\hat{\mathbf{z}}, \mathbf{z}_m) = \min_{\mathbf{z} \in \mathcal{Z}} \sum_{m=1}^M w_m L(\mathbf{z}, \mathbf{z}_m)$$

That is to say, there exists one subsequence of $\mathbf{z}_\sqcup$ *that achieves the minimum Bayes risk.*

Proof. Here we assume that there is only one type of event. Since the distances of different types of events are calculated separately, our conclusion is easy to be extended to the general case.

Suppose $\hat{\mathbf{z}}$ is an optimal decode, that is,

$$\sum_{m=1}^{M} w_m L(\mathbf{z}_m, \hat{\mathbf{z}}) = \min_{\mathbf{z}\in\mathcal{Z}} \sum_{m=1}^{M} w_m L(\mathbf{z}_m, \mathbf{z}).$$

If $\hat{\mathbf{z}} \subseteq \mathbf{z}_{\sqcup}$, the proof is done. If not, we can choose some $t_i \notin \mathbf{z}_{\sqcup}$. Let $t_l = \max\{t \in \mathbf{z}_{\sqcup} : t < t_i\}$ and $t_r = \min\{t \in \mathbf{z}_{\sqcup} : t > t_i\}$. (These sets are nonempty because $\mathbf{z}_{\sqcup}$ always contains the endpoints 0 and T.) We will show that if we move t_i around, as long as $t_i \in [t_l, t_r]$, the weighted optimal transport distance, i.e. $\sum_{m=1}^{M} w_m L(\mathbf{z}_m, \hat{\mathbf{z}})$, will neither increase nor decrease.

Suppose $\hat{\mathbf{a}} = \arg\min_{\mathbf{a}_m \in \mathcal{A}(\mathbf{z}_m, \hat{\mathbf{z}})} \sum_{m=1}^{M} w_m D(\mathbf{z}_m, \hat{\mathbf{z}}, \mathbf{a}_m)$. Let's use $r(t)$ to indicate the weighted transport distance of $\hat{\mathbf{z}}$ with fixed alignment if we move t_i to t, that is,

$$r(t) \stackrel{\text{def}}{=} \sum_{m=1}^{M} w_m D(\mathbf{z}_m, \hat{\mathbf{z}}(t), \hat{\mathbf{a}}),$$

where $\hat{\mathbf{z}}(t)$ is the sequence $\hat{\mathbf{z}}$ with t_i moved to t. Because $\hat{\mathbf{z}}(t_i)$ is an optimal decode, and $\hat{\mathbf{a}}$ is the optimal alignment for $\hat{\mathbf{z}}(t_i)$, we should have

$$r(t_i) = \min_t r(t).$$

Note that the transport distance is comprised of three parts: deletion, insertion and alignment costs. Since every $\hat{\mathbf{a}}$ is fixed, if we change t, only the alignment cost that related to token t will affect $r(t)$. This part of $r(t)$ is linear to t, since we have a constraint $t \in [t_l, t_r]$, which guarantees that it will not cross any other tokens in $\mathbf{z}_{\sqcup}$.

Since $r(t)$ is linear to $t \in [r_l, t_r]$ and $r(t)$ gets minimized at $t_i \in (t_l, t_r)$, we conclude that

$$r(t) = r(t_i) = \text{Const}, \forall t \in [t_l, t_r].$$

Since $r(t)$ is the upper bound of the weighted optimal transport distance, namely $\sum_{m=1}^{M} w_m L(\mathbf{z}_m, \hat{\mathbf{z}}(t))$, which also gets the same minimal value at $t_i \in (t_l, t_r)$ as $r(t)$, we could conclude that $\forall t \in [t_l, t_r]$:

$$\sum_{m=1}^{M} w_m L(\mathbf{z}_m, \hat{\mathbf{z}}(t)) = \sum_{m=1}^{M} w_m L(\mathbf{z}_m, \hat{\mathbf{z}}(t_i)) = \text{Const}$$

Therefore we could move token t_i to either t_l or t_r without increasing the Bayes risk. We could do this movement for each $t_i \notin \mathbf{z}_{\sqcup}$ to get a new decode $\hat{\mathbf{z}} \subseteq \mathbf{z}_{\sqcup}$, which is also an optimal decode.

□

7.G Experimental Details

In this section, we elaborate on the details of data generation, processing, and experimental results.

In all of our experiments, the distribution p is trained on the complete (uncensored) version of the training data. The system is then asked to complete the incomplete (censored) version of the test (or dev) data. For particle smoothing, the proposal distribution is trained using both the complete and incomplete versions of the training data, as explained at the end of section 7.4.3. We used the Adam algorithm with its default settings (Kingma and Ba, 2015). Adam is a stochastic gradient optimization algorithm that continually adjusts the learning rate in each dimension based on adaptive estimates of low-order moments. Each training example for Adam is a complete event sequence $\mathbf{x} \sqcup \mathbf{z}$ over some time interval $[0, T)$. We stop training early when we detect that log-likelihood has stopped increasing on the held-out development dataset. We do no other regularization.

Dataset	K	# of Event Tokens			Sequence Length		
		Train	Dev	Test	Min	Mean	Max
Synthetic	4	≈ 74967	≈ 7513	≈ 7507	10	≈ 15	20
NYCTaxi	10	157916	15826	15808	22	32	38
Elevator	10	313043	31304	31206	235	313	370

Table 7.1: Statistics of each dataset. We write "$\approx N$" to indicate that N is the average value over multiple datasets of one kind (synthetic); the variance is small in each such case.

7.G.1 Dataset Statistics

Table 7.1 shows statistics about each dataset that we use in this chapter.

7.G.2 Training Details

We used single-layer LSTMs (Hochreiter and Schmidhuber, 1997), selected the number D of hidden nodes of the left-to-right LSTM, and then D' of the right-to-left one from a small set $\{16, 32, 64, 128, 256, 512, 1024\}$ based on the performance on the dev set of each dataset. The best-performing (D, D') pairs are $(256, 128)$ on Synthetic, $(256, 256)$ on Elevator $(256, 256)$ on NYC Taxi, but we empirically found that the model performance is robust to these hyperparameters. For the chosen (D, D') pair on each dataset, we selected β based on the performance on the dev set, and $\beta = 1.0$ yields the best performance across all the datasets we use. For learning, we used Adam with its default settings (Kingma and Ba, 2015).

Our Monte Carlo integral estimates are in fact unbiased (section 7.C.3). As a result, our stochastic gradient estimate is also unbiased, as required (assuming that the complete data is distributed according to p). Why? Since $\beta = 1$, our stochastic gradient is simply equation (7.6). No particle filtering or smoothing is used to estimate equation (7.6), because we train it using complete data, as explained in the last long

paragraph of section 7.4.3. The only randomness is the integral over $[0, T)$ (similar to the one in equation (7.15)) that is required to estimate the term $\log q(\mathbf{z} \mid \mathbf{x})$ in equation (7.6): as just noted, this integral estimate is unbiased.

It is true that if $\beta < 1$, we would compute the exclusive KL gradient using particle filtering or smoothing with M particles, and this would introduce bias in the gradient. Nonetheless, since the bias vanishes as $M \to \infty$, it would be possible to restore a theoretical convergence guarantee by increasing M at an appropriate rate as SGD proceeds (Spall, 2005, page 107).[17]

7.G.3 Details of the Synthetic Datasets

Each of the ten neural Hawkes processes has its parameters sampled from the uniform distribution $\mathrm{Unif}[-1.0, 1.0]$. Then a set of event sequences is drawn from each of them via the plain vanilla thinning algorithm (see section 2.3). For each of the ten synthetic datasets, we took $K = 4$ as the number of event types. To draw each event sequence, we first chose the sequence length I (number of event tokens) uniformly from $\{11, 12, \ldots, 20\}$ and then used the thinning algorithm to sample the first I events over the interval $[0, \infty)$. For subsequent training or testing, we treated this sequence (appropriately) as the complete set of events observed on the interval $[0, T)$ where $T = t_I$, the time of the last generated event.

We generate 5000, 500 and 500 sequences for each training, dev, and test set respectively. For the missingness mechanism: in the deterministic settings, we censor all events of type 3 and 4—in other words, we set $\rho_1 = \rho_2 = 0$ and $\rho_3 = \rho_4 = 1$; in the stochastic settings, we set $\rho_k = 0.5$ for all k.

[17] SGD methods succeed, both theoretically and practically, with even high-variance estimates of the batch gradient (e.g., where each stochastic estimate is derived from a single randomly chosen training example). Thus, one should be fine with a noisy sampling-based gradient as long as it is unbiased.

7.G.4 Elevator System Dataset Details

We examined our method in a simulated 5-floor building with 2 elevator cars. During a typical afternoon down-peak rush hour (when passengers go from floor-2,3,4,5 down to the lobby), elevator cars travel to each floor and pick up passengers that have (stochastically) arrived there according to a traffic profile (Bao et al., 1994). Each car will also avoid floors that already are or will soon be taken care of by the other. Having observed when and where car-1 has stopped (to pick up or drop off passengers) over this hour, we are interested in when and where car-2 has stopped during the same time period. In this dataset, each event type is a tuple of (car number, floor number) so there are $K = 10$ in total in this simulated 5-floor building with 2 elevator cars.

Passenger arrivals at each floor are assumed to follow a inhomogeneous Poisson process, with arrival rates that vary during the course of the day. The simulations we use follows a human-recorded traffic profile (Bao et al., 1994) which dictates arrival rates for every 5-minute interval during a typical afternoon down-peak rush hour. Table 7.2 shows the mean number of passengers (who are going to the lobby) arriving at floor-2,3,4,5 during each 5-minute interval.

We simulated the elevator behavior following a naive baseline strategy documented in Crites and Barto (1996).[18] In details, each car has a small set of primitive actions. If it is stopped at a floor, it must either "move up" or "move down". If it is in motion between floors, it must either "stop at the next floor" or "continue past the next floor". Due to passenger expectations, there are two constraints on these actions: a car cannot pass a floor if a passenger wants to get off there and cannot turn until it has serviced all the car buttons in its current direction. Three additional action constraints were

[18] We rebuilt the system in Python following the original Fortran code of Crites and Barto (1996).

START TIME (MIN)	00	05	10	15	20	25	30	35	40	45	50	55
MEAN # PASSENGER	1	2	4	4	18	12	8	7	18	5	3	2

Table 7.2: The Down-Peak Traffic Profile

made in an attempt to build in some primitive prior knowledge: 1) a car cannot stop at a floor unless someone wants to get on or off there; 2) it cannot stop to pick up passengers at a floor if another car is already stopped there; 3) given a choice between moving up and down, it should prefer moving up (since the down-peak traffic tends to push the cars toward the bottom of the building). Because of this last constraint, the only real choices left to each car are the stop and continue actions, and the baseline strategy always chooses to continue. The actions of the elevator cars are executed asynchronously since they may take different amounts of time to complete.

We repeated the (one-hour) simulation 700 times to collect the event sequences, each of which has around 300 time-stamped records of which car stops at which floor. We randomly sampled disjoint train, dev and test sets with 500, 100 and 100 sequences respectively.

