Neural Datalog Through Time:
Informed Temporal Modeling via Logical Specification

Hongyuan Mei [1] **Guanghui Qin** [1] **Minjie Xu** [2] **Jason Eisner** [1]

Abstract

Learning how to predict future events from patterns of past events is difficult when the set of possible event types is large. Training an unrestricted neural model might overfit to spurious patterns. To exploit domain-specific knowledge of how past events might affect an event's present probability, 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 Datalog programs improve prediction by encoding appropriate domain knowledge in their architecture.

1. Introduction

Temporal sequences are abundant in applied machine learning. A common task is to predict the future from the past or to impute other missing events. Often this is done by fitting a generative probability model. For evenly spaced sequences, historically popular generative models have included hidden Markov models and discrete-time linear dynamical systems, with more recent interest in recurrent neural network models such as LSTMs. For irregularly spaced sequences, a good starting point is the Hawkes process (a self-exciting temporal point process) and its many variants, including neuralized versions based on LSTMs.

Under any of these models, each event e_i updates the state of the system from $\mathbf{s}_i$ to $\mathbf{s}_{i+1}$, which then deter-

mines 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.

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.

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.)

[1]Computer Science Dept., Johns Hopkins Univ. [2]Bloomberg LP. Correspondence to: Hongyuan Mei <hmei@cs.jhu.edu>.

Since the database defines possible events and is updated by the event that happens, it already resembles the system state 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 define the topology of a deep recurrent neural architecture, which can be trained via back-propagation through time (Williams & 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.

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 & Taskar, 2007; Richardson & 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 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 (Mei & Eisner, 2017) 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 domain-specific models for irregularly spaced event sequences in continuous time, and demonstrate that their structure improves their ability to predict held-out data.

2. 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, §2 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 §3.

2.1. Datalog

We adapt our notation from Datalog (Ceri et al., 1989), where one can write **deductive rules** of the form

$$head \text{ :- } condit_1, \ldots, condit_N. \qquad (1)$$

Such a rule states that the head is true provided that the conditions are all true.[1] 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 $[\![h]\!] \in \{\text{true}, \text{null}\}$ to denote the semantic value of atom h, where $[\![h]\!] = \text{true}$ iff h is a fact.

2.2. 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 rule 3:

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

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

With this extension, $[\![h]\!] \in \mathbb{R}^{D_h} \cup \{\text{null}\}$, 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 $D_h = 8$.[2]

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 rule 4, $[\![\text{rel(eve,adam)}]\!]$ 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 §3.1 below.

---

[1] Appendix A.2 discusses an extension to negated conditions.

[2] 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.

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 rule 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 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 §3.1.

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 et al., 2015), and DAG encoders (Goller & Kuchler, 1996; Le & 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)`⟧. Rule 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.[3]

We currently consider it to be a model specification error if any atom h participates in its own proof, leading to a circular definition of ⟦h⟧. This would happen in rules 9–10 only if `parent` were bizarrely defined to make some cursed person their own ancestor. Appendix A.1 discusses extensions that would define ⟦h⟧ even in these cyclic cases.

2.3. Datalog Through Time

For temporal modeling, we use atoms such as <u>help</u>`(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 2), but as orange rather than blue.

We extend Datalog with **update rules** so that whenever a <u>help</u>`(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.[4] The ! symbol means "not":

```
12│ !grateful(Y,X) ⇐ harm(X,Y).
```

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

$$head \Leftarrow event, condit_1, ..., condit_N. \quad (2a)$$

$$!head \Leftarrow event, condit_1, ..., condit_N. \quad (2b)$$

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 :- 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 (e.g., Mei & Eisner, 2017).

2.4. Neural Datalog Through Time

In §2.2, 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 rule 11, the new embedding ⟦`grateful(Y,X)`⟧ is computed—using parameters associated with rule 11—from the embeddings of <u>help</u>`(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 rule 11 to include `person(Y)` as a condition.) Each time X helps Y again, ⟦`grateful(Y,X)`⟧ is further updated by rule 11, so this gratitude vector records the *history* of help. The updates are LSTM-like (see §3.3 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 (1) says that ⟦$head$⟧(t) depends parametrically on $\{⟦condit_i⟧(t) : 1 \leq i \leq N\}$. A ⇐ rule as in equation (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[5] ⟦$head$⟧(s) along with ⟦$event$⟧(s), $\{⟦condit_i⟧(s) : 1 \leq i \leq N\}$, and the elapsed time $t - s$. We will detail the parametric formulas in §3.3.

Thus, ⟦$head$⟧(t) depends via :- rules on $head$'s *provenance* in the database at time t, and depends via ⇐ rules

---

[3]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 rule 4 pooled over multiple topics.

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

[5]More precisely, it depends on the LSTM cells that contributed to that value, as we will see in §3.3.

on its *experience* of events at strictly earlier times.[6] 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.

2.5. 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 $\llbracket\texttt{help}(\texttt{X},\texttt{Y})\rrbracket \in \mathbb{R}^8$. This is what the underlined functor really indicates.

At times s when `help(X,Y)` is not a fact, the semantic value $\llbracket\texttt{help}(\texttt{X},\texttt{Y})\rrbracket(s)$ 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 and probability are computed deterministically from $\llbracket\texttt{rel}(\texttt{X},\texttt{Y})\rrbracket$ using parameters associated with rule 14, as detailed in §3.2.

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 events. Notice that in our approach (recall §1), the *draws* from the distribution over possible events are stochastic, but the resulting *updates* to that distribution are deterministic—just as in a recurrent neural network language model (Mikolov et al., 2010; Sundermeyer et al., 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)`.

2.6. 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 rule 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.

---

[6]See §3.3 for the precise interaction of :- and ← rules.

2.7. 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,[7] 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 §3.2 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 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. Appendix A.3 discusses potential extensions to handle these cases.

