TRANSFORMER EMBEDDINGS OF IRREGULARLY SPACED EVENTS AND THEIR PARTICIPANTS

Anonymous authors
Paper under double-blind review

ABSTRACT

The Transformer (Vaswani et al., 2017) has enjoyed remarkable success at learning representations for *discrete-time* sequences, such as natural language sentences. Yet there exists little work on generalizing its success to sequences of discrete events in *continuous* time. We design a generative point process model for this case 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** (Mei et al., 2020a), a neural-symbolic framework for event sequence modeling that automatically derives deep recurrent neural models from temporal logic programs. Our Transformers replace the LSTM-style mechanism to capture temporal dependencies. We evaluated our novel architecture on several synthetic and real-world datasets, evaluated by likelihood and prediction accuracy. On most of them, it surpasses the previous version and other strong competitors; on the others, it is comparable to the state-of-the-art.

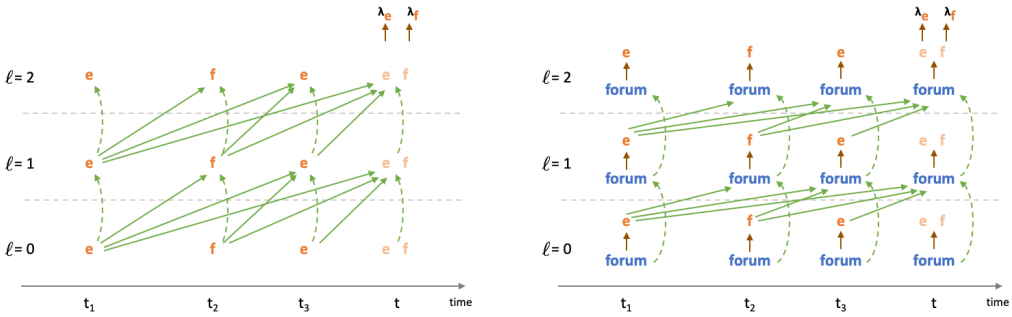
1 INTRODUCTION

Event sequences are ubiquitous in real-world applications, such as healthcare, finance, education, commerce, gaming, audio, news, security, and social media. 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, similar to word token embeddings in BERT. In particular, we consider the task of predicting the next event (generative modeling): at each time $t \in \mathbb{R}$, all of the possible events are embedded using masked attention over the preceding actual events, and those embeddings yield the instantaneous probabilities at time t of the possible events.

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 paper that they are also effective for irregularly spaced continuous-time sequences, even in lower-data regimes.

We further show how to derive models from arbitrary NDTT programs. NDTT, or *neural Datalog through time* (Mei et al., 2020a), is a neural-symbolic framework for modeling event sequences. An NDTT program specifies logical rules that derive facts within a deductive database. Moreover, the facts in the database are embedded using a deep recurrent neural architecture that is automatically determined by the facts' provenance. Among the derived facts at time t are the possible events at time t.

As a simple NDTT example, *if* u is a user *and* d is a document *and* u has appropriate permissions, *then* u may open d. The embedding and instantaneous probability of this *possible* event at time t depend on the embeddings at time t of u and d (or more precisely, of the facts that u is a user and d is a document). If the event *actually* occurs, it may change what facts are true at times $> t$ and how their embeddings evolve over times $> t$. For instance, the NDTT program can state that if u opens d, this makes editing events possible, and also affects the future embeddings of u and d. These updated embeddings feed forward to help predict future events involving u and/or d. (They could also be exploited for other downstream predictive tasks, via multitask training or pretraining.) All of these effects are controlled by the hand-crafted rules and learned parameters of the NDTT program,



(a) A-NHP (b) A-NDTT

Figure 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, $\lambda_{e}(t)$ and $\lambda_{f}(t)$, 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 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 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, 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).

which are analogous to the hand-crafted structure and learned parameters of a graphical model, in that they enforce domain-appropriate conditional independence and parameter sharing.

We modify the NDTT architectures to use attention. In our architecture, a fact that can be derived in multiple ways, by different NDTT rules, uses multiple attention heads to examine the past. Du et al. (2016); Mei & Eisner (2017); Mei et al. (2020a) instead summarize the past using recurrent neural networks (RNNs). Our attentive approach has three advantages over theirs:

① A Transformer does not summarize the past. Our predictions at time t can examine an unboundedly large representation of the past ($\mathbb{R}^d$ embeddings of *every* event before t), not merely a fixed-dimensional summary that was computed greedily from left to right (an RNN's state at time t).

② A Transformer's computation graph is broader and shallower. The breadth makes it easier to learn long-distance influences. The shallowness does make it impossible to represent inherently deep concepts such as parity, but it enables 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 arguably more natural, while remaining competitive in our experiments. To describe how embeddings vary continuously during the period $(t_i, t_{i+1}]$ between two events, Mei & Eisner (2017) had to stipulate an arbitrary family of parametric decay functions on t, and Mei et al. (2020a) had to design a complex method for pooling the parameters of these decay functions across multiple NDTT rules. In contrast, our embeddings are constructed "from scratch" at each time t simply by attending to 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 attention weights over the first i events vary continuously with t, so the embeddings do so too.

2 Continuous-Time Transformer for Embedding 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 $\ell > 0$, the layer-ℓ embedding of $e@t$ is computed as

$$[\![e]\!]^{(\ell)}(t) \stackrel{\text{def}}{=} \underbrace{[\![e]\!]^{(\ell-1)}(t)}_{\text{residual connection}} + \tanh\left(\sum_{f@s \in \mathcal{H}(e@t)} \frac{\mathbf{v}^{(\ell)}(f@s)\ \alpha^{(\ell)}(f@s, e@t)}{\sum_{f@s \in \mathcal{H}(e@t)} \alpha^{(\ell)}(f@s, e@t)}\right) \tag{1}$$

where the unnormalized attention weight on each relevant event $f@s \in \mathcal{H}(k@t)$ is

$$\alpha^{(\ell)}(f@s, e@t) \stackrel{\text{def}}{=} \exp\left(\frac{1}{\sqrt{D}}\ \mathbf{k}^{(\ell)}(f@s)^\top \mathbf{q}^{(\ell)}(e@t)\right) \in \mathbb{R} \tag{2}$$