For the missingness mechanism: in the deterministic settings, we set $\rho_k = 0$ for $k = 1, \ldots, 5$ and $\rho_k = 1$ for $k = 6, \ldots, 10$ (meaning that the events (of arriving at floor 1, 2, ..., 5) of car 1 are all observed, but those of car 2 are not); in the stochastic settings, we set $\rho_k = 0.5$ for all k.

7.G.5 New York City Taxi Dataset Details

The New York City Taxi dataset (section 7.6.2) includes 189,550 taxi pick-up and drop-off records in the city of New York in 2013. Each record has its medallion ID, driver license and time stamp. Each combination of medallion ID and driver

license naturally forms a sequence of time-stamped pick-up and drop-off events. Following the processing recipe of previous work (Du et al., 2016), we construct shorter sequences by breaking each long sequence wherever the temporal gap between a drop-off event and its following pick-up event is larger than six hours. Then the left boundary of this gap is treated as the EOS of the sequence before it, while the right boundary is set as the BOS of the following sequence.

We randomly sampled a month from 2013 and then randomly sampled disjoint train, dev and test sets with 5000, 500 and 500 sequences respectively from that month.

In this dataset, each event type is a tuple of (location, action). The location is one of the 5 boroughs {Manhattan, Brooklyn, Queens, The Bronx, Staten Island}. The action can be either pick-up or drop-off. Thus, there are $K = 5 \times 2 = 10$ event types in total.

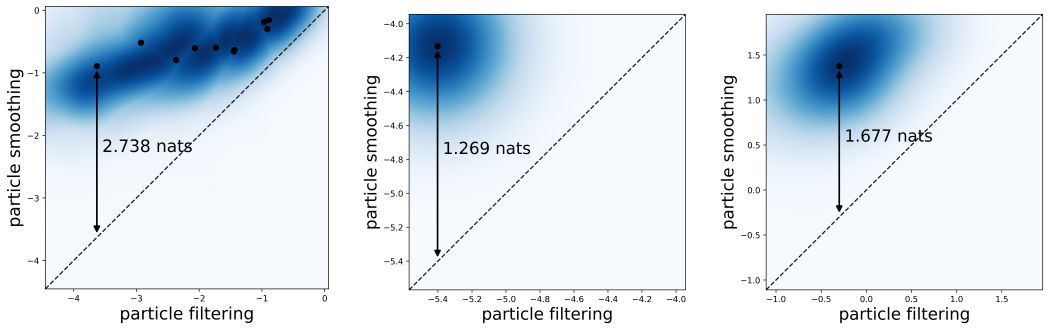
For the missingness mechanism: in the deterministic settings, we set $\rho_k = 0$ for $k = 1, \ldots, 5$ and $\rho_k = 1$ for $k = 6, \ldots, 10$ (which means that all drop-off events but no pick-up events are observed); in the stochastic settings, we set $\rho_k = 0.5$ for all k.

7.G.6 Experiments with Deterministic Missingness Mechanisms

We show our experimental results for the deterministic missingness mechanisms in Figures 7.4 and 7.5.

7.G.7 Sensitivity Experiment Details

Figures 7.6 and 7.7 displays the optimal transport distance with various values of ρ: our particle smoothing method consistently outperforms the filtering baseline.



(a) Synthetic **(b)** Elevator System **(c)** NYC Taxi

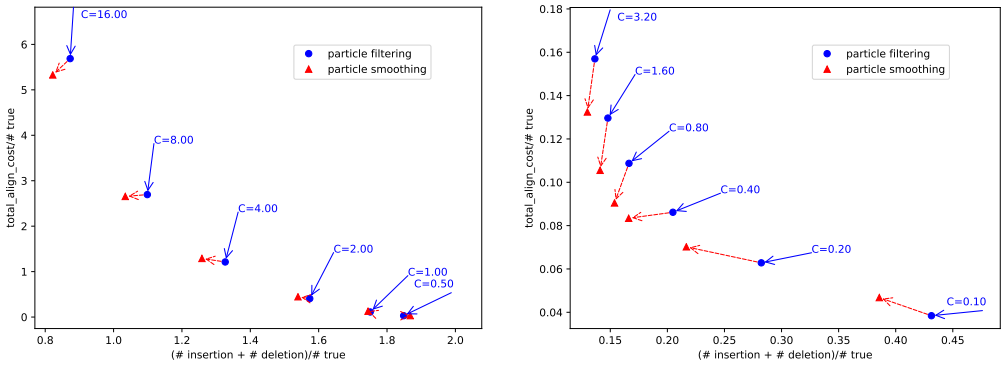
Figure 7.4: Scatterplots with a deterministic missingness mechanism. Again, the method works, with very similar qualitative behavior to Figure 7.2.

7.G.8 Wall-Clock Runtime Details

A given run of particle smoothing begins by drawing $O(I)$ time points from $\mathrm{Unif}([0, T))$, where I is the number of observed events. All particles are evaluated using integrals that are estimated by evaluating the function at these time points (section 7.C.3).

The theoretical runtime complexity is $O(MI)$ because drawing a particle requires $O(I)$ time—the outer loop over time steps (line 11 of Algorithm 7.1)—and we draw M particles in total—the inner loop over particles (line 12 in Algorithm 7.1). Our GPU implementation (which we will release) parallelizes the inner loop over particles. We sample 50 particles in parallel in these experiments, but we have tested with 1000 particles in parallel as well. So this is not a real problem with a GPU.

We reported experiments that we performed to demonstrate the practicality. On average, drawing an ensemble of 50 particles takes 5 seconds per example on the synthetic datasets (average length 15 events), 12 seconds per example on the NYC Taxi dataset (average length 32 events) and 100 seconds per example on the Elevator System dataset (average length 313 events)—that is, 300-400 milliseconds per event.



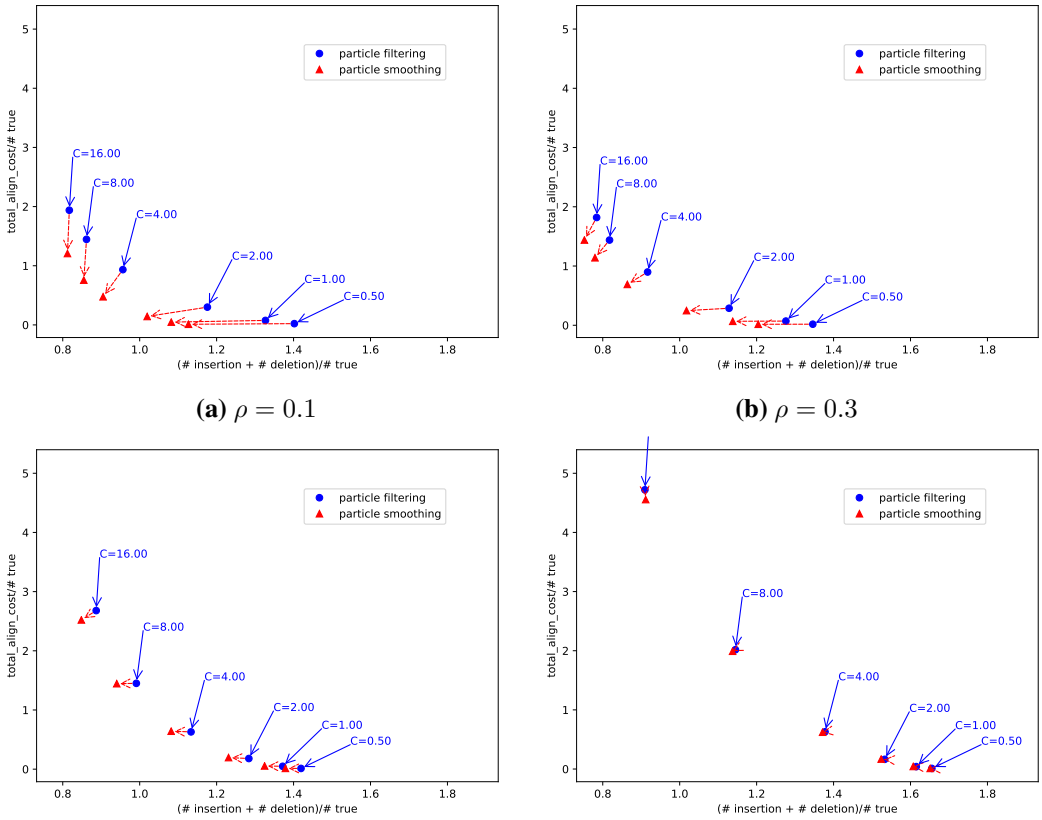
(a) Elevator System **(b)** NYC Taxi

Figure 7.5: Optimal transport distance results with a deterministic missingness mechanism. Again, the method works, with very similar qualitative behavior to Figure 7.3.

Such speeds are acceptable in many incomplete data applications, compared to the cost of collecting complete data—all the applications in section 7.1 involve real-time decision making at a human timescale.

7.H Monte Carlo EM

We normally assume (section 7.4.3) that some complete sequences are available for training the neural Hawkes process models. If incomplete sequences are also available, our particle smoothing method can be used to (approximately) impute the missing events, which yields additional complete sequences for training. Indeed, if we are willing to make a MAR assumption (Little and Rubin, 1987), then we can do imputation without modeling the missingness mechanism. Training on such imputed sequences is an instance of **Monte Carlo expectation-maximization (MCEM)** (Dempster, Laird, and Rubin, 1977; Wei and Tanner, 1990; McLachlan and Krishnan, 2007), with particle smoothing as the Monte Carlo E-step, and makes it possible to train with incomplete data only.

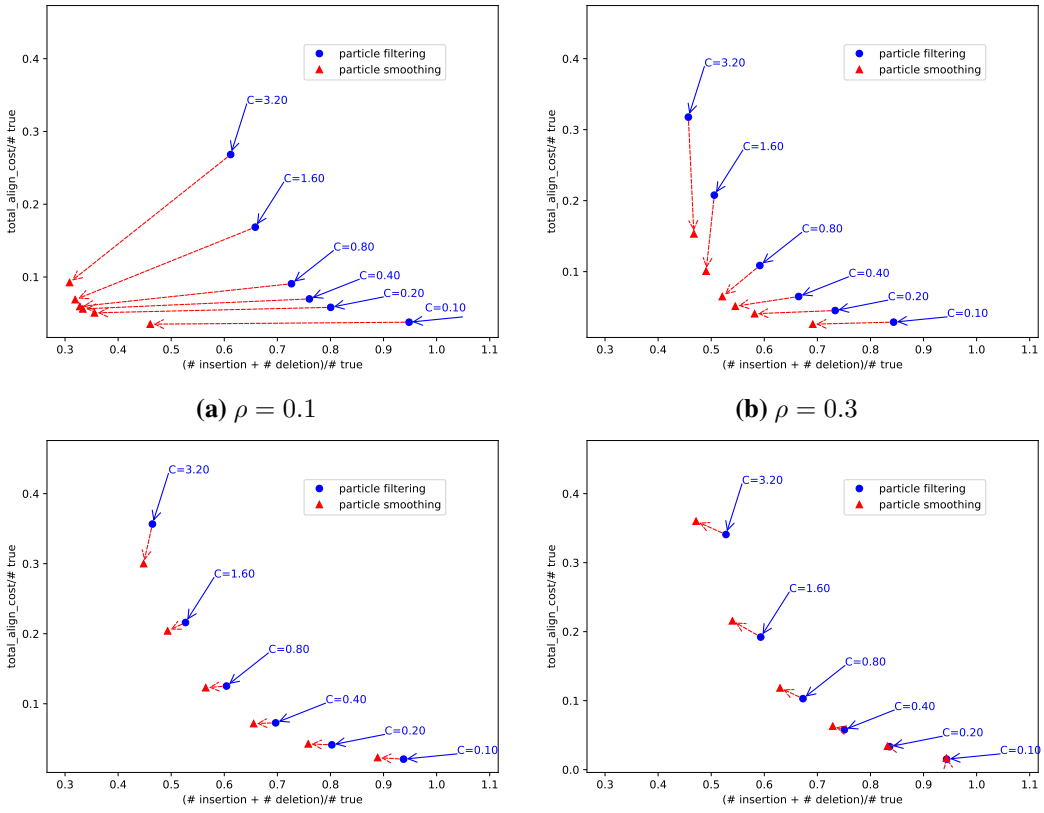


(a) $\rho = 0.1$ **(b)** $\rho = 0.3$

(c) $\rho = 0.7$ **(d)** $\rho = 0.9$

Figure 7.6: Optimal transport distance results with varying missingness rate ρ on the Elevator System dataset. As we can see, our particle smoothing consistently outperforms the filtering baseline with different ρ, although no clear trend with increasing ρ is found.