3. Formulas Associated With Rules

3.1. Neural Datalog

Recall from §2.1 that if h is a fact, it is provable by at least one :-- rule in at least one way. For neural Datalog (§2.2), we then choose to define the embedding $[\![h]\!] \neq \mathsf{null}$ as

$$[\![h]\!] \;\overset{\text{def}}{=}\; \tanh\Big(\sum_r [h]_r^{\,\text{:--}}\Big) \;\in (-1,1)^{D_h} \qquad (3)$$

where $[h]_r^{\,\text{:--}}$ represents the **contribution** of the rth rule of the Datalog program. For example, $[\![\text{opinion}(\text{eve},\text{apples})]\!]$ receives non-zero contributions from *both* rule 2 and rule 6.[8] For a given Y, $[\![\text{cursed}(\text{Y})]\!]$ may receive a non-zero contribution from rule 9, rule 10, or neither, according to whether Y is cain himself, a descendant of cain, or neither.

The contribution $[h]_r^{\,\text{:--}}$ has been pooled over all the ways (if any) that the rth rule proves h. For example, for any entity

---

[7]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.

[8]Recall that we renamed `likes` in rule 2 to `opinion`.

Y, $[\text{cursed}(\text{Y})]_{10}^{\,\text{:--}}$ needs to compute the *aggregate* effect of the curses that Y inherits through *all* of Y's cursed parents X in rule 10. Similarly, $[\text{rel}(\text{X},\text{Y})]_4^{\,\text{:--}}$ computes the aggregate effect on the relationship from *all* of X and Y's shared interests U in rule 4. Recall from §2.1 that a rule with variables represents a collection of ground rules obtained by instantiating those variables. We define its contribution by

$$[h]_r^{\,\text{:--}} \;\overset{\text{def}}{=}\; \bigoplus_{g_1,\ldots,g_N}^{\beta_r} \mathbf{W}_r \underbrace{[1; [\![g_1]\!]; \ldots; [\![g_N]\!]]}_{\text{concatenation of column vectors}} \;\in \mathbb{R}^{D_h} \qquad (4)$$

where for the summation, we allow $h \;\text{:--}\; g_1, \ldots, g_N$ to range over all instantiations of the rth rule such that the head equals h and $g_1, \ldots, g_N$ are all facts. There are only finitely many such instantiations (see §2.7). $\mathbf{W}_r$ is a conformable parameter matrix associated with the rth rule. (Appendix 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 $\{\mathbf{x}_1, \ldots, \mathbf{x}_M\}$:

$$\bigoplus_m^\beta \mathbf{x}_m \;\overset{\text{def}}{=}\; v^{-1}\Big(\sum_m v(\mathbf{x}_m)\Big) \qquad (5)$$

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), but only the nonlinear $\tanh$ in equation (3).

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

$$v(\mathbf{x}) \;\overset{\text{def}}{=}\; \text{sign}(\mathbf{x})\,|\mathbf{x}|^\beta \qquad (6\text{a})$$

$$v^{-1}(\mathbf{y}) = \text{sign}(\mathbf{y})\,|\mathbf{y}|^{1/\beta} \qquad (6\text{b})$$

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 (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,[9] and use $\bigoplus^{\beta_r}$ in (4).

3.2. 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

---

[9]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.

also a positive probability. We extend our setup by appending an extra row to the matrix $\mathbf{W}_r$ in (4), leading to an extra element in the column vectors $[h]_r^{:\text{-}}$. We then pass only the first D_h elements of $\sum_r [h]_r^{:\text{-}}$ through $\tanh$, obtaining the same $[\![h]\!]$ as equation (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 (§2.4), 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 et al., 2012).

When time is continuous, as in our experiments (§6), we need instantaneous probabilities. We take $\lambda_h(t)$ to be the (Poisson) **intensity** of h at time t: that is, it models the limit as $dt \to 0^+$ of the expected *rate* of h on the interval $[t, t+dt]$ (i.e., the expected number of occurrences of h divided by dt). This follows the setup of the neural Hawkes process (Mei & Eisner, 2017). Also following that paper, we replace $\exp(x) > 0$ in the above definition of λ_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 h.

In both discrete and continuous time, the exact model likelihood (§4) will involve a summation (at each time t) over the finite set of event types (§2.7) that are possible at time t.

Appendix A.5 offers an extension to simultaneous events.

3.3. 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 $\leftarrow$ rules (§2.3). Recall that an LSTM is constructed from **memory cells** that can be increased or decreased as successive inputs arrive.

Every atom h has a **cell block** $\boxed{h} \in \mathbb{R}^{D_h} \cup \{\text{null}\}$. When $\boxed{h} \neq \text{null}$, we augment h's embedding formula (3) to[10]

$$[\![h]\!] \stackrel{\text{def}}{=} \tanh\left(\boxed{h} + \sum_r [h]_r^{:\text{-}}\right) \in (-1, 1)^{D_h} \quad (7)$$

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

At times when $\boxed{h} = \text{null}$, we like to say that h is **docked**. Every atom h is docked initially (at $t = 0$), but may be **launched** through an update of type (2a), which ensures that $\boxed{h} \neq \text{null}$ and thus $[\![h]\!] \neq \text{null}$ by (7). h is subsequently **adrift** (and remains a fact) until it is docked again through an update of type (2b), which sets $\boxed{h} = \text{null}$.

---

[10]Recall from §3.2 that if h is an event, we extend $[\![h]\!]$ with an extra dimension to carry the probability. For equation (7) to work, we must likewise extend $\boxed{h}$ with an extra cell (when $\boxed{h} \neq \text{null}$).

How is $\boxed{h}$ updated by an event (or events[11]) occurring at time s? Suppose the rth rule is an update rule of type (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 mth instantiation, define

$$[h]_{rm}^{\leftarrow} \stackrel{\text{def}}{=} \mathbf{W}_r \underbrace{[1; [\![e]\!]; [\![g_1]\!]; \ldots; [\![g_N]\!]]}_{\text{concatenation of column vectors}} \quad (8)$$

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

Discrete-time setting. Here we treat the update vectors $[h]_{rm}^{\Delta}$ as increments to $\boxed{h}$. To update $\boxed{h}$ from time s to time $t = s + 1$, we pool these increments within and across rules (much as in (3)–(4)) and increment by the result:

$$\boxed{h} \mathrel{+}= \sum_r \bigoplus_m^{\beta_r} [h]_{rm}^{\Delta} \quad (9)$$

We skip the update (9) if h has no update vectors. If we apply (9), 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 (2b) rule (docking).