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 $\mathbf{V}^{(\ell)}, \mathbf{K}^{(\ell)}, \mathbf{Q}^{(\ell)}$:

$$\mathbf{v}^{(\ell)}(e@t) \stackrel{\text{def}}{=} \mathbf{V}^{(\ell)}\ \left[1; [\![t]\!]; [\![e]\!]^{(\ell-1)}(t)\right] \tag{3a}$$

$$\mathbf{k}^{(\ell)}(e@t) \stackrel{\text{def}}{=} \mathbf{K}^{(\ell)}\ \left[1; [\![t]\!]; [\![e]\!]^{(\ell-1)}(t)\right] \tag{3b}$$

$$\mathbf{q}^{(\ell)}(e@t) \stackrel{\text{def}}{=} \mathbf{Q}^{(\ell)}\ \left[1; [\![t]\!]; [\![e]\!]^{(\ell-1)}(t)\right] \tag{3c}$$

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:

$$[\![t]\!]_d = \cos(t/10000^{\frac{d-1}{D}}) \text{ if } d \text{ is odd and } [\![t]\!]_d = \sin(t/10000^{\frac{d}{D}}) \text{ if } d \text{ is even} \tag{4}$$

We concatenate $[\![t]\!]$ to the rest of the embedding rather than adding it (cf. Kitaev & Klein, 2018; He et al., 2020); furthermore we do so again at every layer.

According to equations (1)–(3), to compute the layer-ℓ embedding of an event, we only need the layer-($\ell-1$) 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 $\underline{\texttt{mask}}@t_i$. Such embeddings could be pretrained with a BERT-like objective and then fine-tuned to predict properties of the observed events.

3 Generative Modeling of Continuous-Time Event Sequences

In this paper, 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 called the (Poisson) **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 $e@t$:

$$\lambda_e(t) \stackrel{\text{def}}{=} \mathrm{softplus}(\mathbf{w}_e^\top [1; [\![e]\!]^L(t)], \tau_e) \quad \text{where } \mathrm{softplus}(x, \tau) = \tau \log(1 + \exp(x/\tau)) > 0 \tag{5}$$

with learnable parameters $\mathbf{w}_e$ and $\tau_e > 0$. We do this separately for each possible $e@t$, computing the embedding $[\![e]\!]^L(t)$ using equations (1)–(3).

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 (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 equation 1).

Embedding each possible $e@t$ separately is somewhat expensive. The attention weights $\alpha^{(1)}, \ldots, \alpha^{(L)}$ used to compute its embedding and intensity must be recomputed for each e and t. This is more

expressive but also more expensive than a standard Transformer language model, which instead models the probability of word e at time t as proportional to $\exp(\mathbf{w}_e^\top [1; [\![f]\!]^L(s)])$ where $[\![f]\!]^L(s)$ is the top-layer embedding of the *previous* word $f@s$, where $t = s + 1$, and can be reused for all possible choices of $e@t$. Why do we not follow that cheaper approach, using the *previous* event $f@s$ in place of $e@t$ in equation (5)? Because then $\lambda_e(t)$ would depend only on s and not on t (a language model can rely on $t = s + 1$, but we may have to evaluate intensities at all real-valued $t \geq s$ since our events are irregularly spaced). Our more expressive design also allows the attention vector used to compute $\lambda_e(t)$ to depend on e, to support the "selective attention" of section 4 below.

As a speedup, however, we can associate each event type e with a **coarse event type** $\bar{e}$ that has the same selective attention pattern, and replace $[\![e]\!]^L(t)$ with $[\![\bar{e}]\!]^L(t)$ in equation (5). (However, equation (5) still uses the fine-grained $\mathbf{w}_e$.) Now to compute $\lambda_e(t)$, we only have to embed $\bar{e}@t$, which is a speedup if many of the possible event types e are associated with the same $\bar{e}$. In the case where we do not use selective attention, we can get away with using only a single coarse event type for the whole model—saving a runtime factor of E as in the cheaper approach. Note that the history $\mathcal{H}$ still uses fine-grained embeddings, so if $e@t$ actually occurs, we must then compute $[\![e]\!]^0(t), \ldots, [\![e]\!]^L(t)$.

Our model falls into the family of **multivariate point processes**. Its log-likelihood is given by

$$\sum_{i=1}^{I} \log \lambda_{e_i}(t_i) - \int_{t=0}^{T} \sum_{e=1}^{E} \lambda_e(t) \mathrm{d}t \tag{6}$$

This formula has been derived and discussed in previous work (Hawkes, 1971; Liniger, 2009; Mei & Eisner, 2017). 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.

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: $\underline{\mathtt{e}}$ means that Eve emails Adam, while $\underline{\mathtt{f}}$ means that Frank emails Eve. As Frank does not know when Eve emailed Adam, past events of type $\underline{\mathtt{e}}$ cannot influence his behavior. Therefore, $\mathcal{H}(\underline{\mathtt{f}}@t)$ should include past events of type $\underline{\mathtt{f}}$ but not $\underline{\mathtt{e}}$, so that the embedding of $\underline{\mathtt{f}}@t$ and hence the intensity function $\lambda_{\underline{\mathtt{f}}}(t)$ pay zero attention to $\underline{\mathtt{e}}$ events. In contrast, $\mathcal{H}(\underline{\mathtt{e}}@t)$ 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 $\underline{\mathtt{f}} \leftarrow \underline{\mathtt{f}}$, $\underline{\mathtt{e}} \leftarrow \underline{\mathtt{e}}$, $\underline{\mathtt{e}} \leftarrow \underline{\mathtt{f}}$. These edges are called **rules**, for reasons that will become clear in section 5.