In the more general MNAR scenario, we can extend the E-step to consider the not-at-random missingness mechanism (see equation (7.3) below), but then we need both complete and incomplete sequences at training time in order to fit the parameters of the missingness mechanism (unless these parameters are already known) jointly with those of the neural Hawkes process. Although training with incomplete data is out of the scope of our experiments, we describe the methods here and provide MCEM pseudocode.



(a) $\rho = 0.1$ **(b)** $\rho = 0.3$

(c) $\rho = 0.7$ **(d)** $\rho = 0.9$

Figure 7.7: Optimal transport distance results with varying missingness rate ρ on the NYC Taxi dataset. Similar to Figure 7.6, our particle smoothing consistently outperforms the filtering baseline with different ρ, although no clear trend with increasing ρ is found.

In this case, we would like to know the (marginal) probability of the observed data $\mathbf{x}$ under the target distribution p:

$$p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{x} \sqcup \mathbf{z}) p_{\text{miss}}(\mathbf{z} \mid \mathbf{x} \sqcup \mathbf{z}) \tag{7.26}$$

If we propose $\mathbf{z}$ from $q(\mathbf{z} \mid \mathbf{x})$, then it can be rewritten as:

$$p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{x} \sqcup \mathbf{z}) p_{\text{miss}}(\mathbf{z} \mid \mathbf{x} \sqcup \mathbf{z}) \frac{q(\mathbf{z} \mid \mathbf{x})}{q(\mathbf{z} \mid \mathbf{x})} \tag{7.27a}$$

$$= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})}[\frac{p(\mathbf{x} \sqcup \mathbf{z}) p_{\text{miss}}(\mathbf{z} \mid \mathbf{x} \sqcup \mathbf{z})}{q(\mathbf{z} \mid \mathbf{x})}] \tag{7.27b}$$

Given a finite number M of proposed particles $\{\mathbf{z}_m\}_{m=1}^M$, this expectation can be estimated with empirical average:

$$p(\mathbf{x}) = \frac{1}{M} \sum_{m=1}^{M} \frac{p(\mathbf{x} \sqcup \mathbf{z}_m) p_{\text{miss}}(\mathbf{z}_m \mid \mathbf{x} \sqcup \mathbf{z}_m)}{q(\mathbf{z}_m \mid \mathbf{x})} \tag{7.28}$$

and it is obvious that

$$\log p(\mathbf{x}) \geq \frac{1}{M} \sum_{m=1}^{M} (b_m - \log q(\mathbf{z}_m \mid \mathbf{x})) \tag{7.29a}$$

$$b_m = \log p(\mathbf{x} \sqcup \mathbf{z}_m) + \log p_{\text{miss}}(\mathbf{z}_m \mid \mathbf{x} \sqcup \mathbf{z}_m) \tag{7.29b}$$

where the right-hand-side (RHS) term of equation (7.29a) is the **Evidence Lower Bound (ELBO)** that we would maximize in order to maximize the log-likelihood.

The MCEM algorithm is composed of two steps:

E(xpectation)-step We train the proposal distribution $q(\mathbf{z} \mid \mathbf{x})$ using the method in section 7.4.3 and then sample M weighted particles from $q(\mathbf{z} \mid \mathbf{x})$ by calling Algorithm 7.1.

M(aximization)-step We train the neural Hawkes process $p(\mathbf{x} \sqcup \mathbf{z})$ by maximizing the ELBO (equation (7.29a)).

Note that in the MAR case, $p_{\text{miss}}(\mathbf{z} \mid \mathbf{x} \sqcup \mathbf{z})$ is constant of $\mathbf{z}$ so the it can be omitted from the formulation (and thus the algorithms). Also note that, for particle filtering, the proposal distribution $q(\mathbf{z} \mid \mathbf{x})$ is only part of $p(\mathbf{x} \sqcup \mathbf{z})$ so we do not need to train $q(\mathbf{z} \mid \mathbf{x})$ at the E-step.

Maximum-likelihood estimation remains sensible in the MNAR case provided that we know one of the distributions p or p_{miss}, in which case we can use EM to estimate

the other distribution.

(1) If p_{miss} is known and fixed, as in our experiments, this gives a minor variant of ordinary EM. Ordinary EM makes the MAR assumption that the p_{miss} factor of equation (7.1) can be ignored. However, if we know p_{miss}, we can incorporate it rather than ignoring it; then it need not satisfy the MAR assumption.

(2) Conversely, if p is known and fixed because we estimated it from a sufficient quantity of *complete* data, then we can use incomplete data to learn the MNAR missingness distribution p_{miss}. This setting would even lets us learn contextual missingness mechanisms in which the probability that an event is censored depends not only on the event itself, but also on the surrounding events and whether they are censored. For example, one could try to fit p_{miss} with an LSTM model or a BiLSTM-CRF model (Huang, Xu, and Yu, 2015) that performs structured joint prediction of the missingness of all events in the sequence. Extending that method to use continuous-time LSTMs would allow it to take timing into account.

The E step of Monte Carlo EM uses the current guesses of p and/or p_{miss} to sample from the posterior distribution $p(\mathbf{z} \mid \mathbf{x})$ of the missing values. That posterior is uncontroversially defined by the simple Bayesian formula (7.1). Notice that even if p and p_{miss} were *both* unknown, we could still run MCEM to locally maximize the likelihood $p(\mathbf{x})$, but unfortunately the parameters would be unidentifiable in this case. Thus, there would be many missing-data models with the same likelihood, as explained in section 7.A, and they would make different predictions of $\mathbf{z}$.

Algorithm 7.2 Dynamic Programming to Find Optimal Transport Distance

Input: proposal $\hat{\mathbf{z}}$; reference $\mathbf{z}^*$
Output: optimal transport distance d; alignment $\mathbf{a}$

procedure OTD($\hat{\mathbf{z}}, \mathbf{z}^*$)
  $d \leftarrow 0$; $\mathbf{a} \leftarrow$ empty collection $\{\}$
  for $k \leftarrow 1$ **to** K :
    $d^{(k)}, \mathbf{a}^{(k)} \leftarrow \text{ALIGN}(\hat{\mathbf{z}}^{(k)}, \mathbf{z}^{*(k)})$
    $d \leftarrow d + d^{(k)}$; $\mathbf{a} \leftarrow \mathbf{a} \cup \mathbf{a}^{(k)}$
  return $d, \mathbf{a}$
procedure ALIGN($\hat{\mathbf{z}}^{(k)}, \mathbf{z}^{*(k)}$)
  $\hat{I} \leftarrow |\hat{\mathbf{z}}^{(k)}|$; $I^* \leftarrow |\mathbf{z}^{*(k)}|$ ▷ $\hat{\mathbf{z}}^{(k)} = \hat{t}_1, \ldots, \hat{t}_{\hat{I}}$ *and* $\mathbf{z}^{*(k)} = t^*_1, \ldots, t^*_{I^*}$
  $\mathbf{D} \leftarrow$ zero matrix with $(\hat{I}+1)$ rows and (I^*+1) columns
  $\mathbf{P} \leftarrow$ empty matrix with $\hat{I}$ rows and I^* columns ▷ *back pointers*
  for $\hat{i} \leftarrow 1$ **to** $\hat{I}$: ▷ *transport reference of length* 0 *to proposal of length* $\hat{i}$
    $\mathbf{D}_{\hat{i},0} \leftarrow \mathbf{D}_{\hat{i}-1,0} + C_{\text{delete}}$ ▷ *delete* $\hat{t}_{\hat{i}}$ *(and prefixes are matched)*
  for $i^* \leftarrow 1$ **to** I^* : ▷ *transport preference of length* i^* *to proposal of length* 0
    ▷ *insert* $\hat{t}_{i^*} = t^*_{i^*}$ *to decode (and their prefixes are matched)*
    $\mathbf{D}_{0,i^*} \leftarrow \mathbf{D}_{0,i^*} + C_{\text{insert}}$
  for $\hat{i} \leftarrow 1$ **to** $\hat{I}$: ▷ *proposal prefix of length* $\hat{i}$
    for $i^* \leftarrow 1$ **to** I^* : ▷ *to match reference of length* i^*
      $D_{\text{delete}} \leftarrow \mathbf{D}_{\hat{i}-1,i^*} + C_{\text{delete}}$ ▷ *if the event token at* $\hat{t}_{\hat{i}}$ *is deleted from* $\hat{\mathbf{z}}^{(k)}$
      $D_{\text{insert}} \leftarrow \mathbf{D}_{\hat{i},i^*-1} + C_{\text{insert}}$ ▷ *if an event token at* $t^*_{i^*}$ *is inserted to* $\hat{\mathbf{z}}^{(k)}$
      $\Delta \leftarrow |\hat{t}_{\hat{i}} - t^*_{i^*}|$ ▷ *distance between event at* $t^*_{i^*}$ *of* $\mathbf{z}^{*(k)}$ *and event at* $\hat{t}_{\hat{i}}$ *of* $\hat{\mathbf{z}}^{(k)}$
      $D_{\text{move}} \leftarrow \mathbf{D}_{\hat{i}-1,i^*-1} + \Delta$ ▷ *if these events are aligned*
      $\mathbf{D}_{\hat{i},i^*} \leftarrow \min\{D_{\text{insert}}, D_{\text{delete}}, D_{\text{move}}\}$ ▷ *choose the shortest distance*
      $\mathbf{P}_{\hat{i},i^*} \leftarrow \arg\min_{e \in \{\text{insert,delete,move}\}} D_e$ ▷ *and the edit which yields that distance*
  $\hat{i} \leftarrow \hat{I}$; $i^* \leftarrow I^*$; $\mathbf{a} \leftarrow$ empty collection$\{\}$
  while $\hat{i} > 0$ **and** $i^* > 0$: ▷ *back trace*
    if $\mathbf{P}_{\hat{i},i^*} =$ delete : $\hat{i} \leftarrow \hat{i} - 1$ ▷ *token* $\hat{t}_{\hat{i}}$ *is deleted.*
    if $\mathbf{P}_{\hat{i},i^*} =$ insert : $i^* \leftarrow i^* - 1$ ▷ *a token at* $t^*_{i^*}$ *is inserted*
    if $\mathbf{P}_{\hat{i},i^*} =$ move : ▷ *token* $t^*_{i^*}$ *is aligned to* $\hat{t}_{\hat{i}}$
      $\hat{i} \leftarrow \hat{i} - 1$; $i^* \leftarrow i^* - 1$
      $\mathbf{a} \leftarrow \mathbf{a} \cup \{(\hat{t}_{\hat{i}}, t^*_{i^*})\}$
  return $\mathbf{D}_{\hat{I},I^*}, \mathbf{a}$