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

$$\boxed{h}_{\text{new}} = \mathbf{f} \cdot \boxed{h}_{\text{old}} + \mathbf{i} \cdot (2\mathbf{z} - 1) \quad (10)$$

corresponding to an increment

$$\boxed{h} \mathrel{+}= (\mathbf{f} - 1) \cdot \boxed{h} + \mathbf{i} \cdot (2\mathbf{z} - 1) \quad (11)$$

where the forget gates $\mathbf{f}$, input gates $\mathbf{i}$, and inputs $\mathbf{z}$ are all in $(0, 1)^{D_h}$. Thus, we define $[h]_{rm}^{\Delta}$ as the right side of (11) when $(\mathbf{f}; \mathbf{i}; \mathbf{z}) \stackrel{\text{def}}{=} \sigma([h]_{rm}^{\leftarrow})$, with $[h]_{rm}^{\leftarrow} \in \mathbb{R}^{3D_h}$ from (8).

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 (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, (9) pools their effects across instantiations and sums them across rules.

Continuous-time setting. Here we use the continuous-time LSTM as defined by Mei & Eisner (2017), 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

---

[11]If exogeneous events are used (§2.4), then the instantiations in (8) could include multiple events e that occurred at time s.

given by (7), but $\boxed{h}(t)$ in that equation is given by $\boxed{h}$'s parametric functions as most recently updated (at some earlier time $s < t$). Appendix 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 $[h]_{rm}^{\triangle}$ 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 (§3.2)— 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. Training and Inference

Suppose we observe that the events on time interval $[0, T]$ are $e_1, \ldots, e_I$ at respective times $t_1 < \cdots < t_I$. In the *continuous-time* setting, the log-likelihood of the parameters is

$$\ell \overset{\text{def}}{=} \sum_{i=1}^{I} \log \lambda_{e_i}(t_i) - \int_{t=0}^{T} \lambda(t)\, dt \qquad (12)$$

where $\lambda(t) \overset{\text{def}}{=} \sum_{e \in \mathcal{E}(t)} \lambda_e(t)$ and $\mathcal{E}(t)$ is the set of event types that are possible at time t. We can estimate the parameters by locally maximizing ℓ using any stochastic gradient method. Details are given in Appendix D, including Monte Carlo approximations to the integral. In the *discrete-time* setting,[12] the integral is replaced by $\sum_{t=1}^{T} \log \lambda(t)$.

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 E.

5. Related Work

Past work (Sato, 1995; Poole, 2010; Richardson & Domingos, 2006; Raedt et al., 2007; Bárány et al., 2017) has used logic programs to help define probabilistic relational models (Getoor & 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 & Winn, 2008; van de 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 et al., 2014, 2016; Weston et al., 2015; Sukhbaatar et al., 2015; Kumar et al., 2016; Kiddon et al., 2016; Dyer et al., 2016; Lample et al., 2019; Xiao et al., 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; Van der Heijden et al., 2014; Shelton & Ciardo, 2014; Meek, 2014; Bhattacharjya et al., 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 §6, 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 (§2.1) and have derived embeddings (§2.2). ② Updates to the set of possible events (§2.5). ③ Embeddings of entities and relations that reflect selected past events (§2.4 and §2.6).

6. 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. Experimental details are given in Appendix F.

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 §2.

6.1. 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)`.

---

[12]Here each time t has exactly one event (possibly just a <u>none</u> event), as the event probabilities sum to 1. So $I = T$ and $t_i = i$.

(a) $M = 8$ **(b)** $M = 16$
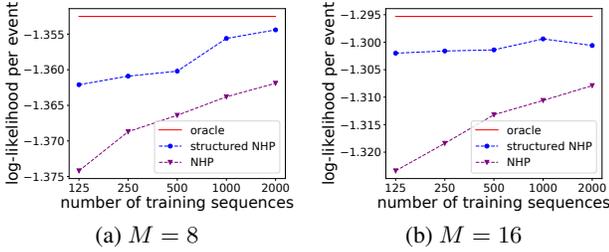
Figure 1. Learning curves of structured model ● and NHP ▼ on sequences drawn from the structured model. The former is significantly better at each training size ($p < 0.01$, paired perm. test).

```
1 | is_process(1).        3 | is_type(1).
    ⋮                         ⋮
2 | is_process(M).        4 | is_type(N).
```

The baseline model is a neural Hawkes process (NHP). It assigns to each event type a separate embedding[13]

```
5 | :- embed(is_event, 8).
6 | is_event(1,1) :- is_process(1), is_type(1).
7 | 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:

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

Note that `e(M,N)` in rule 12 has no embedding, since any such embedding would vary along with the probability. As explained in §3.3, rule 12 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 rules 9–12 with

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

For various small N and M values (see Appendix 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 correctly structured model versus the standard NHP on increasingly long prefixes of the training set, and evaluating them on held-out data. Figure 1 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

---

[13]The list of facts like rules 6 and 7 can be replaced by a single rule if we use "parameter names" as explained in Appendix B.

higher likelihood. In short, it helps to use domain knowledge of which events come from which processes.

6.2. Real-World Domains: IPTV and RoboCup

IPTV Domain (Xu et al., 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 (rule 1) but also track which users have recently watched programs with that tag (rule 2):

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

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

Second, we have a dynamic hard constraint that a program is not available to watch until released:

```
3 | watch(U,P) :- released(P), user(U), profile(P),
                ...
4 | released(P) <- release(P).
```

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

RoboCup Domain (Chen & Mooney, 2008). This dataset logs actions of robot soccer players such as `kick(P)` and `pass(P,Q)` during RoboCup Finals 2001–2004. We made minimum Bayes risk predictions on event time given history, and also of the player P for each event given its time and action type.

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

```
1 | !has_ball(P) <- pass(P,Q). % ball passed from P
2 | 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`):