Furthermore, when Eve decides whether to email Adam ($\underline{\mathtt{e}}@t$), we may reasonably suppose that she *separately* considers the embeddings of the past $\underline{\mathtt{e}}$ events (e.g., "when was my last relevant email to Adam?") versus the past $\underline{\mathtt{f}}$ events (e.g., "what have I heard from Frank?"). For this reason, we associate different attention heads with the two rules that affect $\underline{\mathtt{e}}$, namely $\underline{\mathtt{e}} \leftarrow \underline{\mathtt{e}}$ and $\underline{\mathtt{e}} \leftarrow \underline{\mathtt{f}}$. 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 (1) with

$$[\![e]\!]^{(\ell)}(t) \stackrel{\text{def}}{=} [\![e]\!]^{(\ell-1)}(t) + \tanh\left(\sum_r \boxed{e}_r^{(\ell)}(t)\right) \tag{7}$$

$$\boxed{e}_r^{(\ell)}(t) \stackrel{\text{def}}{=} \sum_{f@s \in \mathcal{H}_r(e@t)} \frac{\mathbf{v}_r^{(\ell)}(f@s)\ \alpha_r^{(\ell)}(f@s, e@t)}{\sum_{f@s \in \mathcal{H}_r(e@t)} \alpha_r^{(\ell)}(f@s, e@t)} \tag{8}$$

where r in the summation ranges over rules $e \leftarrow \cdots$. The history $\mathcal{H}_r(\underline{\mathtt{e}}@t)$ contains only those past events $f@s$ that rule r makes visible to e, as well as $e@t$ itself as discussed in section 3. The r-specific α and $\mathbf{v}$ quantities are defined using versions of equations (2)–(3) with r-specific parameters.[1]

5 Attentive Neural Datalog Through Time

[1]The NDTT formalism does allow parameters to be explicitly shared across rules when desired. Its :: notation (Mei et al., 2020a, Appendix B) can also be used in our setup.

Edges such as e <- f can be regarded as simple examples of rules in a Neural Datalog Through Time program (Mei et al., 2020a). We briefly review this formalism and then extend our approach from section 4 to handle all NDTT programs.

Datalog. Recall that Datalog (Ceri et al., 1989) is a traditional formalism for deductive databases. A deductive database holds both **extensional facts**, which are placed there by some external process, and **intensional facts**, which are transitively deduced from the extensional facts. A Datalog program is simply a set of rules that govern these deductions:

- h :- g_1, ..., g_n says to **deduce** h at any time t when $g_1, \ldots, g_n$ are all true (in the database).

A single rule can license many deductions. That is because the facts can have structured names, and $h, g_1, \ldots g_n$ can be patterns that match against those names. In the example at the bottom of page 1, we would have a rule like open(U,D) :- user(U), group(G), document(D), member(U,G), readable(D,G), where U, G, and D are variables. In English, this says that the event open(U,D) is possible[2] if user U is a member of some group G such that document D is readable by G.

Datalog Through Time. Whenever extensional facts are added or removed, the intensional facts are instantly recomputed according to the deductive rules. Datalog Through Time is an extension in which extensional facts are *automatically* added and removed when the database is notified of the occurrence of events. This behavior is governed by two additional rule types:

- h <- f, g_1, ..., g_n says to **add** h at any time s when event f occurs and the g_i are all true.
- !h <- f, g_1, ..., g_n says to **remove** h at any time s satisfying the same conditions.

Thus, the proposition h is true at time t (i.e., appears as a fact in the database at time t) iff either ❶ h is deduced at time t, or ❷ h was added at some time $s < t$ and never removed at any time in (s, t).

Neural Datalog Through Time. It would be difficult to train a neural architecture to encode thousands or millions of structured boolean facts about the world in its state and to modify those encoded facts systematically and transitively in response to events. As a *neuro-symbolic* method, NDTT delegates that task to a symbolic database governed by manually specified Datalog Through Time rules. However, it augments the database by neurally embedding every fact in a vector space. Thus, iff a proposition h appears as a fact in the NDTT database at time t, it will be associated not only with the simple truth value true but also with an embedding $[\![h]\!](t)$. This embedding is a *learned representation* of that fact at that time, and can be trained to be useful in downstream tasks. It captures details of when and how that fact was established (the fact's **provenance**), since it is computed using learned parameters associated with the rules that deduced and/or added it.

For example, if we write NDTT rules user(U) <- $\cdots$ that add user(U) (perhaps redundantly) each time a relevant event affects it, then $[\![$user(456)$]\!](t)$ reflects the embeddings and timings of all events before t that affected user(456). This embedding may help predict what user 456 will do in future. In our example, if open(456,78) is deduced with the help of user(456), then the embedding $[\![$open(456,78)$]\!](t)$—and its probability—in turn depends on $[\![$user(456)$]\!](t)$ and varies along with it as t changes.

In general, if e denotes the time-varying boolean proposition that an event of type e is possible, it follows that $[\![e]\!](t)$ will be defined at all times t when that proposition is true—i.e., at all times when e is possible. This embedding is used in equation (5) to model the intensity function $\lambda_e(t)$, which governs when e is likely to actually occur.

Contribution. Our goal 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 1.

Under A-NDTT, the layer-ℓ embedding of $h@t$ is then

$$[\![h]\!]^{(\ell)}(t) \stackrel{\text{def}}{=} [\![h]\!]^{(\ell-1)}(t) + \tanh\left([h]^{(\ell)}(t) + \sum_r \boxed{h}_r^{(\ell)}(t)\right) \tag{9}$$

which is an augmented version of equation (7). Suppose h is true at time t because it was added by rule r (i.e., condition ❷). Then the summand $\boxed{h}_r^{(\ell)}(t)$ exists and is computed much as in equation (8), now

[2]Following Mei et al. (2020a), we use orange to indicate facts like open(U,D) that specify possible events.

with attention over all add times s. In other words, $\mathcal{H}_r(h@t)$ includes just those past events $f@s$ such that f added h via r at some time $s < t$ and h was never removed at any time in (s, t). It also includes $h@t$ itself, as before. We make one enhancement: when the rule h `<-` f `,` g_1 `, ... ,` g_n causes $h@t$ to attend to the specific past event $f@s$, we actually want attention to consider the embedding at time s of the *entire* add condition $f, g_1, \ldots, g_n$. Thus the attention key of $f@s$ for this purpose is now defined as $\mathbf{k}^{(\ell)}(f@s) \stackrel{\text{def}}{=} \mathbf{K}_r^{(\ell)} \left[1; [\![s]\!]; [\![f]\!]^{(\ell-1)}(s); [\![g_1]\!]^{(\ell-1)}(s); \ldots [\![g_n]\!]^{(\ell-1)}(s)\right]$ (compare equation (3b)). Its attention value $\mathbf{v}^{(\ell)}(f@s)$ is defined analogously, using a different matrix $\mathbf{V}_r^{(\ell)}$.

How about the vector $[h]^{(\ell)}(t)$ in equation (9)? For each rule that deduces h at time t (i.e., condition ❶), say h `:-` $g_1, \ldots g_n$, we compute a nonlinear transform of the concatenated embeddings $[\![g_i]\!]^{(\ell)}(t)$. 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 Mei et al. (2020a), except that we now do it anew at each level ℓ.