Algorithm 7.3 Approximate Consensus Decoding

Input: collection of weighted particles $\mathcal{Z}_M = \{(\mathbf{z}_m, w_m)\}_{m=1}^M$
Output: consensus sequence $\hat{\mathbf{z}}$ with low $\sum_{m=1}^M w_m L(\hat{\mathbf{z}}, \mathbf{z}_m)$

procedure APPROXMBR($\mathcal{Z}_M$)
  $\hat{\mathbf{z}} \leftarrow$ empty sequence
  for $k = 1$ **to** K :
    ▷ *decode for type-k by calling* DECODEK
    $\hat{\mathbf{z}}^{(k)} \leftarrow \text{DECODEK}(\{(\mathbf{z}_m^{(k)}, w_m)\}_{m=1}^M)$; $\hat{\mathbf{z}} \leftarrow \hat{\mathbf{z}} \sqcup \hat{\mathbf{z}}^{(k)}$
  return $\hat{\mathbf{z}}$
procedure DECODEK($\mathcal{Z}_M$)
  ▷ *$\mathcal{Z}_M$ actually means $\mathcal{Z}_M^{(k)} = \{(\mathbf{z}_m^{(k)}, w_m)\}_{m=1}^M$ throughout the procedure*
  ▷ *$\mathbf{z}_m$ is constant*
  ▷ *init decode as highest weighted particle and it is a global variable*
  $\mathbf{z} \leftarrow \arg\max_{\mathbf{z} \in \{\mathbf{z}_m\}_{m=1}^M} w_m$
  repeat
    for $m = 1$ **to** M : ▷ **Align Phase**
      $d_m, \mathbf{a}_m \leftarrow \text{ALIGN}(\mathbf{z}, \mathbf{z}_m)$ ▷ *call method in Algorithm 7.2; $d_m, \mathbf{a}_m$ are global*
    $r_{\text{min}} \leftarrow \sum_m w_m d_m$ ▷ *track the risk of current* $\mathbf{z}$
    $\mathbf{z}, \{d_m, \mathbf{a}_m\}_{m=1}^M \leftarrow \text{MOVE}(\mathbf{z}, \{\mathbf{z}_m, d_m, \mathbf{a}_m\}_{m=1}^M)$ ▷ *see Algorithm 7.3*
    $\mathbf{z}, \{d_m, \mathbf{a}_m\}_{m=1}^M \leftarrow \text{DELETE}(\mathbf{z}, \{\mathbf{z}_m, d_m, \mathbf{a}_m\}_{m=1}^M)$ ▷ *see Algorithm 7.3*
    $\mathbf{z}, \{d_m, \mathbf{a}_m\}_{m=1}^M \leftarrow \text{INSERT}(\mathbf{z}, \{\mathbf{z}_m, d_m, \mathbf{a}_m\}_{m=1}^M)$ ▷ *see Algorithm 7.3*
  until $\sum_{m=1}^M w_m d_m = r_{\text{min}}$ ▷ *risk stops decreasing*
  return $\mathbf{z}$
procedure MOVE($\mathbf{z}, \{\mathbf{z}_m, d_m, \mathbf{a}_m\}_{m=1}^M$) ▷ **Move Phase**
  for t **in** $\mathbf{z}$:
    for $t' \in \{t' : (t', t) \in \bigcup_{m=1}^M \mathbf{a}_m\}$: ▷ *may replace t with t' which is aligned to t*
      $(\forall m) d'_m \leftarrow d_m$
      for $(t'', m) \in \{(t'', m) : (t'', t) \in \mathbf{a}_m, m \in \{1, \ldots, M\}\}$:
        $d'_m \leftarrow d'_m - |t'' - t| + |t'' - t'|$
      if $\sum_m w_m d'_m < \sum_m w_m d_m$:
        $(\forall m) d_m \leftarrow d'_m$; $t \leftarrow t'$ ▷ *t move to t' for lower risk*
  return $\mathbf{z}, \{d_m, \mathbf{a}_m\}_{m=1}^M$

Algorithm 7.3 (Continued) Approximate Consensus Decoding

procedure DELETE($\mathbf{z}, \{\mathbf{z}_m, d_m, \mathbf{a}_m\}_{m=1}^M$) ▷ **Delete Phase**
  for t **in** $\mathbf{z}$: ▷ *may delete this event*
    for $m = 1$ **to** M : ▷ *update each* d_m
      if $\exists t' \in \mathbf{z}_m$ **and** $(t', t) \in \mathbf{a}_m$: ▷ *find the only, if any,* $t' \in \mathbf{z}_m$ *that is aligned to* t
        ▷ *if we delete* t *and its alignment* (t', t), d_m *decreases by the alignment cost*
        ▷ *(because we do not need to align it)*
        ▷ *but increases by an insertion cost*
        ▷ *(because we need to insert an event at* t *to match* $\mathbf{z}_m$*)*
        $d'_m \leftarrow d_m + C_{\text{insert}} - |t' - t|$
      else ▷ *otherwise, this event has been deleted when matching with* $\mathbf{z}_m$
        ▷ *if we do not have this event at* t *in* $\mathbf{z}$
        $d'_m \leftarrow d_m - C_{\text{delete}}$ ▷ *no need to pay deletion cost when matching with* $\mathbf{z}_m$
      if $\sum_m w_m d'_m < \sum_m w_m d_m$:
        delete t from $\mathbf{z}$; $(\forall m)$ delete (t', t) from $\mathbf{a}_m$; $d_m \leftarrow d'_m$
  return $\mathbf{z}, \{d_m, \mathbf{a}_m\}_{m=1}^M$
procedure INSERT($\mathbf{z}, \{\mathbf{z}_m, d_m, \mathbf{a}_m\}_{m=1}^M$) ▷ **Insert Phase**
  repeat
    $t \leftarrow$ None, $\Delta \leftarrow -\infty$
    for $t_c \in \bigsqcup_m \mathbf{z}_m$ such that $t_c \notin \mathbf{z}$: ▷ *may insert* t_c *if it is not in* $\mathbf{z}$ *yet*
      for $m = 1$ **to** M :
        ▷ *find* t' *in* $\mathbf{z}_m$ *that is not aligned yet*
        $\mathbf{z}'_m \leftarrow \{t' : \forall t'', (t'', t') \notin \mathbf{a}_m \text{ and } t' \in \mathbf{z}_m\}$
        if $\mathbf{z}'$ is not empty **and** $\min_{t' \in \mathbf{z}'_m} |t' - t_c| < C_{\text{insert}} + C_{\text{delete}}$:
          ▷ *if there is any that is close enough to* t_c
          ▷ *align the closest one to* t_c
          $d'_m \leftarrow d_m - C_{\text{insert}} + \min_{t' \in \mathbf{z}'_m} |t' - t_c|$; $\mathbf{a}'_m \leftarrow \mathbf{a}_m \cup \{(t_c, t')\}$
        else
          $d'_m \leftarrow d_m + C_{\text{delete}}$; $\mathbf{a}'_m \leftarrow \mathbf{a}_m$
      if $\sum_m w_m d_m - \sum_m w_m d'_m > \Delta : t \leftarrow t_c$; $\Delta \leftarrow \sum_m w_m d_m - \sum_m w_m d'_m$
    if $\Delta > 0 : \mathbf{z} \leftarrow \mathbf{z} \sqcup \{t\}$; $(\forall m) \mathbf{a}_m \leftarrow \mathbf{a}'_m$; $d_m \leftarrow d'_m$
  until $\Delta \leq 0$
  return $\mathbf{z}, \{d_m, \mathbf{a}_m\}_{m=1}^M$

Chapter 8

Conclusion and Future Work

We have presented an arsenal of technical tools for probabilistic modeling of event sequences (Chapter 2). In particular, they are:

- **Neural Hawkes process** (Chapter 3; Mei and Eisner (2017)), a purely neural model that can capture complex patterns in real-world sequences, and empirically outperformed strong competitors on several tasks and multiple datasets.
- **Neural Datalog through time** (Chapter 4; Mei et al. (2020)), a neural-symbolic generalization of the neural Hawkes process, whose architecture is determined by a temporal logic program written in the light-weight modeling language that we designed. Empirically, it improved prediction by encoding appropriate domain knowledge in its architecture.
- **Attentive models** (Chapter 5, Mei, Yang, and Eisner (2020)), in particular, Transformer-based variants of the neural Hawkes process and neural Datalog through time.
- **Noise-contrastive estimation** (Chapter 6; Mei, Wan, and Eisner (2020)), which is able to estimate the parameters of point process models with low computational cost, yet still allowing them to achieve the same level of likelihood as if

they were trained by maximum likelihood estimation.

- **Particle smoothing** (Chapter 7; Mei, Qin, and Eisner (2019)), a general sequential importance sampling method that trains a smart proposal distribution to impute missing events in an given incomplete event sequence.

These technical innovations have significantly advanced the state-of-the-art in the field of event sequence modeling; they are my main contributions.

The methods we introduced in Chapters 4–7 are very general (although their exposition and experimentation heavily relied upon the neural Hawkes process): our neural Datalog through time framework can handle discrete-time sequences as well; our noise-contrastive estimation method can be used to train any autoregressive point process over time; our particle smoothing method can work with any kind of point process that has been trained on complete data.

Several avenues for further work have been discussed in section 3.C and section 4.A. In this chapter, I elaborate on a few of them.

8.1 Continuous-Time Reinforcement Learning

Continuous-time reinforcement learning (RL) is an emerging area (Upadhyay, De, and Rodriguez, 2018; Alt, Schultheis, and Koeppl, 2020). In this setting, an RL agent is trained to insert and suppress certain actions at certain times, bearing the goal of improving the future course of events. Such an agent would be useful in various applied domains, including

Medicine An intelligent medical assistant might be able to suggest certain measurements, treatments, and schedule of future visits for a given patient, which would help improve the patient's future health condition.

Online shopping An intelligent shopping assistant could learn to present certain offers and recommendations at certain times, which might help improve long-term customer satisfaction, including engagement and transaction speed.

Our family of models could be used as the environment model in such a reinforcement learner. For any action proposed at any time, it predicts what would happen afterwards if that action was actually taken at that time; such predictions might help improve the agent's policy, leading to a course of future events that yield higher reward. This is a general benefit of deploying an environment model in reinforcement learning (Sutton and Barto, 2018).