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

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

```
4 | 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:

(a) IPTV Dataset

(b) RoboCup Dataset
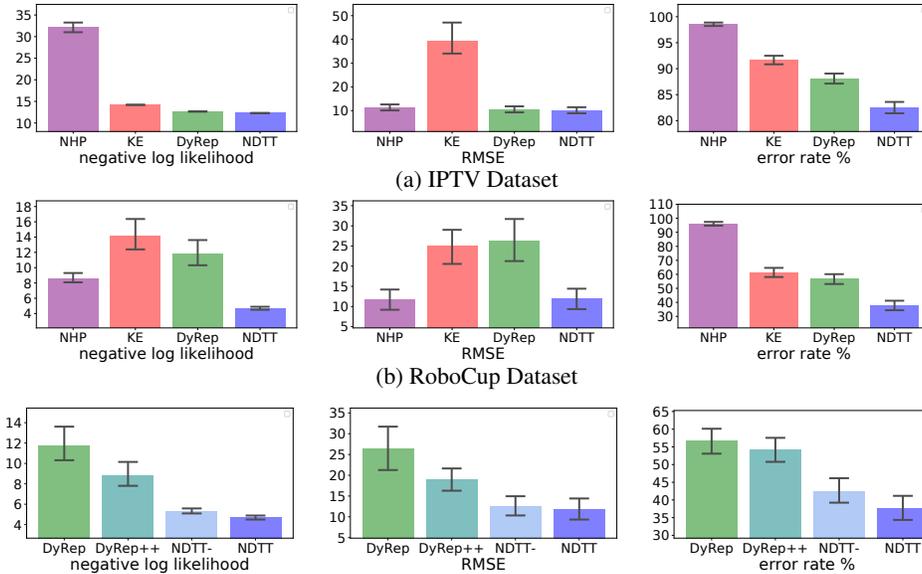
Figure 2. Evaluation results with 95% bootstrap confidence intervals on the real-world datasets of our Datalog program vs. the neural Hawkes process (NHP), KnowEvolve (KE) and DyRep. The RMSE is the root of mean squared error for predicted time. Error rate % denotes the fraction of incorrect predictions of the watched TV program (in IPTV) or the specific player (in RoboCup), given the event time.

Figure 3. Ablation study in the RoboCup domain. "DyRep++" has the same $\Leftarrow$ rules as our structured model and "Ours−" doesn't have `team` states.

```
5 │ 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 Appendix F.4, including our full Datalog programs. The hard logical constraints on possible events are not found in past models.

Results and Analysis. After training, we used minimum Bayes risk (§4) to predict events in test data (details in Appendix E). Figure 2 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(red8)` 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 $\Leftarrow$ rules related to `tag`s.

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.

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 $\Leftarrow$ rules: the first "+" stands for the $\Leftarrow$ rules in which some conditions are not neighbors of the head, and the second "+" stands for the $\Leftarrow$ rules that update event observers.

As Figure 3 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 $\Leftarrow$ rules that see past events.

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. The performance turns out to degrade greatly, indicating that the neural networks had been learning representations that are actually helpful for prediction. See Appendix F.6 for discussion and experiments.

7. Conclusion

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 irregularly spaced event sequences in real-world domains.

Acknowledgments

References

Acar, U. A. and Ley-Wild, R. Self-adjusting computation with Delta ML. In *International School on Advanced Functional Programming*, 2008.

Aldous, D., Ibragimov, I., Jacod, J., and Aldous, D. Exchangeability and related topics. In *École d'Été de Probabilités de Saint-Flour XIII — 1983*, Lecture Notes in Mathematics. 1985.

Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. Learning to compose neural networks for question answering. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2016.

Bárány, V., ten Cate, B., Kimelfeld, B., Olteanu, D., and Vagena, Z. Declarative probabilistic programming with Datalog. *ACM Transactions on Database Systems*, 42 (4):22:1–35, October 2017.

Bhattacharjya, D., Subramanian, D., and Gao, T. Proximal graphical event models. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 8136–8145, 2018.

Blei, D. and Lafferty, J. Correlated topic models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 18, pp. 147–154, 2006.

Blei, D. M. and Frazier, P. I. Distance-dependent Chinese restaurant processes. In *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 87–94, 2010.

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.

Dyer, C., Kuncoro, A., Ballesteros, M., and Smith, N. A. Recurrent neural network grammars. In *NAACL*, pp. 199–209, 2016.

Elman, J. L. Finding structure in time. *Cognitive Science*, 1990.

Filardo, N. W. and Eisner, J. A flexible solver for finite arithmetic circuits. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP)*, 2012.

Fisher, R. A., Corbet, A. S., and Williams, C. The relation between the number of species and the number of individuals in a random sample of an animal population. *J. Animal Ecology*, 12:42–58, 1943.

Getoor, L. and Taskar, B. (eds.). *Introduction to Statistical Relational Learning*. MIT Press, 2007.

Goller, C. and Kuchler, A. Learning task-dependent distributed representations by backpropagation through structure. In *IEEE International Conference on Neural Networks*, volume 1, pp. 347–352, 1996.

Graves, A., Wayne, G., and Danihelka, I. Neural Turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 2016.

Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017a.

Hamilton, W. L., Ying, R., and Leskovec, J. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017b.

Hammer, M. A. *Self-Adjusting Machines*. PhD thesis, Computer Science Department, University of Chicago, 2012.

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

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

Kiddon, C., Zettlemoyer, L., and Choi, Y. Globally coherent text generation with neural checklist models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2016.

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

Kumar, A., Irsoy, O., Ondruska, P., Iyyer, M., Bradbury, J., Gulrajani, I., Zhong, V., Paulus, R., and Socher, R. Ask me anything: Dynamic memory networks for natural language processing. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2016.

Lample, G., Sablayrolles, A., Ranzato, M., Denoyer, L., and Jégou, H. Large memory layers with product keys. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Le, P. and Zuidema, W. 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)*, 2015.

Lin, C., Zhu, H., Gormley, M. R., and Eisner, J. M. Neural finite-state transducers: Beyond rational relations. In *NAACL-HLT*, pp. 272–283, 2019.

Lin, C.-C. and Eisner, J. 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)*, 2018.

Ling, W., Luís, T., Marujo, L., Astudillo, R. F., Amir, S., Dyer, C., Black, A. W., and Trancoso, I. 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)*, 2015.

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

Meek, C. Toward learning graphical and causal process models. In *Uncertainty in Artificial Intelligence Workshop on Causal Inference: Learning and Prediction*, volume 1274, pp. 43–48, 2014.

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.

Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., and Khudanpur, S. Recurrent neural network-based language model. In *Proceedings of the Annual Conference of the International Speech Communication Association (INTERSPEECH)*, 2010.

Minka, T. and Winn, J. Gates: A graphical notation for mixture models. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 1073–1080, 2008.