The specification of parameter dimensionalities is discussed in Appendix A0.

Mei et al. (2020a) 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 (called `forum`) whose embedding (called $[\![$`forum`$]\!]$) always reflects all messages posted so far 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:

```
1 forum <- create.
2 forum <- e.
3 forum <- f.
4 e :- forum.
5 f :- forum.
6 !forum <- destroy.
```

The resulting neural architecture is drawn in Figure 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 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.[3] 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.

```
7  forum(T) <- create(T).
8  forum(T) <- message(E,T).
9  message(E,T) :- forum(T), member(E,T).
10 !forum(T) <- destroy(T).
```

Additional rules could describe how the `member` facts and their embeddings change over time, which may in turn affect the embeddings and predictions of future messages. See the discussion in section 1.

6 Training and Inference

We can estimate the parameters by locally maximizing the log-likelihood given in equation (6) using any stochastic gradient method. Details are given in Appendix A, including Monte Carlo approximations to the integral. Given the learned parameters, we may wish to make a minimum Bayes risk prediction about the next event given the past history. A recipe can be found in Appendix B.

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 et al., 2016; Zhang et al., 2020b), network analysis (Choi 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 & Schmidhuber, 1997): in particular Du et al. (2016); Mei & Eisner (2017); Xiao et al. (2017a;b); Omi et al. (2019); Shchur et al. (2020); Mei et al. (2020a); Boyd et al. (2020). Models of this kind enjoy continuous and

[3] Although a notation is available to demand employee-specific and forum-specific parameters if desired.

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 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.[4] Our model embeds all *possible* events at all *possible* times with attention as well (see ③ in section 1), thus being more flexible (see section 3) and achieving better performance (see section 8).

8 EXPERIMENTS

On several synthetic and real-world datasets, we evaluate our model's held-out log-likelihood, and its success at predicting the time and type of the next event. We compare with multiple strong competitors. Experimental details are given in Appendix C.

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 (Mei et al., 2020a). We then used it to implement our individual models. Our code and datasets are available at `code-public-after-paper-published`.

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 & 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 7 and footnote 4.

Self-Attentive Hawkes Process (SAHP (Zhang et al., 2020a). SAHP is very similar to THP but uses a slightly more flexible intensity function; see footnote 4.

For the above models, we make use of their published implementation.[5] Details are in Appendix C.2.

Attentive neural Hawkes process (A-NHP) This is our unstructured generative model from section 3. Since this model does not use selective attention, we speed up the intensity computations by defining them in terms of a single coarse event type, as described beneath equation (5). Thus, each event intensity $\lambda_e(t)$ is computed by attention over all previous events, where the attention weights are independent of e. This parameter-sharing mechanism resembles the NHP, except that we now use a Transformer in place of an LSTM.[6]

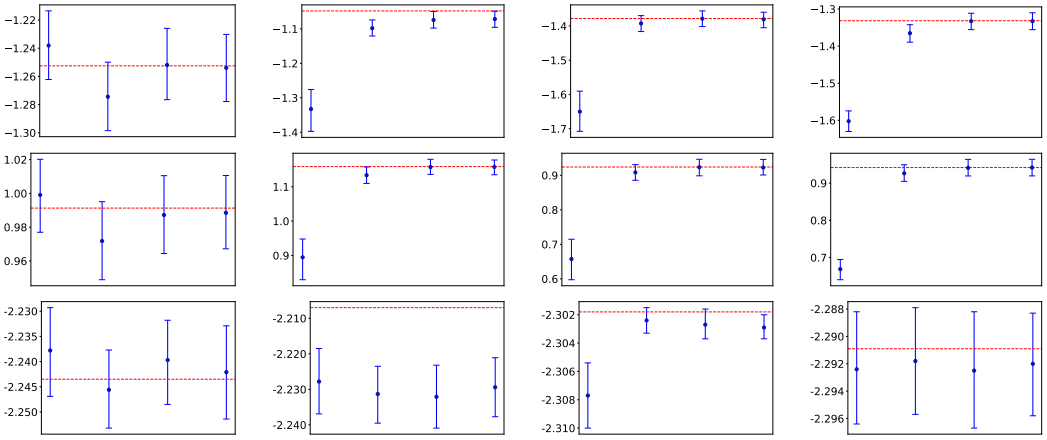
In a pilot experiment, we draw sequences from randomly initialized models (details in Appendix C.1.1), and fit the above models on these synthetic datasets.The results are shown in Figure 2. As shown in Figure 2c, on the synthetic NHP data, the performance of A-NHP is essentially the same as that of NHP, meaning that A-NHP is still a good choice even when it is misspecified. This result is reassuring because A-NHP has less capacity in some ways (the circuit depth of a Transformer is fixed, whereas the circuit depth of an LSTM grows with the length of the sequence) and excess capacity in other ways (the Transformer has unbounded memory whereas the LSTM has finite-dimensional memory). We also tried fitting A-NHP to the synthetic A-NHP, THP, and SAHP data. As shown in Figure 2d, the A-NHP data can not be well modeled by THP or SAHP, but it can be successfully modeled by—unsurprisingly—A-NHP and—maybe surprisingly—NHP. On the SAHP and THP data,

[4]The Transformer Hawkes process defines $\lambda_e(t) \stackrel{\text{def}}{=} \text{softplus}(\mathbf{w}_e^\top [1; t/t_i; [\![e_i]\!](t_i)])$ where $e_i @ t_i$ is the most recent event before t; the self-attentive Hawkes process conditions the linear projection $\mathbf{w}$ on $[\![e]\!](t_i)$ as well.

[5]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).

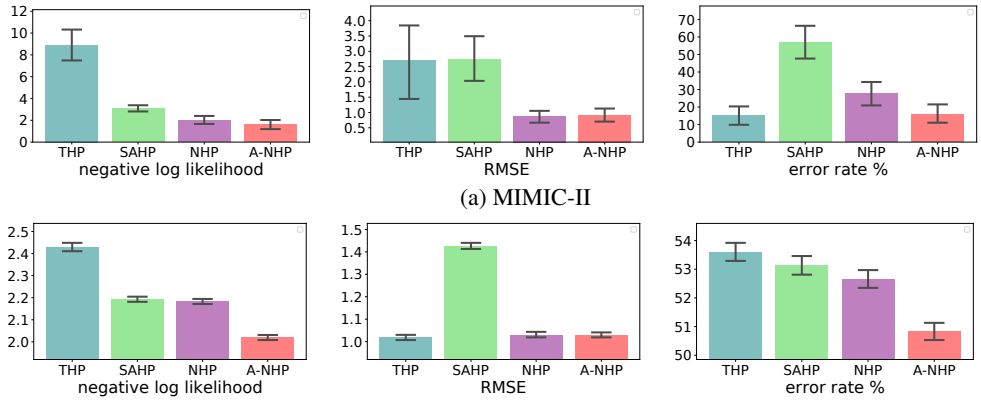
[6]For users' convenience, we made a standalone GPU-friendly PyTorch implementation of A-NHP.