Challenges arise in developing *continuous-time* reinforcement learning methods as well as applying them to real-world applications. In sections 8.1.1–8.1.3, I discuss a few challenges that I have identified, and propose my ideas for addressing them.

8.1.1 Challenge I: Formalism

Conventional RL methods are built upon the formalism of MDP. This is not applicable in this new setting, since we have to consider timing of actions and observables. Therefore, I propose a new formalism where

- the agent generates actions, which are discrete events that may occur stochastically in continuous time;
- the environment generates observables, which are discrete events in continuous time as well;
- at any time, there may be an action being taken, an observable being generated, or nothing happening.

Under this formalism, actions and observables need not alternate strictly, and the time

intervals between events are irregular and stochastic. The RL agent learns to select actions—with their timings—in order to maximize future reward; a scheduled action might be preempted by an observable, and vice versa. Such new formalism would lead to a new set of methodologies, including (say) new versions of Bellman equations and thus new (value-based) learning methods.

This new formalism is a generalization of semi-Markov decision process. The semi-Markov decision process (semi-MDP or SMDP) is an extension of MDP that handles *continuous-time* dynamics (Puterman, 2014). It generalizes MDP by allowing the time spent in a particular state to follow an arbitrary probability distribution.[1] The main restriction of this formalism is that the RL agent is *passive*: it is *allowed* and *required* to take actions *only* at the decision epochs, i.e., when the state transitions happen.[2] However, under my formalism, the RL agent is allowed to decide the timings of its actions, and possibly preempt a previously scheduled state transition. For example, in a conference meeting, Hongyuan (the agent) may want to pick the right time to tell a smart and relevant joke, before other participants (the environment) change the topic (a state transition); although the topic change might have already been planned in their minds (a previously scheduled state transition), Hongyuan's joke may still direct them to a new topic, thus having the planned one discarded (a preemption). However, under the semi-MDP formalism, Hongyuan is only allowed to say things when he is asked—how frustrating that would be![3]

[1] The continuous-time Markov chain is a special case of semi-MDP in which the time between transitions is exponentially distributed.

[2] This is different from the "epoch" concept in supervised learning, which means a complete pass through the training data.

[3] Technically, one could move the time distribution from the environment dynamics into the policy function, enabling RL agents to learn to choose the timings of state transitions. This is the approach taken by Hua et al. (2021) and Du, Futoma, and Doshi-Velez (2020). But such design is still not applicable to more complicated and realistic scenarios, in which all actions and observables are events that may stochastically occur at any time, and the agent and environment each control some of them

Some readers may wonder why we don't use the formulation of continuous-time control problems. Control problems are primarily concerned with the scenarios in which actions form a continuous flow over time (e.g., pressing the accelerator pedal in a car), while I am now interested in the applications where actions form a sequence of time-stamped discrete events (e.g., a series of marketing calls at different times). Moreover, in control problems, actions are in continuous spaces (e.g., the pedal-pressing force is taken from $[0, \infty)$); in the applications of my interest, actions are all discrete (e.g., advertising iPad or Kindle at this particular call).

8.1.2 Challenge II: Scarcity of Real-World Interactions

In many real-world scenarios (e.g., medical and education), real interactions are expensive or prohibitive, such that we have to rely on observational interventions; this is known as **offline reinforcement learning** (Levine et al., 2020). Environment models learned in such settings often suffer from **distribution shift** due to lack of real interventions. The model errors would lead to the **model exploitation** problem: policy learning processes may intentionally produce rare states and actions that are erroneously predicted to have higher reward than they actually have under the real dynamics.

Existing solutions to these problems include constraining the deviation of learned models and policies from training data (Berkenkamp et al., 2017; Rhinehart, McAllister, and Levine, 2018) and making conservative value estimation for states and actions whose successor states can not be predicted with high certainty under the model (Kidambi et al., 2020; Yu et al., 2020). These methods are all developed and analyzed under MDP formalism. It will be interesting to explore them—for both

with their timings included.

their theoretical guarantees and practical effectiveness—under the semi-MDP and generalized semi-MDP formalism.

8.1.3 Challenge III: Interpretability and Reliability

Real-world applications sometimes require high level of interpretability and reliability from machine learning methods. For example, an intelligent medical assistant may have to explain why it has made certain clinical predictions and suggestions so that human doctors can judge their correctness and credibility. This may need human in the loop: e.g., a human doctor may use our NDTT framework (see Chapter 4) to incorporate their domain knowledge and safety constraints into the model architecture, thus making the model outputs more interpretable and reliable; the human doctor then may further revise the NDTT program after analyzing the results.

8.2 Higher Model Capacity and More Complex Data

Another interesting and important direction is to extend the capacity of my methods to handle a wider variety of data. In this section, I will document some specific ideas.

8.2.1 Cross-Domain Pre-Training

Inspired by the success of large pre-trained language models such as BERT (Devlin et al., 2018) and GPT-3 (Brown et al., 2020), I plan to explore the usefulness of a large neural Hawkes process pre-trained across various domains: can it perform well on new tasks with few shot learning? The practical challenge is that event type identifiers, unlike natural language characters and words, are hardly shared across domains. My idea is, at least for some domains (e.g., online social media), to leverage the textual descriptions: precisely, we may only have one event type but event tokens are discriminated by natural language sentences describing them. My hope is that

the word embeddings learn to encode information that is useful to estimate event probabilities across domains.

8.2.2 Latent Events

In many real-world applications, we can't observe the actual interactions but only perceive the signals generated by them. For example, we don't see an actual company acquisition but read new stories about it; we don't observe key presses but hear audio created by them. A model for such data should have latent variables: it can use my neural Hawkes process for the latent events and any appropriate emission model for the observed data. Training such a model requires Monte Carlo Expectation Maximization (MCEM): in the E-step, latent events are proposed conditioned on the observed data; in the M-step, the complete-data log-likelihood is maximized. Note that proposing an event at any time would consult both the past and future observations: this needs my continuous-time particle smoother.

8.2.3 Language-to-Datalog Parsers

In many domains, it is costly to write knowledge down as Datalog programs because knowledge is only stored in unstructured formats (e.g., texts) or it is not known at all (e.g., map of an unexplored maze). The former case raises the research question about building a semantic parser that automatically converts unstructured texts to Datalog facts and rules. This task is very challenging and there is little supervised data. One idea is to first leverage the existing tools to parse texts into structured forms like neo-Davidsonian logic forms (Parsons, 1990) or abstract meaning representation (AMR) (Banarescu et al., 2013), and then convert these forms to Datalog syntax by manually defined rules. For the latter case, one has to learn the Datalog program along with other things, just like simultaneous localization and mapping (SLAM) algorithms

that construct a map of an unknown environment while simultaneously keeping track of an agent's location within it (Thrun, Burgard, and Fox, 2005). In either case, NDTT needs to be extended to allow an unbounded number of atoms: a couple of possible ways were documented in the appendices of Chapter 4.

A natural testbed for such a parser would be the interactive fiction (IF) environment: the agent uses text commands to control a character to take actions and interact with the environment; the environment produces natural language feedback that describes what happens next and then waits for additional input. Here is a small transcript from Zork, one of the earliest interactive fictions:

```
West of House
This is an open field west of a white house, with a boarded front
  door. There is a small mailbox here.
> OPEN MAILBOX
You open the mailbox, revealing a small leaflet.
> GET LEAFLET
Taken.
> READ LEAFLET
WELCOME TO ZORK

  ⋮
This is a forest, with trees in all directions. To the east,
  there appears to be sunlight.
> GO EAST
Forest Path
... One particularly large tree with some low branches stands at
  the edge of the path ...
> CLIMB TREE
Up a tree

  ⋮
```

As we can see, knowledge about the world is either expressed in natural language texts (e.g., objects and their spatial relations) or needs to be discovered by navigating

through the maze (e.g., map). Therefore, we need to learn a NDTT program by (say) parsing texts and exploring the world, as discussed in future plans.

How can we know what actions should be taken on each object? For example, how can we know a "small leaflet" is something we should take and that a "large tree" is something we can climb? Since the agent is trained by reinforcement learning, a naive way is just to wait for the agent to pick up the right actions for each object by many rounds of try-and-trial. But that would be notoriously sample-inefficient: the agent may try many brainless actions like "jump onto the leaflet" before getting it. An alternative is to leverage a pretrained language model that has learned (say) the statistical correlation between "get" and "leaflet". The language model may also have learned common senses like "a troll is dangerous" and "a sword is a weapon" and thus can help increase the probability that the agent "kill the troll with sword" in the Troll Room. Other ways to improve the agent may include:

- building an accurate (perhaps neural-symbolic) environment model for the agent to do explicit planning;
- pulling the agent out of the local optima of repeatedly collecting small rewards but not moving the game forward.

The End

Now we have come to the end of this thesis. I wish that I still had more time to document more of my ideas as well as to improve the writing of existing content. But maybe it is true that "a good thesis is a done thesis; a great thesis is a published thesis; a perfect thesis is neither." So I will call it over here, and move on in life.

References

Acar, Umut A and Ruy Ley-Wild (2008). "Self-adjusting computation with Delta ML". In: *International School on Advanced Functional Programming*.

Aldous, David, Illdar Ibragimov, Jean Jacod, and David Aldous (1985). "Exchangeability and related topics". In: *École d'Été de Probabilités de Saint-Flour XIII — 1983*. Lecture Notes in Mathematics.

Alt, Bastian, Matthias Schultheis, and Heinz Koeppl (2020). "POMDPs in Continuous Time and Discrete Spaces". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Andreas, Jacob, Marcus Rohrbach, Trevor Darrell, and Dan Klein (2016). "Learning to compose neural networks for question answering". In: *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics – Human Language Technologies (NAACL HLT)*.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2015). "Neural Machine Translation by Jointly Learning to Align and Translate". In: *Proceedings of the International Conference on Learning Representations (ICLR)*.

Banarescu, Laura, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider (2013). "Abstract meaning representation for sembanking". In: *Proceedings of the 7th linguistic annotation workshop and interoperability with discourse*.

Bao, G., C. G. Cassandras, T. E. Djaferis, A. D. Gandhi, and D. P. Looze (1994). *Elevators Dispatchers for Down-Peak Traffic*. Technical Report.

Baran, Ilya, Erik D. Demaine, and Dimitriy A. Katz (2008). "Optimally Adaptive Integration of Univariate Lipschitz Functions". In: *Algorithmica*.

Bárány, Vince, Balder ten Cate, Benny Kimelfeld, Dan Olteanu, and Zografoula Vagena (2017). "Declarative Probabilistic Programming with Datalog". In: *ACM Transactions on Database Systems* 42.4, 22:1–35.

Benamou, Jean-David (2003). "Numerical Resolution of an "Unbalanced" Mass Transport Problem". In: *ESAIM: Mathematical Modelling and Numerical Analysis*.

Berkenkamp, Felix, Matteo Turchetta, Angela P Schoellig, and Andreas Krause (2017). "Safe model-based reinforcement learning with stability guarantees". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Bhattacharjya, D., D. Subramanian, and T. Gao (2018). "Proximal graphical event models". In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 8136–8145.

Blei, D. and J. Lafferty (2006). "Correlated Topic Models". In: *Advances in Neural Information Processing Systems*. Vol. 18, p. 147.