Natarajan, S., Bui, H. H., Tadepalli, P., Kersting, K., and Wong, W.-K. Logical hierarchical hidden Markov models for modeling user activities. In *Proceedings of the International Conference on Inductive Logic Programming (ICILP)*, 2008.

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.

Poole, D. AILog user manual, version 2.3, 2010.

Raedt, L. D., Kimmig, A., and Toivonen, H. Problog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 2462–2467, 2007.

Richardson, M. and Domingos, P. Markov logic networks. *Machine Learning*, 2006.

Sato, T. A statistical learning method for logic programs with distribution semantics. In *Proceedings of the International Conference on Logic Programming (ICLP)*, pp. 715–729, 1995.

Shelton, C. R. and Ciardo, G. Tutorial on structured continuous-time Markov processes. *Journal of Artificial Intelligence Research*, 51:725–778, 2014.

Socher, R., Huval, B., Manning, C. D., and Ng, A. Y. Semantic compositionality through recursive matrix-vector spaces. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2012.

Sukhbaatar, S., Weston, J., Fergus, R., et al. End-to-end memory networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

Sundermeyer, M., Ney, H., and Schluter, R. LSTM neural networks for language modeling. In *Proceedings of the Annual Conference of the International Speech Communication Association (INTERSPEECH)*, 2012.

Swift, T. and Warren, D. S. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12(1–2):157–187, 2012.

Tai, K. S., Socher, R., and Manning, C. D. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2015.

Tran, K. M., Bisk, Y., Vaswani, A., Marcu, D., and Knight, K. Unsupervised neural hidden Markov models. In *Proceedings of the Workshop on Structured Prediction for NLP*, pp. 63–71, Austin, TX, November 2016.

Trivedi, R., Dai, H., Wang, Y., and Song, L. Know-Evolve: Deep temporal reasoning for dynamic knowledge graphs. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2017.

Trivedi, R., Farajtabar, M., Biswal, P., and Zha, H. DyRep: Learning representations over dynamic graphs. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

van de Meent, J.-W., Paige, B., Yang, H., and Wood, F. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756*, 2018.

Van der Heijden, M., Velikova, M., and Lucas, P. J. Learning Bayesian networks for clinical time series analysis. *Journal of Biomedical Informatics*, 2014.

Wang, Y., Smola, A., Maddix, D. C., Gasthaus, J., Foster, D., and Januschowski, T. Deep factors for forecasting. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2019.

Weston, J., Chopra, S., and Bordes, A. Memory networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

Williams, R. J. and Zipser, D. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

Xiao, C., Teichmann, C., and Arkoudas, K. Grammatical sequence prediction for real-time neural semantic parsing. In *Proceedings of the ACL Workshop on Deep Learning and Formal Languages: Building Bridges*, 2019.

Xu, D., Ruan, C., Korpeoglu, E., Kumar, S., and Achan, K. Inductive representation learning on temporal graphs. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

Xu, H., Luo, D., and Carin, L. Online continuous-time tensor factorization based on pairwise interactive point processes. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

Zhang, X., Lu, L., and Lapata, M. Top-down tree long short-term memory networks. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pp. 310–320, San Diego, California, June 2016.

Appendices

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, and more of them may be supported in future versions.

A.1. Cyclicity

Our embedding definitions in §2.2 and §3.1 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:

```
1  reachable(start).
2  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 $[\![\text{reachable}(V)]\!]$ 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) and (8) are modified to replace each $[\![g_i]\!]$ with $[\![g_i/(S \cup \{h\})]\!]$.[14] As usual, these formulas skip pooling over instantiations where any $[\![\cdot]\!]$ values in the body are null. In particular, this scheme defines the **acyclic embedding** $[\![h/\emptyset]\!]$; the recursive definition terminates because S grows at each recursive step but its size is bounded above (§2.7). The proba-

---

[14]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 paper.

bility of an event e is derived from $\lambda_{e/\emptyset}$, which is computed in the usual way (§3.2) as an extra dimension of the acyclic embedding $[\![e/\emptyset]\!]$.

Forward propagation. This is a more practical approach, used by Hamilton et al. (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 §3.1 and §3.3, using the $[\![g_i]\!]$ values from the previous iteration (also at time t) and the cell block $\boxed{h}$ (determined by events at times $s < t$).

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 §3.1 and §3.3. 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 et al. (2017a)) are well-defined: they depend only on the graph structure, not on any arbitrary ordering of the computations.

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 (1) or (2), each of the conditions $condit_i$ 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) and (8) omits $[\![g_i]\!]$ if $condit_i$ is negated, since then g_i is not a fact and does not have a vector (rather, $[\![g_i]\!] = $ null).

Many dialects of Datalog permit programs with negation. If we allow cycles (Appendix 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 §2, 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 (§3.2) 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).
```

A.3. Infinite Domains

§2.7 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.[15] This allows us to build up an infinite set of atoms from a finite set of initial entities:

```
31 birth(X,Y,child(X,Y)) ← procreate(X,Y).
```

Thus, 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.

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

```
32 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 $\boxed{\texttt{birth(eve,adam,cain)}}$ is updated from an initial value of **0** by equation (10) 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:[16]

```
33 person(Z) :- birth(X,Y,Z).
34 parent(X,Z) :- birth(X,Y,Z).
35 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 further 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).
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 rule 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) (rule 3). That embedding reflects only the number of times X has been ordered previously (rule 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 Appendix A.4 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

---

[15]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. This means that computation at that time may not terminate, and our $\bigoplus^\beta$ operators may have to aggregate over infinite sets (see §2.7).

[16]Somewhat awkwardly, under our design, rule 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.

in which the relative probability of X is also affected by previous orders of dishes Y ≠ X (cf. Blei & Lafferty, 2006) or by the exogenous events at the present time and at times when X was ordered previously (cf. Blei & Frazier, 2010).

Appendix A.5 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 atoms about new anonymous entities.

A.4. Uses of Exogenous Events

The extension to allow exogeneous events was already discussed in the main paper (§2.4). 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 11.) 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 (9) during the intervals between the modeled events that affect those cell blocks.[17]

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 in equation (10) are summed. This architecture is similar to a left-to-right LSTM tagger (cf. Ling et al., 2015; Tran et al., 2016).