A-NHP still gets competitive performance (Figures 2a and 2b). Notably, THP fits the time intervals poorly when it is misspecified, due to its restrictive handling of the passage of time (see section 7).



(a) THP Data (b) SAHP Data (c) NHP Data (d) A-NHP Data

Figure 2: Log-likelihood on held-out data (nats per event, with 95% bootstrap confidence intervals). Larger values are better. Each column is a different experiment, on synthetic data generated from a different model family. Rows (top-down) are log-likelihood on the entire sequence (matching the training objective), on the time interval, and on the event type. Within each figure, the models (from left to right) are THP, SAHP, NHP, and A-NHP. The red dashed line represents the log-likelihood of the true distribution 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.



(a) MIMIC-II

(b) StackOverflow

Figure 3: Evaluation results (smaller is better) with 95% bootstrap confidence intervals on the two real-world datasets, comparing THP, SAHP, and NHP with our A-NHP model. RMSE evaluates the predicted time of the next event (root mean squared error), while error rate evaluates its predicted type given its time.

We then evaluated all these models on following two benchmark real-world datasets.[7]

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.Each patient has a sequence of hospital visit events, and each event records its time stamp and disease diagnosis.

StackOverflow (Leskovec & 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).

On MIMIC-II data (Figure 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 3b), our A-NHP is clearly a winner on 2 out of 3 tasks. Compared to NHP, A-NHP also enjoys a computational

[7]For these datasets, we used the preprocessed versions provided by Mei & Eisner (2017). More details about them can be found in Appendix C.1.2.

advantage, as discussed in sections 1 and 2. Empirically, training an A-NHP only took a fraction of the time that was needed to train an NHP, when sequences are reasonably long. Details can be found in Table 2 of Appendix C.3.

8.2 A-NDTT vs. NDTT

Now we turn to the structured modeling approach presented in section 5. We compare A-NDTT with NDTT on the RoboCup dataset proposed by Mei et al. (2020a). In both cases, we used the NDTT program written by Mei et al. (2020a). 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 the unstructured NHP and A-NHP models on this dataset.

RoboCup (Chen & Mooney, 2008)**.** This dataset logs the actions (e.g., `kick`, `pass`) of robot soccer players in the RoboCup Finals 2001–2004. The ball is frequently transferred between players (by passing or stealing), and this dynamically changes the set of possible event types (e.g., only the ball possessor can kick or pass). There are $K = 528$ event types over all time, but only about 20 of them are possible at any given time. For each prefix of each held-out event sequence, we used minimum Bayes risk to predict the next event's time, and to predict its participant(s) given its time and action type.

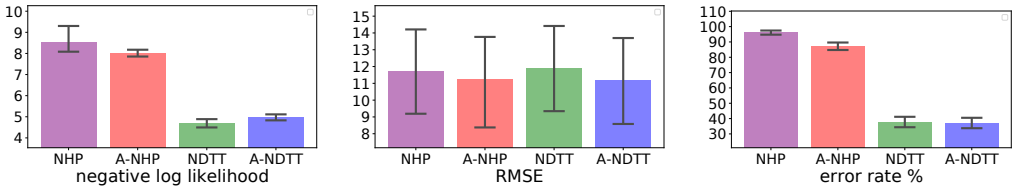


Figure 4: Evaluation results with 95% bootstrap confidence intervals on the RoboCup dataset. Evaluation methods are the same as in Figure 3. Note that the training objective was log-likelihood.

The results are shown in Figure 4. As in section 8.1, we find that A-NHP performs better than NHP on all the evaluation metrics; on log-likelihood and event type prediction, A-NHP is significantly better (paired permutation test, $p < 0.05$). We now inject domain knowledge into both the LSTM and Transformer approaches, by deriving architectures based on the RoboCup NDTT program (which specifies, for example, that only the ball possessor can kick or pass). The resulting models—NDTT and A-NDTT—are substantial and significant improvements. The difference between A-NDTT and NDTT is not significant: perhaps 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.[8] Appendix C.6 includes more results of A-NDTT vs. NDTT broken down by action types.

Additionally, while A-NDTT does not improve the overall accuracy for this particular NDTT program and dataset, it does achieve overall comparable accuracy with a *simpler* and *shallower* architecture (②–③ in section 1). Like other Transformers, the A-NDTT architecture could be trained on a GPU with parallelism, as outlined in Appendix C.4 (future work).

9 Conclusion

We showed how to generalize the Transformer architecture to sequences of discrete events in continuous time. 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.

[8] While NDTT still uses a fixed-dimensional history—① in section 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.

ETHICS STATEMENT

Our paper develops a novel probabilistic model for event sequences. Good probabilistic models make it possible to impute missing events, anticipate possible future events, and react accordingly. They can also be used in exploratory data analysis. In addition to advancing the state-of-the-art in this area, our model may require less energy to train than previous models. We used multiple real-world datasets to evaluate our model and those datasets are all de-identified; see section 8 and Appendix C.1.

However, like many other machine learning models, our model may be applied to unethical ends. For example, its abilities of better fitting data and making more accurate predictions could potentially be used for unwanted tracking of individual behavior, e.g. for advertising or surveillance.

REPRODUCIBILITY STATEMENT

To facilitate the community to reproduce our results, we are committed to releasing our code when the paper is published. For this submission, we made a thorough documentation of our methodology and experimental details in section 8 and Appendices A–C.

REFERENCES

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Boyd, A., Bamler, R., Mandt, S., and Smyth, P. User-dependent neural sequence models for continuous-time event data. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Carbonell, P., `jcdouet`, Alves, H. C., and Tim, A. pyDatalog, 2016.

Ceri, S., Gottlob, G., and Tanca, L. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1989.

Chen, D. L. and Mooney, R. J. Learning to sportscast: A test of grounded language acquisition. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2008.