Blei, David M. and Peter I. Frazier (2010). "Distance-Dependent Chinese Restaurant Processes". In: *Proc. of ICML*, pp. 87–94.

Boyd, Alex, Robert Bamler, Stephan Mandt, and Padhraic Smyth (2020). "User-Dependent Neural Sequence Models for Continuous-Time Event Data". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Brown, Tom B, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. (2020). "Language models are few-shot learners". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Carbonell, Pierre, `jcdouet`, Henrique Carvalho Alves, and Andras Tim (2016). *pyDatalog*. https://pypi.org/project/pyDatalog/.

Ceri, Stefano, Georg Gottlob, and Letizia Tanca (1989). "What you always wanted to know about Datalog (and never dared to ask)". In: *IEEE transactions on knowledge and data engineering*.

Chelba, Ciprian, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson (2013). "One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling". In: *Computing Research Repository*.

Chen, David L. and Raymond J. Mooney (2008). "Learning to Sportscast: A Test of Grounded Language Acquisition". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Chen, Ricky TQ, Brandon Amos, and Maximilian Nickel (2021). "Learning neural event functions for ordinary differential equations". In: *Proceedings of the International Conference on Learning Representations (ICLR)*.

Chen, Ricky TQ, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud (2018). "Neural ordinary differential equations". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Chizat, Lenaic, Gabriel Peyré, Bernhard Schmitzer, and François-Xavier Vialard (2015). "Unbalanced Optimal Transport: Geometry and Kantorovich Formulation". In: *arXiv preprint arXiv:1508.05216*.

Chizat, Lenaic, Gabriel Peyré, Bernhard Schmitzer, and François-Xavier Vialard (2018). "An Interpolating Distance Between Optimal Transport and Fisher-Rao Metrics". In: *Foundations of Computational Mathematics*.

Choi, Edward, Nan Du, Robert Chen, Le Song, and Jimeng Sun (2015). "Constructing Disease Network and Temporal Progression Model via Context-Sensitive Hawkes Process". In: *Data Mining (ICDM), 2015 IEEE International Conference on*.

Chopin, Nicolas and Sumeetpal S. Singh (2015). "On particle Gibbs sampling". In: *Bernoulli*.

Crites, Robert H. and Andrew G. Barto (1996). "Improving Elevator Performance Using Reinforcement Learning". In: *Advances in Neural Information Processing Systems*.

Dempster, Arthur P., Nan M. Laird, and Donald B. Rubin (1977). "Maximum Likelihood from Incomplete Data via the EM Algorithm". In: *Journal of the Royal Statistical Society. Series B (Methodological)*.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova (2018). "BERT: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805*.

Doucet, Arnaud, Simon Godsill, and Christophe Andrieu (2000). "On Sequential Monte Carlo Sampling Methods for Bayesian Filtering". In: *Statistics and Computing*.

Doucet, Arnaud and Adam M. Johansen (2009). "A Tutorial on Particle Filtering and Smoothing: Fifteen Years Later". In: *Handbook of Nonlinear Filtering*.

Du, Jianzhun, Joseph Futoma, and Finale Doshi-Velez (2020). "Model-based reinforcement learning for semi-Markov decision processes with neural ODEs". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Du, Nan, Hanjun Dai, Rakshit Trivedi, Utkarsh Upadhyay, Manuel Gomez-Rodriguez, and Le Song (2016). "Recurrent Marked Temporal Point Processes: Embedding Event History to Vector". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Du, Nan, Mehrdad Farajtabar, Amr Ahmed, Alexander J. Smola, and Le Song (2015a). "Dirichlet-Hawkes Processes with Applications to Clustering Continuous-Time Document Streams". In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Du, Nan, Yichen Wang, Niao He, Jimeng Sun, and Le Song (2015b). "Time-Sensitive Recommendation from Recurrent User Activities". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Dyer, C., A. Kuncoro, M. Ballesteros, and Noah A. Smith (2016). "Recurrent neural network grammars". In: *NAACL*, pp. 199–209.

Eichler, Michael, Rainer Dahlhaus, and Johannes Dueck (2017). "Graphical Modeling for Multivariate Hawkes Processes with Nonparametric Link Functions". In: *Journal of Time Series Analysis*.

Elman, Jeffrey L (1990). "Finding structure in time". In: *Cognitive science*.

Etesami, Jalal, Negar Kiyavash, Kun Zhang, and Kushagra Singhal (2016). "Learning Network of Multivariate Hawkes Processes: A Time Series Approach". In: *arXiv preprint arXiv:1603.04319*.

F. Zaidan, Omar and Jason Eisner (2008). "Modeling Annotators: A Generative Approach to Learning from Annotator Rationales". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Fan, Yu, Jing Xu, and Christian R. Shelton (2010). "Importance Sampling for Continuous-Time Bayesian Networks". In: *Journal of Machine Learning Research*.

Feng, Da-Fei and Russell F. Doolittle (1987). "Progressive Sequence Alignment as a Prerequisite to Correct Phylogenetic Trees". In: *Journal of Molecular Evolution*.

Ferguson, Thomas S (1996). *A Course in Large Sample Theory*. Chapman and Hall.

Filardo, Nathaniel Wesley and Jason Eisner (2012). "A flexible solver for finite arithmetic circuits". In: *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*.

Fisher, R. A., A. S. Corbet, and C.B. Williams (1943). "The Relation Between the Number of Species and the Number of Individuals in a Random Sample of an Animal Population". In: *J. Animal Ecology* 12, pp. 42–58.

Frogner, Charlie, Chiyuan Zhang, Hossein Mobahi, Mauricio Araya, and Tomaso A. Poggio (2015). "Learning with a Wasserstein Loss". In: *Advances in Neural Information Processing Systems*.

Getoor, Lise and Ben Taskar, eds. (2007). *Introduction to Statistical Relational Learning*. MIT Press.

Goller, Christoph and Andreas Kuchler (1996). "Learning Task-Dependent Distributed Representations by Backpropagation Through Structure". In: *IEEE International Conference on Neural Networks*. Vol. 1, pp. 347–352.

Gomez Rodriguez, Manuel, Jure Leskovec, and Bernhard Schölkopf (2013). "Structure and Dynamics of Information Pathways in Online Media". In: *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*.

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio (2014). "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Gotoh, Osamu (1996). "Significant Improvement in Accuracy of Multiple Protein Sequence Alignments by Iterative Refinement as Assessed by Reference to Structural Alignments". In: *Journal of Molecular Biology*.

Graves, Alex (2012). *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer.

Graves, Alex, Navdeep Jaitly, and Abdel-Rahman Mohamed (2013). "Hybrid Speech Recognition with Deep Bidirectional LSTM". In: *IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*.

Graves, Alex, Greg Wayne, and Ivo Danihelka (2014). "Neural Turing Machines". In: *arXiv preprint arXiv:1410.5401*.

Graves, Alex, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. (2016). "Hybrid Computing Using a Neural Network with Dynamic External Memory". In: *Nature*.

Guo, Fangjian, Charles Blundell, Hanna Wallach, and Katherine Heller (2015). "The Bayesian Echo Chamber: Modeling Social Influence via Linguistic Accommodation". In: *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*.

Guo, Ruocheng, Jundong Li, and Huan Liu (2018). "INITIATOR: Noise-Contrastive Estimation for Marked Temporal Point Process". In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Gusfield, Dan (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*.

Gutmann, Michael and Aapo Hyvärinen (2010). "Noise-contrastive estimation: A new estimation principle for unnormalized statistical models". In:

Gutmann, Michael U. and Aapo Hyvärinen (2012). "Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics". In:

Hamilton, Will, Zhitao Ying, and Jure Leskovec (2017a). "Inductive representation learning on large graphs". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Hamilton, William L, Rex Ying, and Jure Leskovec (2017b). "Representation learning on graphs: Methods and applications". In: *arXiv preprint arXiv:1709.05584*.

Hammer, Matthew Arthur (2012). "Self-Adjusting Machines". PhD thesis. Computer Science Department, University of Chicago.

Hawkes, Alan G. (1971). "Spectra of Some Self-Exciting and Mutually Exciting Point Processes". In: *Biometrika*.

He, Pengcheng, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen (2021). "DeBERTa: Decoding-enhanced BERT with Disentangled Attention". In: *Proceedings of the International Conference on Learning Representations (ICLR)*.

He, Xinran, Theodoros Rekatsinas, James Foulds, Lise Getoor, and Yan Liu (2015). "HawkesTopic: A Joint Model for Network Inference and Topic Modeling from

Text-Based Cascades". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Heijden, Maarten Van der, Marina Velikova, and Peter JF Lucas (2014). "Learning Bayesian networks for clinical time series analysis". In: *Journal of biomedical informatics*.

Hinton, Geoffrey, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. (2012). "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups". In: *IEEE Signal processing magazine*.

Hirosawa, Makoto, Yasushi Totoki, Masaki Hoshida, and Masato Ishikawa (1995). "Comprehensive Study on Iterative Algorithms of Multiple Sequence Alignment". In: *Bioinformatics*.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation*.

Hua, William, Hongyuan Mei, Sarah Zohar, Magali Giral, and Yanxun Xu (2021). "Personalized Dynamic Treatment Regimes in Continuous Time: A Bayesian Joint Model for Optimizing Clinical Decisions with Timing". In: *Bayesian Analysis*.

Huang, Zhiheng, Wei Xu, and Kai Yu (2015). "Bidirectional LSTM-CRF models for sequence tagging". In: *arXiv preprint arXiv:1508.01991*.

Isham, Valerie and Mark Westcott (1979). "A self-correcting point process". In: *Stochastic Processes and their Applications*.

Jozefowicz, Rafal, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu (2016). "Exploring the Limits of Language Modeling". In: *arXiv: 1602.02410*.

Kalman, Rudolph E. and Richard S. Bucy (1961). "New Results in Linear Filtering and Prediction Theory". In: *Journal of Basic Engineering*.

Kalman, Rudolph Emil (1960). "A New Approach to Linear Filtering and Prediction Problems". In: *Journal of Basic Engineering*.

Kantorovitch, Leonid (1958). "On the Translocation of Masses". In: *Management Science*.

Kidambi, Rahul, Aravind Rajeswaran, Praneeth Netrapalli, and Thorsten Joachims (2020). "Morel: Model-based offline reinforcement learning". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Kiddon, Chloé, Luke Zettlemoyer, and Yejin Choi (2016). "Globally coherent text generation with neural checklist models". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Kingma, Diederik and Jimmy Ba (2015). "Adam: A Method for Stochastic Optimization". In: *Proceedings of the International Conference on Learning Representations (ICLR)*.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks". In:

Kumar, Ankit, Ozan Irsoy, Peter Ondruska, Mohit Iyyer, James Bradbury, Ishaan Gulrajani, Victor Zhong, Romain Paulus, and Richard Socher (2016a). "Ask Me Anything: Dynamic Memory Networks for Natural Language Processing". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Kumar, Srijan, Francesca Spezzano, VS Subrahmanian, and Christos Faloutsos (2016b). "Edge weight prediction in weighted signed networks". In:

Lample, Guillaume, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou (2019). "Large Memory Layers with Product Keys". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Larkin, Mark A., Gordon Blackshields, N. P. Brown, R. Chenna, Paul A. McGettigan, Hamish McWilliam, Franck Valentin, Iain M. Wallace, Andreas Wilm, Rodrigo Lopez, et al. (2007). "Clustal W and Clustal X Version 2.0". In: *Bioinformatics*.