A.5. Modeling Multiple Simultaneous Events

§3.2 explained how to model a discrete-time event sequence:

> 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 <u>none</u> can be used to encode "nothing occurred."

The continuous-time setting is similar: almost surely, there are no times t with multiple events. Recall from §3.2 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. Specifcially, multiple events that simultaneously update the same head are pooled within and across rules by equation (9).)

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, then exactly one event e' in g 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 $\llbracket e \rrbracket$ and $\llbracket g \rrbracket$ 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:[18]

$$\text{:- eventgroup}(\textit{functor}, \textit{dimension}). \qquad \text{(13a)}$$
$$\textit{group} \lll \textit{event}, \textit{condit}_1, \ldots, \textit{condit}_N. \quad \text{(13b)}$$
$$\textit{event} \texttt{<=} \textit{group}, \textit{condit}_1, \ldots, \textit{condit}_N. \quad \text{(13c)}$$

`eventgroup` declarations of the form (13a) are used to declare that atoms with a particular functor refer to event groups, similar to `event` declarations. We will display such functors with a double underline.

---

[17]In fact, tick events can *also* be used in the continuous case, if desired (Mei & 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).

[18]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.

Rules of the form (13b) are 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 $\bar{g} \lll \bar{e}, c_1, \ldots, c_N$ for which the c_i are all facts at time t. The embedding of this token pools over all such instantiations:

$$\llbracket g \rrbracket \stackrel{\text{def}}{=} \bigoplus_{c_1,\ldots,c_N}^{\beta_r} \mathbf{W}_r \underbrace{[1; \llbracket e \rrbracket; \llbracket c_1 \rrbracket; \ldots; \llbracket c_N \rrbracket]}_{\text{concatenation of column vectors}} \in \mathbb{R}^{D_g} \quad (14)$$

where all embeddings are evaluated at time t.

Rules of the form (13c) 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 $\bar{e}' \mathrel{<=} \bar{g}, c_1, \ldots, c_N$ for which the c_i are all facts at time t. The embedding of this copy pools over all such instantiations:

$$\llbracket e' \rrbracket \stackrel{\text{def}}{=} \bigoplus_{c_1,\ldots,c_N}^{\beta_r} \mathbf{W}_r \underbrace{[1; \llbracket g \rrbracket; \llbracket c_1 \rrbracket; \ldots; \llbracket c_N \rrbracket]}_{\text{concatenation of column vectors}} \in \mathbb{R}^{D_g} \quad (15)$$

where all embeddings are evaluated at time t.

Since each e' in group g is an event, we compute not only an embedding $\llbracket e' \rrbracket$ but also an unnormalized probability $\lambda_{e'}$, computed just as in §3.2 (using $\exp$ rather than 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 (§4) or sampling (Appendix 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 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 $p(e' \mid e)$ as proportional to $\lambda_{e'}$. The set of possible marks for an event is defined by rules of the form (13) 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,

```
1  :- eventgroup(restaurants, 5).
2  :- event(eat_at, 0).
3  restaurants <<- eat_meal(X).
4  eat_at(Y) <= restaurants, is_restaurant(Y).
5  eat_at(home) <= restaurants.
```

which associates some dependent restaurant Y (or home) with the meal.[19] At the same time, the meal may select a *set* of foods to eat, where each food U[20] is in competition with none[21] to indicate that it may or may not be chosen:

```
6   :- eventgroup(optdish, 7).
7   :- event(eat_dish, 0).
8   :- event(none, 0).
9   optdish(U) <<- eat_meal(X),
        food(U), opinion(X,U).
10  eat_dish(U) <= optdish(U).
11  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 et al., 2016). Observing such sentences in training data would then provide evidence of the underlying embeddings of the events. For example, to generate

---

[19]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 $\llbracket \text{restaurants} \rrbracket$, which depends on $\llbracket \text{eat\_meal(X)} \rrbracket$.

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

[21]The annotation `: 0` in the last line (explained in Appendix 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.

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 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 (Appendix A.4) to trigger one group for each person:

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

This is a group-wise version of rule 14 in the main paper.

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 et al., 2017b; Xu et al., 2020). 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:

```
1  :- embed(node, 8).
2  :- 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,[22] 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).
```

---

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

```
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 depends stochastically on the embeddings of the node and its children, as well as the embedding of the color:

```
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).
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 (2a), (2b), or (13b). 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:

```
36  :- eventgroup(act, 5).
37  act(X)  <<-  sleep(X).
38  act(X)  <<-  help(X,Y), person(Y).
      ⋮
39  person(Y)  <-  act(X), parent(X,Y), person(Y).
40  animal(Y)  <-  act(X), own(X,Y), animal(Y).
```

This means that whenever X takes any action—`sleep`, `help`, etc.—rules 39–40 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 class `help`(eve,adam) has a method that returns its

embedding in $\mathbb{R}^{D_{\text{help}}}$. But rules 39–40 instead view this as an instance of the superclass <u>act</u>(eve), and hence call a method of that superclass to query it for its embedding in $\mathbb{R}^{D_{\text{act}}} = \mathbb{R}^5$, which it computes from its previous embedding via equation (14).

In the above example, the event group is actually empty, as there are no rules of type (13c) 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.

B. Parameter Sharing Details

Throughout §3, the parameters $\mathbf{W}$ and β are indexed by the rule number r. (They appear in equations (4) and (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:

$$head \ : \ beta \ :-$$
$$: \ bias\_vector$$
$$condit_1 \ : \ matrix_1,$$
$$\vdots$$
$$condit_N \ : \ matrix_N.$$

Now β_r in equation (4) is replaced by a scalar parameter named by the atom $beta$. Similarly, the affine transformation matrix $\mathbf{W}_r$ in equation (4) is replaced by a parameter matrix that is constructed by *horizontally* concatenating the vector and matrices named by the atoms $bias\_vector, matrix_1, \ldots, matrix_N$ respectively. For example, $matrix_i$ will have D_{head} rows and D_{condit_i} columns, and will in effect be multiplied by the embedding of the atom that instiatiates $condit_i$.

These parameter annotations with the $:$ symbol are optional (and were not used in the main paper). If any of them is not specified, it is set automatically to be rule- and position-specific: in the rth rule, $beta$ defaults to params(r,beta), $bias\_vector$ defaults to params(r,bias), and $matrix_i$ 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:

```
41  cursed(Y) :- cursed(X),parent(X,Y) :: inherit.
42  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 instantiations of its body.)