Choi, E., Du, N., Chen, R., Song, L., and Sun, J. Constructing disease network and temporal progression model via context-sensitive Hawkes process. In *IEEE International Conference on Data Mining (ICDM)*, 2015.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Du, N., Farajtabar, M., Ahmed, A., Smola, A. J., and Song, L. 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*, 2015a.

Du, N., Wang, Y., He, N., Sun, J., and Song, L. Time-sensitive recommendation from recurrent user activities. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2015b.

Du, N., Dai, H., Trivedi, R., Upadhyay, U., Gomez-Rodriguez, M., and Song, L. 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*, 2016.

Etesami, J., Kiyavash, N., Zhang, K., and Singhal, K. Learning network of multivariate Hawkes processes: A time series approach. *arXiv preprint arXiv:1603.04319*, 2016.

Guo, F., Blundell, C., Wallach, H., and Heller, K. The Bayesian echo chamber: Modeling social influence via linguistic accommodation. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, 2015.

Hawkes, A. G. Spectra of some self-exciting and mutually exciting point processes. *Biometrika*, 1971.

He, P., Liu, X., Gao, J., and Chen, W. DeBERTa: Decoding-enhanced bert with disentangled attention. *arXiv preprint arXiv:2006.03654*, 2020.

He, X., Rekatsinas, T., Foulds, J., Getoor, L., and Liu, Y. HawkesTopic: A joint model for network inference and topic modeling from text-based cascades. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 1997.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

Kitaev, N. and Klein, D. Constituency parsing with a self-attentive encoder. In *Proceedings of the Association for Computational Linguistics*, 2018.

Lee, J., Scott, D. J., Villarroel, M., Clifford, G. D., Saeed, M., and Mark, R. G. Open-access MIMIC-II database for intensive care research. In *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 2011.

Leskovec, J. and Krevl, A. SNAP Datasets: Stanford large network dataset collection, 2014.

Liniger, T. J. *Multivariate Hawkes processes*. Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 18403, 2009.

Lukasik, M., Srijith, P. K., Vu, D., Bontcheva, K., Zubiaga, A., and Cohn, T. 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)*, 2016.

Mei, H. and Eisner, J. The neural Hawkes process: A neurally self-modulating multivariate point process. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Mei, H., Qin, G., and Eisner, J. Imputing missing events in continuous-time event streams. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2019.

Mei, H., Qin, G., Xu, M., and Eisner, J. Neural datalog through time: Informed temporal modeling via logical specification. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2020a.

Mei, H., Wan, T., and Eisner, J. Noise-contrastive estimation for multivariate point processes. 2020b.

Omi, T., Ueda, N., and Aihara, K. Fully neural network based model for general temporal point processes. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in PyTorch. 2017.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019.

Rao, R., Meier, J., Sercu, T., Ovchinnikov, S., , and Rives, A. Transformer protein language models are unsupervised structure learners. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

Shchur, O., Biloš, M., and Günnemann, S. Intensity-free learning of temporal point processes. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Xiao, S., Yan, J., Farajtabar, M., Song, L., Yang, X., and Zha, H. Joint modeling of event sequence and time series with attentional twin recurrent neural networks. *arXiv preprint arXiv:1703.08524*, 2017a.

Xiao, S., Yan, J., Yang, X., Zha, H., and Chu, S. Modeling the intensity function of point process via recurrent neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2017b.

Xu, H., Farajtabar, M., and Zha, H. Learning Granger causality for Hawkes processes. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2016.

Zhang, Q., Lipani, A., Kirnap, O., and Yilmaz, E. Self-attentive Hawkes process. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2020a.

Zhang, W., Panum, T. K., Jha, S., Chalasani, P., and Page, D. CAUSE: Learning Granger causality from event sequences using attribution methods. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2020b.

Zuo, S., Jiang, H., Li, Z., Zhao, T., and Zha, H. Transformer Hawkes process. In *International Conference on Machine Learning*, pp. 11692–11702. PMLR, 2020.

Appendices

A0 PARAMETER DIMENSIONALITY SPECIFICATION FOR A-NDTT

In this section we discuss the dimensionality of the fact embeddings $[\![h]\!](t)$ in section 5.

The **type** of a fact in the database is given by its functor (`forum`, `member`, `create`, etc.). All facts of the same type have embedding vectors of the same dimensionality, and these dimensionalities are declared by the NDTT program.

This is enough to determine the dimensions of the parameter matrices associated with the deduction rules (Mei et al., 2020a). How about the add rules, however? The form of equation (9) implies that the value vectors $\mathbf{v}_r^{(\ell)}$ for add rule r have the same dimensionality as the embedding of the head of r. The key and query vectors for rule r (as used in equation (2)) can share this dimensionality by default, although we are free to override this and specify a different dimensionality for them. The foregoing choices determine the dimensions of the parameter matrices $\mathbf{V}_r^{(\ell)}, \mathbf{K}_r^{(\ell)}, \mathbf{Q}_r^{(\ell)}$ associated with rule r.

A final wrinkle is that while the add rule $h \leftarrow f, g_1, \ldots, g_n$ mainly allows an h fact to attend to preceding f events, it also allows the h fact to attend to itself. Unfortunately, we cannot extract h's attention key and value using $\mathbf{V}_r^{(\ell)}$ and $\mathbf{K}_r^{(\ell)}$, because h may have a different dimensionality from f. A reasonable approach would be to always use zero vectors for this key and value.[9]

A LIKELIHOOD COMPUTATION DETAILS

In this section we discuss the log-likelihood formulas in section 6.

The log-likelihood formula equation 6 has been derived and discussed in previous work (Hawkes, 1971; Liniger, 2009; Mei & Eisner, 2017). Intuitively, during parameter training, each $\log \lambda_{e_i}(t_i)$ is increased to explain why e_i happened at time t_i while $\int_{t=0}^{T} \lambda(t)\mathrm{d}t$ is decreased to explain why no event of any possible type $e \in \{1, \ldots, E\}$ ever happened at other times. Note that there is no $\log$ under the integral in equation (6). Why? The probability that there was not an event of any type in the infinitesimally wide interval $[t, t + \mathrm{d}t)$ is $1 - \lambda(t)\mathrm{d}t$, whose $\log$ is $-\lambda(t)\mathrm{d}t$.