Le, Phong and Willem Zuidema (2015). "The forest convolutional network: Compositional distributional semantics with a neural chart and without binarization". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Lee, Joon, Daniel J Scott, Mauricio Villarroel, Gari D Clifford, Mohammed Saeed, and Roger G Mark (2011). "Open-access MIMIC-II database for intensive care research". In: *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*.

Lee, Young, Kar Wai Lim, and Cheng Soon Ong (2016). "Hawkes Processes with Stochastic Excitations". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Leskovec, Jure, Daniel Huttenlocher, and Jon Kleinberg (2010). "Governance in social media: A case study of the Wikipedia promotion process". In:

Leskovec, Jure and Andrej Krevl (2014). *SNAP Datasets: Stanford Large Network Dataset Collection*.

Levenshtein, Vladimir Iosifovich (1965). "Binary codes capable of correcting deletions, insertions, and reversals". In: *Doklady Akademii Nauk*.

Levina, E. and P. Bickel (2001). "The Earth Mover's Distance is the Mallows Distance: Some Insights from Statistics". In: *Proceedings of the Eighth IEEE International Conference on Computer Vision (ICCV)*.

Levine, Sergey, Aviral Kumar, George Tucker, and Justin Fu (2020). "Offline reinforcement learning: Tutorial, review, and perspectives on open problems". In: *arXiv: 2005.01643*.

Lewis, J. (1991). "A Dynamic Load Balancing Approach to the Control of Multiserver Polling Systems with Applications to Elevator System Dispatching". PhD thesis. University of Massachusetts, Amherst.

Lewis, Peter A. and Gerald S. Shedler (1979). "Simulation of Nonhomogeneous Poisson Processes by Thinning". In: *Naval Research Logistics Quarterly*.

Lin, C., H. Zhu, M. R. Gormley, and J. M. Eisner (2019). "Neural Finite-State Transducers: Beyond Rational Relations". In: *NAACL-HLT*, pp. 272–283.

Lin, Chu-Cheng and Jason Eisner (2018). "Neural Particle Smoothing for Sampling from Conditional Sequence Models". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*.

Linderman, Scott W., Yixin Wang, and David M. Blei (2017). "Bayesian Inference for Latent Hawkes Processes". In: *Advances in Approximate Bayesian Inference Workshop, 31st Conference on Neural Information Processing Systems*.

Ling, Wang, Tiago Luís, Luís Marujo, Ramón Fernandez Astudillo, Silvio Amir, Chris Dyer, Alan W Black, and Isabel Trancoso (2015). "Finding function in form: Compositional character models for open vocabulary word representation". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Liniger, Thomas Josef (2009). "Multivariate Hawkes processes". Diss. Eidgenössische Technische Hochschule ETH Zürich, Nr. 18403.

Listgarten, Jennifer, Radford M. Neal, Sam T. Roweis, and Andrew Emili (2005). "Multiple Alignment of Continuous Time Series". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Little, Roderick J. A. and Donald B. Rubin (1987). *Statistical Analysis with Missing Data*.

Liu, Jun S. and Rong Chen (1998). "Sequential Monte Carlo Methods for Dynamic Systems". In: *Journal of the American Statistical Association*.

Lukasik, Michal, P. K. Srijith, Duy Vu, Kalina Bontcheva, Arkaitz Zubiaga, and Trevor Cohn (2016). "Hawkes Processes for Continuous Time Sequence Classification: An Application to Rumour Stance Classification in Twitter". In: *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

Ma, Zhuang and Michael Collins (2018). "Noise-Contrastive Estimation and Negative Sampling for Conditional Models: Consistency and Statistical Efficiency".

In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

McLachlan, Geoffrey and Thriyambakam Krishnan (2007). *The EM algorithm and Extensions*.

Meek, C. (2014). "Toward learning graphical and causal process models". In: *Uncertainty in Artificial Intelligence Workshop on Causal Inference: Learning and Prediction*. Vol. 1274, pp. 43–48.

Meent, Jan-Willem van de, Brooks Paige, Hongseok Yang, and Frank Wood (2018). "An introduction to probabilistic programming". In: *arXiv: 1809.10756*.

Mei, Hongyuan and Jason Eisner (2017). "The Neural Hawkes Process: A Neurally Self-Modulating Multivariate Point Process". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Mei, Hongyuan, Guanghui Qin, and Jason Eisner (2019). "Imputing Missing Events in Continuous-Time Event Streams". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Mei, Hongyuan, Guanghui Qin, Minjie Xu, and Jason Eisner (2020). "Neural Datalog Through Time: Informed Temporal Modeling via Logical Specification". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Mei, Hongyuan, Tom Wan, and Jason Eisner (2020). "Noise-Contrastive Estimation for Multivariate Point Processes". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Mei, Hongyuan, Chenghao Yang, and Jason Eisner (2020). "Continuous-Time Attention is All You Need". In: *Under submission*.

Mikolov, Tomas, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur (2010). "Recurrent Neural Network-Based Language Model". In: *Proceedings of the Annual Conference of the International Speech Communication Association (INTERSPEECH)*.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean (2013). "Distributed Representations of Words and Phrases and Their Compositionality". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Minka, T. and J. Winn (2008). "Gates: A Graphical Notation for Mixture Models". In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 1073–1080.

Mnih, Andriy and Geoffrey E Hinton (2009). "A scalable hierarchical distributed language model". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Mnih, Andriy and Yee Whye Teh (2012). "A Fast and Simple Algorithm for Training Neural Probabilistic Language Models". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Mohan, Karthika and Judea Pearl (2018). "Graphical Models for Processing Missing Data". In: *arXiv preprint arXiv:1801.03583*.

Moral, Pierre Del (1997). "Nonlinear Filtering: Interacting Particle Resolution". In: *Comptes Rendus de l'Academie des Sciences-Serie I-Mathematique*.

Mount, David W. (2004). *Bioinformatics: Sequence and Genome Analysis*.

Natarajan, Sriraam, Hung H Bui, Prasad Tadepalli, Kristian Kersting, and Weng-Keen Wong (2008). "Logical hierarchical hidden Markov models for modeling user activities". In: *Proceedings of the International Conference on Inductive Logic Programming (ICILP)*.

Notredame, Cédric, Desmond G. Higgins, and Jaap Heringa (2000). "T-Coffee: A Novel Method for Fast and Accurate Multiple Sequence Alignment". In: *Journal of molecular biology*.

Omi, Takahiro, Naonori Ueda, and Kazuyuki Aihara (2019). "Fully neural network based model for general temporal point processes". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Palm, C. (1943). *Intensitätsschwankungen im Fernsprechverkehr*. L. M. Ericcson.

Panzarasa, Pietro, Tore Opsahl, and Kathleen M Carley (2009). "Patterns and dynamics of users' behavior and interaction: Network analysis of an online community". In: *Journal of the American Society for Information Science and Technology*.

Paranjape, A., A. R. Benson, and J. Leskovec (2017). "Motifs in temporal networks". In: *Web Search and Data Mining*.

Parsons, Terence (1990). *Events in the Semantics of English*.

Paszke, Adam, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer (2017). "Automatic differentiation in PyTorch". In:

Pearl, Judea (2009). "Causal Inference in Statistics: An Overview". In: *Statistics Surveys*.

Peters, Matthew, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer (2018). "Deep Contextualized Word Representations". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*.

Poole, David (2010). *AILog User Manual, Version 2.3*. Available at https://www.cs.ubc.ca/~poole/aibook/code/ailog/ailog_man.html.

Puterman, Martin L. (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Rabiner, Lawrence R. (1989). "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition". In: *Proceedings of the IEEE*.

Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever (2019). "Language models are unsupervised multitask learners". In:

Raedt, L. De, A. Kimmig, and H. Toivonen (2007). "ProbLog: A Probabilistic Prolog and its Application in Link Discovery". In: *Proc. of IJCAI*, pp. 2462–2467.

Rao, Roshan, Joshua Meier, Tom Sercu, Sergey Ovchinnikov, and Alexander Rives (2021). "Transformer protein language models are unsupervised structure learners". In: *Proceedings of the International Conference on Learning Representations (ICLR)*.

Rao, Vinayak and Yee W. Teh (2012). "MCMC for Continuous-Time Discrete-State Systems". In: *Advances in Neural Information Processing Systems*.

Rao, Vinayak and Yee Whye Teh (2013). "Fast MCMC sampling for Markov Jump Processes and Extensions". In: *The Journal of Machine Learning Research*.

Rauch, Herbert E., C. T. Striebel, and F. Tung (1965). "Maximum Likelihood Estimates of Linear Dynamic Systems". In: *AIAA Journal*.

Rhinehart, Nicholas, Rowan McAllister, and Sergey Levine (2018). "Deep imitative models for flexible inference, planning, and control". In: *arXiv: 1810.06544*.

Richardson, Matthew and Pedro Domingos (2006). "Markov logic networks". In: *Machine learning*.

Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review*.

Rubanova, Yulia, Ricky TQ Chen, and David Duvenaud (2019). "Latent odes for irregularly-sampled time series". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1986). "Learning representations by back-propagating errors". In: *Nature*.

Sakoe, Hiroaki and Seibi Chiba (1971). "A Dynamic Programming Approach to Continuous Speech Recognition". In: *Proceedings of the Seventh International Congress on Acoustics, Budapest*.

Sato, Taisuke (1995). "A Statistical Learning Method for Logic Programs With Distribution Semantics". In: *Proc. of ICLP*, pp. 715–729.

Shchur, Oleksandr, Marin Biloš, and Stephan Günnemann (2020). "Intensity-free learning of temporal point processes". In: *Proceedings of the International Conference on Learning Representations (ICLR)*.

Shelton, C. R. and G. Ciardo (2014). "Tutorial on structured continuous-time Markov processes". In: *Journal of Artificial Intelligence Research* 51, pp. 725–778.

Shelton, Christian R., Zhen Qin, and Chandini Shetty (2018). "Hawkes Process Inference with Missing Data". In: *Proceedings of the AAAI Conference on Artificial Intelligence*.

Silver, David, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *Nature*.

Socher, Richard, Brody Huval, Christopher D. Manning, and Andrew Y. Ng (2012). "Semantic Compositionality through Recursive Matrix-Vector Spaces". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Spall, James C. (2005). *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*.

Srivastava, Rupesh Kumar, Klaus Greff, and Jürgen Schmidhuber (2015). "Highway Networks". In: *arXiv preprint arXiv:1505.00387*.

Sukhbaatar, Sainbayar, Jason Weston, Rob Fergus, et al. (2015). "End-to-End Memory Networks". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Sundermeyer, Martin, Hermann Ney, and Ralf Schluter (2012). "LSTM Neural Networks for Language Modeling". In: *Proceedings of the Annual Conference of the International Speech Communication Association (INTERSPEECH)*.

Sutton, Richard S and Andrew G Barto (2018). *Reinforcement learning: An introduction*. MIT press.