For example, we can modify rules 41 and 42 into

```
43  cursed(Y) : descendant(Y) :-
        cursed(X), parent(X,Y) :: inherit(X,Y).
44  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 (Appendix A.3), 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:

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

Here 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 $\Leftarrow$ 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 (8).

For a $\Leftarrow$ 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 (17) in Appendix C below). For example, forcing $\underline{\mathbf{f}} = \mathbf{1}$ and $\underline{\mathbf{i}} = \mathbf{0}$ in equation (18) ensures via equation (19) 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 <u>tick</u> rules

mentioned in footnote 17, for example.) Similarly, forcing $\bar{\mathbf{f}} = \mathbf{1}$ and $\bar{\mathbf{i}} = \mathbf{0}$ in equation (18) ensures via equation (20) 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 (8) (as this column holds bias terms). We have not yet designed a syntax for such declarations.

We can also name the softplus scale parameter τ in §3.2. For example, we can rewrite line 13 of §2.4 as

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

and allow <u>harm</u> to share τ with <u>help</u>:

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

C. Updating Drift Functions in the Continuous-Time LSTM

Here we give the details regarding continuous-time LSTMs, which were omitted from §3.3 owing to space limitations. We follow the design of Mei & Eisner (2017), in which each cell changes endogenously between updates, or "drifts," according to an exponential decay curve:

$$c(t) \stackrel{\text{def}}{=} \bar{c} + (\underline{c} - \bar{c})\exp(-\delta(t - s)) \quad \text{where } t > s \quad (16)$$

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 $c'(t) = \delta \cdot (\bar{c} - \underline{c})$

We 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 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 $\boxed{h}^{\mathbf{c}}, \boxed{h}^{\bar{\mathbf{c}}}, \boxed{h}^{\boldsymbol{\delta}} \in \mathbb{R}^{D_h}$. Now analogously to equation (16), we define the trajectory of the cell block elementwise:

$$\boxed{h}(t) \stackrel{\text{def}}{=} \boxed{h}^{\bar{\mathbf{c}}} + \left(\boxed{h}^{\mathbf{c}} - \boxed{h}^{\bar{\mathbf{c}}}\right)\exp(-\boxed{h}^{\boldsymbol{\delta}} \cdot (t - s)), \tag{17}$$

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 $[h]_{rm}^{\leftarrow} \in$

$\mathbb{R}^{3D_h}$ by evaluating (8) at time s, and set $(\mathbf{f}; \mathbf{i}; \mathbf{z}) = \sigma([h]_{rm}^{\leftarrow})$. In the continuous-time case, we evaluate (8) at time s to obtain $[h]_{rm}^{\leftarrow} \in \mathbb{R}^{7D_h}$ (so $\mathbf{W}_r$ needs to have more rows), and accordingly obtain 7 vectors in $(0, 1)^{D_h}$,

$$(\underline{\mathbf{f}}; \underline{\mathbf{i}}; \underline{\mathbf{z}}; \bar{\mathbf{f}}; \bar{\mathbf{i}}; \bar{\mathbf{z}}; \mathbf{d}) \stackrel{\text{def}}{=} \sigma([h]_{rm}^{\leftarrow}) \tag{18}$$

which we use similarly to equation (11) to define update vectors for the current cell values (time s) and the asymptotic cell values (time ∞), respectively

$$[h]_{rm}^{\Delta\mathbf{c}} \stackrel{\text{def}}{=} (\underline{\mathbf{f}} - 1) \cdot \boxed{h}(s) + \underline{\mathbf{i}} \cdot (2\underline{\mathbf{z}} - 1) \tag{19}$$

$$[h]_{rm}^{\Delta\bar{\mathbf{c}}} \stackrel{\text{def}}{=} (\bar{\mathbf{f}} - 1) \cdot \boxed{h}^{\bar{\mathbf{c}}} + \bar{\mathbf{i}} \cdot (2\bar{\mathbf{z}} - 1) \tag{20}$$

as well as a vector of proposed decay rates:[23]

$$[h]_{rm}^{\boldsymbol{\delta}} \stackrel{\text{def}}{=} \text{softplus}_1(\sigma^{-1}(\mathbf{d})) \in \mathbb{R}_{>0}^{D_h} \tag{21}$$

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 (9)–(11):

$$\boxed{h}^{\mathbf{c}} \stackrel{\text{def}}{=} \boxed{h}(s) + \sum_r \bigoplus_m^{\beta_r} [h]_{rm}^{\Delta\mathbf{c}} \tag{22}$$

$$\boxed{h}^{\bar{\mathbf{c}}} \stackrel{\text{def}}{=} \boxed{h}^{\bar{\mathbf{c}}} + \sum_r \bigoplus_m^{\beta_r} [h]_{rm}^{\Delta\bar{\mathbf{c}}} \tag{23}$$

The special cases mentioned just below the update (9) are also followed for the updates (22)–(23).

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 ith cell, we should obtain the decay rate δ_i by a weighted harmonic mean of the decay rates $([h]_{rm}^{\boldsymbol{\delta}})_i$ 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 $([h]_{rm}^{\Delta\mathbf{c}})_i$ or $([h]_{rm}^{\Delta\bar{\mathbf{c}}})_i$.

Why harmonic mean? Observe that the exponential decay curve (16) 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 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,[24] which seems like a reasonable pooling principle.

---

[23]Equation (21) simply replaces the σ that produced $\mathbf{d}$ with softplus$_1$ (defined in §3.2), since there is no reason to force decay rates into $(0, 1)$.

[24]It is also equivalent to saying that the $(2/3)$-life is a weighted arithmetic mean of proposed $(2/3)$-lives, since equation (16) 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.

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

$$\boxed{h}^{\boldsymbol{\delta}} \stackrel{\text{def}}{=} \left(\frac{\sum_r \sum_m \mathbf{w}_{rm} \cdot ([h]_{rm}^{\boldsymbol{\delta}})^{-1}}{\sum_r \sum_m \mathbf{w}_{rm}} \right)^{-1} \quad (24)$$

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

$$\mathbf{w}_{rm} \stackrel{\text{def}}{=} \left(\bigoplus_{m'}^{\beta_r} \left| [h]_{rm'}^{\Delta\mathbf{c}} \right| \right) \cdot \frac{\left| [h]_{rm}^{\Delta\mathbf{c}} \right|^{\beta_r}}{\sum_{m'} \left| [h]_{rm'}^{\Delta\mathbf{c}} \right|^{\beta_r}}$$

$$+ \left(\bigoplus_{m'}^{\beta_r} \left| [h]_{rm'}^{\Delta\bar{\mathbf{c}}} \right| \right) \cdot \frac{\left| [h]_{rm}^{\Delta\bar{\mathbf{c}}} \right|^{\beta_r}}{\sum_{m'} \left| [h]_{rm'}^{\Delta\bar{\mathbf{c}}} \right|^{\beta_r}}$$

$$+ \left| \boxed{h}^{\bar{\mathbf{c}}} - \boxed{h}^{\mathbf{c}} \right| \quad (25)$$