As discussed in section 6, the integral term in equation (6) is computed using the Monte Carlo approximation detailed by Algorithm 1 of Mei & Eisner (2017), which samples times t.

However, at each sampled time t, that method still requires a summation over all events to obtain $\lambda(t)$. This summation can be expensive when there are many event types. This is not a serious problem for our standalone A-NHP implementation since it can leverage GPU parallelism. But for the general A-NDTT implementation, it is hard to parallelize the $\lambda_k(t)$ computation over k and t. In that case, we use the downsampling trick detailed in Appendix D of Mei et al. (2020a). An alternative would be to replace maximum-likelihood estimation with noise-contrastive estimation, which is quite effective at training NHP and NDTT models (Mei et al., 2020b).

B HOW TO PREDICT EVENTS

Section 8 includes 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.

We evaluate the time prediction with average L_2 loss (yielding a root-mean-squared error, or **RMSE**) and evaluate the argument prediction with average 0-1 loss (yielding an **error rate**).

[9] However, in our present experiments, we did something more complex: we learned special matrices to extract the key and the value from $h@t$. These matrices were shared by all rules with the same head type, which was only possible because our experimental design in Appendix C.3 used keys of the same dimensionality for all those rules.

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 1: Statistics of each dataset.

Following Mei & Eisner (2017), we use the minimum Bayes risk (MBR) principle to predict the time and type with lowest expected loss. For completeness, we repeat the general recipe in this section.

For the ith event, its time t_i has density $p_i(t) = \lambda(t) \exp(-\int_{t_{i-1}}^{t} \lambda(t')\mathrm{d}t')$. We choose $\int_{t_{i-1}}^{\infty} t p_i(t)\mathrm{d}t$ as the time prediction because it has the lowest expected L_2 loss. The integral can be estimated using i.i.d. samples of t_i drawn from $p_i(t)$ as detailed in Mei & Eisner (2017) and Mei et al. (2019).

Given the next event time t_i, we choose the most probable type $\arg\max_e \lambda_e(t_i)$ as the type prediction because it minimizes expected 0-1 loss. In some circumstances, one might also like to predict the most probable type out of a *restricted* set $\mathcal{E}' \subsetneq \{1, \ldots, E\}$. This allows one to answer questions like "If we know that some event of the form `message(eve,T)` happened at time t_i, then what was the forum `T`, given all past events?" The answer will simply be $\arg\max_{e \in \mathcal{E}'} \lambda_e(t_i)$.

C EXPERIMENTAL DETAILS

C.1 DATASET DETAILS

Table 1 shows statistics about each dataset that we use in this paper.

C.1.1 SYNTHETIC DATA DETAILS

In this experiment, we draw data from randomly initialized NHP, A-NHP, SAHP, and THP. For all of them, we take the number of event types to be $E = 10$. For NHP, the dimensions of event embeddings and hidden states are all 32; for A-NHP, the number of layers (L in our paper) is 2, and the dimensions of time embeddings and event embeddings are 32; for SAHP, the number of layers is 4, and the dimension of hidden states is 32; for THP, the number of layer is 7, and the dimension of hidden states is 32.

For each model, we draw 800, 100, and 100 sequences for training, validation and testing, respectively. For each sequence, the sequence length is drawn from Uniform$(49, 99)$.

C.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 paper.

For RoboCup, we used the version processed by Chen & Mooney (2008); please refer to their paper for more details (e.g., data description, processing method, etc)

C.2 IMPLEMENTATION DETAILS

For NHP, our implementation is based on the public Github repositories at `https://github.com/HMEIatJHU/neurawkes` (Mei & Eisner (2017), with MIT License) and `https://github.com/HMEIatJHU/neural-hawkes-particle-smoothing` (Mei et al. (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.

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 2: Training time of NHP and A-NHP for experiments in section 8.1.

For NDTT, we used the public Github repository at `https://github.com/HMEIatJHU/neural-datalog-through-time` (Mei et al. (2020a), with MIT License). We built A-NDTT upon NDTT.

For THP, we used the public Github repository at `https://github.com/SimiaoZuo/Transformer-Hawkes-Process` (Zuo et al. (2020), no license specified).

For SAHP, we used the public Github repository at `https://github.com/QiangAIResearcher/sahp_repo` (Zhang et al. (2020a), no license specified).

C.3 TRAINING DETAILS

For each model in section 8, we had to specify various dimensionalities. For simplicity, we used a single hyperparameter D and took all vectors to be in $\mathbb{R}^D$. This includes the state vectors of NHP, the fact embeddings of NDTT and A-NDTT, and the query, key, and value vectors for the models with attention mechanisms (THP, SAHP, A-NHP, and A-NDTT). For the models with attention mechanisms, we also had to choose the number of layers L.

We tuned these hyperparameters for each combination of model, dataset, and training size (e.g., each bar in Figures 2–4), always choosing the combination of D 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 & Ba, 2015) with its default settings. We performed early stopping based on log-likelihood on the held-out dev set.

In our preliminary experiments, we also tried adding feed-forward layers and layer normalization (Ba et al., 2016) as in the original Transformer architecture (Vaswani et al., 2017), but they didn't help.

For the experiments in section 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 2 shows their training time per sequence on each dataset.

We run our NDTT and A-NDTT models only on CPUs. This follows Mei et al. (2020a), who 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 Appendix F.2 of Mei et al. (2020a); the A-NDTT program for A-NHP can be found in Appendix C.5.1 below. The training time is 66 and 95 seconds per sequence for NHP and A-NHP, respectively.

C.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 interval $(0, t_1], (t_1, t_2], \ldots, (t_{i-1}, t_I], (t_I, T]$.

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 add them (section 5). Furthermore, let $\mathcal{T}$ be the set of times consisting of $\{t_1, \ldots, t_I\}$ together with the times t that are sampled for the Monte Carlo integral (Appendix A).

A computation graph of size $O(|\mathcal{R}| \cdot I)$ can now be constructed, as illustrated in Figure 1b, to compute the embeddings $[\![h]\!](t)$ of all facts $h \in \mathcal{F}$ at all times $t \in \mathcal{T}$. The layer-ℓ embeddings at time $t \in \mathcal{T}$ depend on the layer-$(\ell - 1)$ embeddings at times $t_i \leq t$, according to the add 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* $t \in \mathcal{T}$ (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 $r \in \mathcal{R}$ can also be handled in parallel, since they use the same r-specific parameters. Thus, a GPU can be effective for this phase. The computation of $\boxed{h}_r^{(\ell)}(t)$ in equation (9) must take care to limit its attention to just those earlier times when an event occurred that added h via rule r, and the computation of $[h]^{(\ell)}(t)$ in equation (9) 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.