Swift, Terrance and David S. Warren (2012). "XSB: Extending Prolog with Tabled Logic Programming". In: *Theory and Practice of Logic Programming* 12.1–2, pp. 157–187. DOI: 10.1017/S1471068411000500.

Tai, Kai Sheng, Richard Socher, and Christopher D Manning (2015). "Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks". In: *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

Thrun, Sebastian, Wolfram Burgard, and Dieter Fox (2005). *Probalistic robotics*. MIT press.

Tran, Ke M., Yonatan Bisk, Ashish Vaswani, Daniel Marcu, and Kevin Knight (2016). "Unsupervised Neural Hidden Markov Models". In: *Proceedings of the Workshop on Structured Prediction for NLP*. Austin, TX, pp. 63–71. DOI: 10.18653/v1/W16-5907.

Trivedi, R., H. Dai, Y. Wang, and L. Song (2017). "Know-Evolve: Deep temporal reasoning for dynamic knowledge graphs". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Trivedi, Rakshit, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha (2019). "DyRep: Learning Representations over Dynamic Graphs". In: *Proceedings of the International Conference on Learning Representations (ICLR)*.

Upadhyay, Utkarsh, Abir De, and Manuel Gomez Rodriguez (2018). "Deep reinforcement learning of marked temporal point processes". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin (2017). "Attention is All You Need". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Vaswani, Ashish, Yinggong Zhao, Victoria Fossum, and David Chiang (2013). "Decoding with large-scale neural language models improves translation". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Villani, Cédric (2008). *Optimal Transport: Old and New*.

Wang, Y., A. Smola, D. C. Maddix, J. Gasthaus, D. Foster, and T. Januschowski (2019). "Deep Factors for Forecasting". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Wang, Yichen, Bo Xie, Nan Du, and Le Song (2016). "Isotonic Hawkes Processes". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Wei, Greg C. G. and Martin A. Tanner (1990). "A Monte Carlo Implementation of the EM Algorithm and the Poor Man's Data Augmentation Algorithms". In: *Journal of the American Statistical Association*.

Weston, Jason, Sumit Chopra, and Antoine Bordes (2015). "Memory Networks". In: *Proceedings of the International Conference on Learning Representations (ICLR)*.

Whong, Chris (2014). *FOILing NYC's Taxi Trip Data*. Blog.

Williams, R.J. (1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning*.

Williams, Ronald J. and David Zipser (1989). "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks". In: *Neural Computation* 1.2, pp. 270–280.

Xiao, Chunyang, Christoph Teichmann, and Konstantine Arkoudas (2019). "Grammatical Sequence Prediction for Real-Time Neural Semantic Parsing". In: *Proceedings of the ACL Workshop on Deep Learning and Formal Languages: Building Bridges*.

Xiao, Shuai, Mehrdad Farajtabar, Xiaojing Ye, Junchi Yan, Xiaokang Yang, Le Song, and Hongyuan Zha (2017a). "Wasserstein Learning of Deep Generative Point Process Models". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Xiao, Shuai, Junchi Yan, Mehrdad Farajtabar, Le Song, Xiaokang Yang, and Hongyuan Zha (2017b). "Joint Modeling of Event Sequence and Time Series with Attentional Twin Recurrent Neural Networks". In: *arXiv preprint arXiv:1703.08524*.

Xiao, Shuai, Junchi Yan, Xiaokang Yang, Hongyuan Zha, and Stephen Chu (2017c). "Modeling the intensity function of point process via recurrent neural networks". In: *Proceedings of the AAAI Conference on Artificial Intelligence*.

Xu, Da, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan (2020). "Inductive representation learning on temporal graphs". In: *Proceedings of the International Conference on Learning Representations (ICLR)*.

Xu, Hongteng, Mehrdad Farajtabar, and Hongyuan Zha (2016). "Learning Granger Causality for Hawkes Processes". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Xu, Hongteng, Dixin Luo, and Lawrence Carin (2018). "Online Continuous-Time Tensor Factorization Based on Pairwise Interactive Point Processes." In: *Proceedings of the International Conference on Learning Representations (ICLR)*.

Yang, Shuang-hong and Hongyuan Zha (2013). "Mixture of Mutually Exciting Processes for Viral Diffusion". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Yu, Tianhe, Garrett Thomas, Lantao Yu, Stefano Ermon, James Zou, Sergey Levine, Chelsea Finn, and Tengyu Ma (2020). "Mopo: Model-based offline policy optimization". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Zhang, Qiang, Aldo Lipani, Omer Kirnap, and Emine Yilmaz (2020a). "Self-attentive Hawkes process". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Zhang, Wei, Thomas Kobber Panum, Somesh Jha, Prasad Chalasani, and David Page (2020b). "CAUSE: Learning Granger Causality from Event Sequences using Attribution Methods". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Zhang, Xingxing, Liang Lu, and Mirella Lapata (2016). "Top-down Tree Long Short-Term Memory Networks". In: *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics – Human Language Technologies (NAACL HLT)*.

Zhao, Qingyuan, Murat A. Erdogdu, Hera Y. He, Anand Rajaraman, and Jure Leskovec (2015). "Seismic: A Self-Exciting Point Process Model for Predicting Tweet Popularity". In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Zhou, Ke, Hongyuan Zha, and Le Song (2013). "Learning Triggering Kernels for Multi-dimensional Hawkes Processes". In: *Proceedings of the International Conference on Machine Learning (ICML)*.

Zuo, Simiao, Haoming Jiang, Zichong Li, Tuo Zhao, and Hongyuan Zha (2020). "Transformer Hawkes Process". In: *International Conference on Machine Learning*. PMLR, pp. 11692–11702.

Hongyuan Mei

Contact Information

Email: hongyuanmei@gmail.com, hmei@cs.jhu.edu
Webpage: `http://www.cs.jhu.edu/~hmei/`
Github: `https://github.com/HMEIatJHU/`

Research Focus

Machine learning, with a focus on modeling sequences (e.g., events, natural language sentences).

Employment

Toyota Technological Institute at Chicago — Chicago, Illinois, USA
Research Assistant Professor — 09/2021 - Present

Education

Johns Hopkins University — Baltimore, Maryland, USA
Ph.D., Computer Science, advised by Jason Eisner — 09/2016 - 07/2021

The University of Chicago — Chicago, Illinois, USA
M.S., Physical Science, advised by Mohit Bansal and Matthew R. Walter — 09/2015 - 06/2016
M.S., Financial Mathematics — 09/2012 - 06/2013

Huazhong University of Science and Technology — Wuhan, Hubei, China
B.E., Electrical Engineering; B.A., Finance — 09/2008 - 06/2012

Honors and Awards

ICLR Outstanding Reviewer	2021
Bloomberg Data Science Ph.D. Fellowship	2018 - 2021
JHU-CLSP Jelinek Fellowship (one student per year)	2020
EMNLP Outstanding Reviewer	2020
ICML Top 33% Reviewer	2020
ACL Outstanding Reviewer	2018
NeurIPS Travel Award	2017
UChicago Merit-based Scholarships	2015, 2016
UChicago GRAD Travel Award	2015
Outstanding Graduate in HUST	2012
Outstanding Student Leadership Award	2011
Outstanding Student Award (1%)	2011
Chinese National Science Foundation, Undergraduate Research Funding	2009, 2010
Model Student Scholarship (10%)	2009

Professional Service

Organizer:

ACL Workshop on Representation Learning for NLP (RepL4NLP)	2018

Committee Member:

ACL Language Grounding for Robotics (RoboNLP)	2017
AAAI Symposium on Natural Communication for Human-Robot Collaboration (NCHRC)	2017

Reviewer:

AAAI	2018
ACL	2017, 2018
EMNLP	2018, 2019, 2020
ICLR	2017, 2019, 2020, 2021
ICML	2019, 2020
NeurIPS	2018, 2019, 2020

Teaching Experience

Course Instructor, Bloomberg ML Course on Modeling Irregular Time Series	Fall, 2020
Guest lecturer, JHU Summer School on Human Language Technology	Summer, 2018

Teaching Assistant, JHU.CS.600.465—Natural Language Processing — Fall, 2017
Guest lecturer, JHU.CS.600.466—Information Retrieval and Web Agents — Spring, 2017

Publications

William Hua, **Hongyuan Mei**, Sarah Zohar, Magali Giral, Yanxun Xu
Personalized Dynamic Treatment Regimes in Continuous Time: A Bayesian Joint Model for Optimizing Clinical Decisions with Timing
Under journal submission
Selected as one of the International Biometric Society ENAR Distinguished Student Paper Awards

Hongyuan Mei, Tom Wan, Jason Eisner
Noise-Contrastive Estimation for Multivariate Point Processes
Proceedings of NeurIPS 2020

Hongyuan Mei, Guanghui Qin, Minjie Xu, Jason Eisner
Neural Datalog Through Time: Informed Temporal Modeling via Logical Specification
Proceedings of ICML 2020

Hongyuan Mei, Guanghui Qin, Jason Eisner
Imputing Missing Events in Continuous-Time Event Streams
Proceedings of ICML 2019

Shijia Liu, **Hongyuan Mei**, Adina Williams, Ryan Cotterell
On the Idiosyncrasies of the Mandarin Chinese Classifier System
Proceedings of NAACL 2019

Hongyuan Mei, Sheng Zhang, Kevin Duh, Benjamin Van Durme
Halo: Learning Semantics-Aware Representations for Cross-Lingual Information Extraction
*Proceedings of *SEM 2018*

Hongyuan Mei, Jason Eisner
The Neural Hawkes Process: A Neurally Self-Modulating Multivariate Point Process
Proceedings of NeurIPS 2017

Hongyuan Mei, Mohit Bansal, Matthew R. Walter
Coherent Dialogue with Attention-based Language Models
Proceedings of AAAI 2017

Hongyuan Mei, Mohit Bansal, Matthew R. Walter
What to talk about and how? Selective Generation using LSTMs with Coarse-to-Fine Alignment
Proceedings of NAACL 2016

Hongyuan Mei, Mohit Bansal, Matthew R. Walter
Listen, Attend, and Walk: Neural Mapping of Navigational Instructions to Action Sequences
Proceedings of AAAI 2016
Selected Oral with NVIDIA GPU Prize in NeurIPS 2015 Multimodal Machine Learning workshop

Research Experience

Bloomberg LP — New York, New York, USA
Research Intern, advised by Minjie Xu — 06/2019 - 08/2019

Microsoft Research — Redmond, Washington, USA
Research Intern, advised by Chris Quirk — 05/2016 - 08/2016

Toyota Technological Institute at Chicago — Chicago, Illinois, USA

Research Assistant, advised by Mohit Bansal and Matthew R. Walter — 03/2015 - 05/2016

Booth School of Business at The University of Chicago — Chicago, Illinois, USA
Research Assistant, advised by Dacheng Xiu — 06/2013 - 09/2013

Huazhong University of Science and Technology — Wuhan, Hubei, China
Research Assistant, advised by Jiaolong Wei — 10/2009 - 06/2011

TECHNICAL SKILLS

- Languages: PyTorch, Theano, Python, MATLAB, R, Java, C++.
- Operating Systems: Unix/Linux, Windows.