C.5 A-NDTT Programs

C.5.1 A-NHP Datalog Program

In this section, we show how to specify A-NHP (section 3) by an A-NDTT program:

```
1  is_type(1).
      ⋮
2  is_type(N).
3  :- embed(world, 8).
4  :- event(e, 8).
5  e(N) :- world, is_type(N) :: prob(N).
6  world <- init.
7  world <- e(N).
```

where the embedding of `world` at time t attends to all past events via rule 7, and is used to derive the embeddings and intensities of all of the events $\underline{\mathtt{e}}(1), \ldots, \underline{\mathtt{e}}(N)$ at time t, via rule 5. 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 mentioned in footnote 1.

This program is simpler than the NDTT program that specifies the NHP model (see Appendix F.2 of Mei et al. (2020a)). In that program, rule 7 will be replaced by a slightly fancier version, namely `world <- e(N), world`. That is because `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 ("attention is all you need").

C.6 More Results

The performance of A-NDTT and NDTT is not always comparable for specific action types, as shown in Figure 5 and Figure 6. In terms of data fitting (left figure), A-NDTT is significantly better at the `kickoff` events while NDTT is better at the others. For time prediction (middle figure), A-NDTT is significantly better at the `goal`, `kickoff`, and `pass` events, but the differences for the other action types are not significant. For action participant prediction (right figure), A-NDTT is significantly better at the `kickoff` events while there is no difference for the others; both do perfectly well at the `goal` and `kick` events such that their dots overlap at the origin.

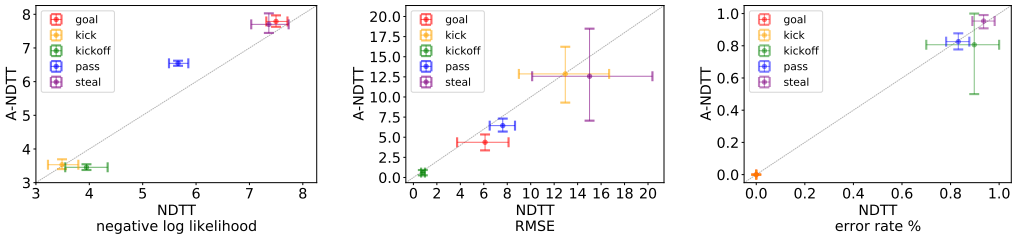


Figure 5: Results of NDTT and A-NDTT in Figure 4 broken down by action types.

In Figure 6, we redraw Figure 5 by training A-NDTT with different random seeds. The figures look very similar, thus the per-action differences between NDTT and A-NDTT are fairly consistent.

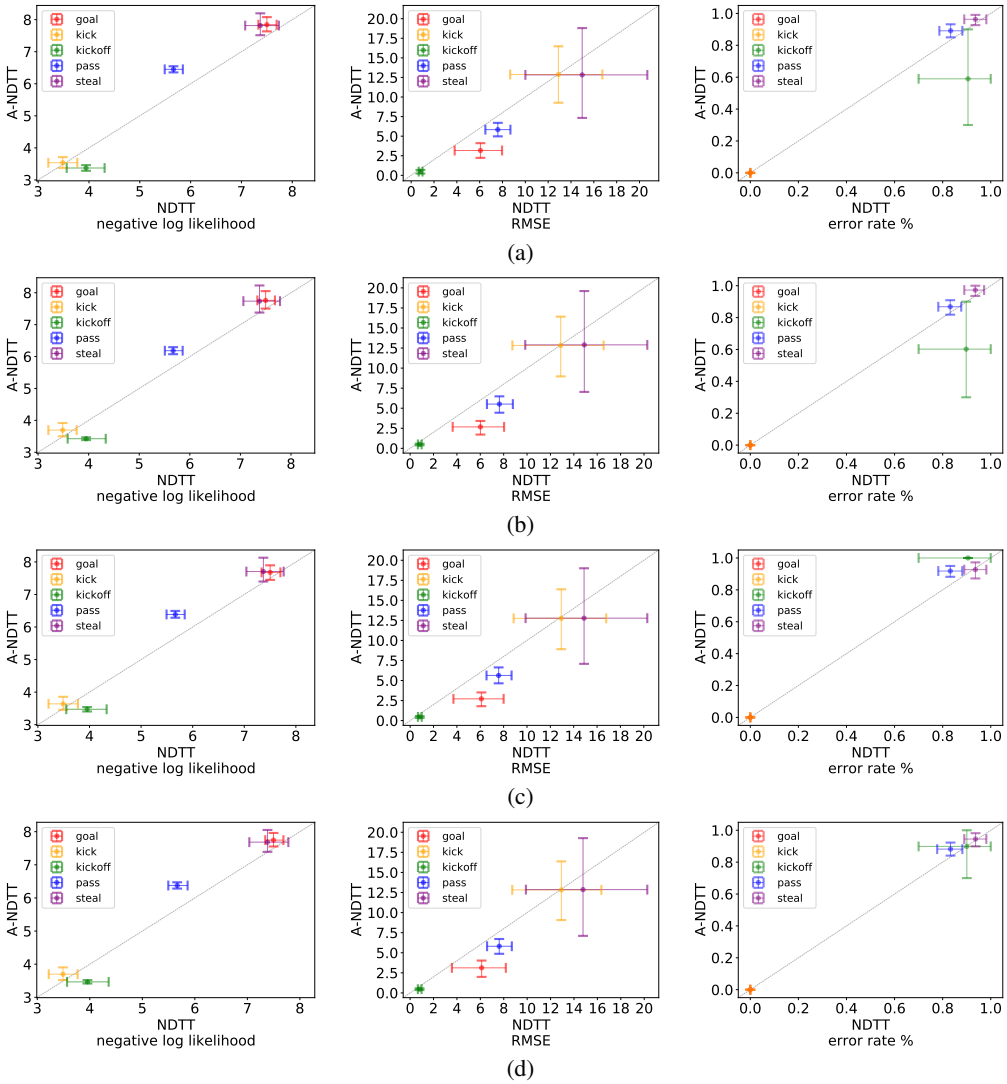


Figure 6: Redraw Figure 5 with different random seeds