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 (25) are valid weights to use for the weighted harmonic mean (24):

- $\mathbf{w}_{rm} \geq 0$, because of the use of absolute value.

- $\mathbf{w}_{rm} > 0$ strictly unless $\boxed{h}^{\bar{\mathbf{c}}} = \boxed{h}^{\mathbf{c}}$. Thus, the decay rate $\boxed{h}^{\boldsymbol{\delta}}$ as defined by equation (24) can only be undefined (that is, $\frac{0}{0}$) if $\boxed{h}^{\bar{\mathbf{c}}} = \boxed{h}^{\mathbf{c}}$, in which case that decay rate is irrelevant anyway.

The way to understand the first line of equation (25) is as a heuristic assessment of how much the cell's curve (16) was affected by (r, m) via $[h]_{rm}^{\Delta\mathbf{c}}$'s effect on $\boxed{h}^{\mathbf{c}}$. First of all, $\bigoplus_{m'}^{\beta_r} \left| [h]_{rm'}^{\Delta\mathbf{c}} \right|$ is the pooled magnitude of *all* of the rth rule's attempts to affect $\boxed{h}^{\mathbf{c}}$. Using the absolute value ensures that even if large-magnitude attempts of opposing sign canceled each other out in equation (22), they are still counted here as large attempts, and thus give the rth rule a stronger total voice in determining the decay rate $\boxed{h}^{\boldsymbol{\delta}}$. This pooled magnitude for the rth rule is then partitioned among the attempts (r, m). In particular, the fraction in the first line denotes the portion of the rth rule's pooled effect on $\boxed{h}^{\mathbf{c}}$ that should be heuristically attributed to (r, m) specifically, given the way that equation (22) pooled over all m (recall that this invokes equation (6a)).

Thus, the first line of equation (25) considers the effect of (r, m) on $\mathbf{c}$. 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 $[h]_{rm}^{\Delta\mathbf{c}}$ and $[h]_{rm}^{\Delta\bar{\mathbf{c}}}$ are small compared to the total height

of the curve, namely $\left| \boxed{h}^{\bar{\mathbf{c}}} - \boxed{h}^{\mathbf{c}} \right|$, 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).

D. Likelihood Computation Details

In this section we discuss the log-likelihood formulas in §4.

For the discrete-time setting, the formula simply follows from the fact that the log-probability of event e at time t was defined to be $\log(\lambda_e(t)/\lambda(t))$.

The log-likelihood formula (12) for the continuous-time case 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 $\int_{t=0}^{T} \lambda(t) dt$ is decreased to explain why no event of any possible type $e \in \mathcal{E}(t)$ ever happened at other times. Note that there is no $\log$ under the integral in equation (12), in contrast to the discrete-time setting.

As discussed in §4, the integral term in equation (12) 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. 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

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

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 & 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 & Ley-Wild, 2008; Hammer, 2012; Filardo & Eisner, 2012).

E. How to Predict Events

Figures 2 and 4 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.

These figures 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**).

To carry out the predictions, we follow Mei & Eisner (2017) and use the minimum Bayes risk (MBR) principle to predict the time and type with lowest expected loss. To predict 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).

- Since we are given the next event time t_i when predicting the type e_i,[25] the most likely type is simply $\arg\max_{e \in \mathcal{E}(t_i)} \lambda_e(t_i)$.

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 $[\![\texttt{rel(eve,adam)}]\!](t_i) = \texttt{null}$ (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 $\arg\max_{e \in \mathcal{E}'(t_i)} \lambda_e(t_i)$.

As another extension, Mei et al. (2019) show how to predict missing events in a neural Hawkes process conditioned on partial observations of both past and future events. They used a particle smoothing technique that had previously been used for discrete-time neural sequence models (Lin

---

[25] Mei & Eisner (2017) also give the MBR prediction rule for predicting e_i *without* knowledge of its time t_i.

& Eisner, 2018). This technique could also be extended to neural Datalog through time (NDTT):

- In **particle filtering**, each particle specifies a hypothesized complete history of past events (both observed and missing). In our setting, this provides enough information to determine the set of possible events $\mathcal{E}(t)$ at time t, along with their embeddings and intensities.

- **Neural particle smoothing** is an extension where the guess of the next event is also conditioned on the sequence of future events (observed only), using a learned neural encoding of that sequence. In our setting, it is not clear what embeddings to use for the future events, as we do not in general have static embeddings for our event types, and their dynamic embeddings cannot yet be computed at time t. We would want to learn a compositional encoding of future events that at least respects their structured descriptions (e.g., `help(eve,adam)`), and possibly also draws on the NDTT program and its parameters in some way. We leave this design to future work.

F. Experimental Details

F.1. Dataset Statistics

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

F.2. Synthetic Dataset Details

We synthesized data by sampling event sequences from the structured NHP specified by our Datalog program in §6.1. We chose $N = 4$ and $M = 4, 8, 16$ and thus end up with three different datasets.

For each M, we set the sequence length $I = 21$ and then used the thinning algorithm (Mei & Eisner, 2017; Mei et al., 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 1 and left out the plot for $M = 4$ because it is boringly similar.

For the unstructured NHP baseline, the program given in §6.1 is not quite accurate. To exactly match the architecture of Mei & Eisner (2017), we have to use the notation of Appendix B to ensure that each of the MN event types has its own parameters:

```
1 is_process(1).          3 is_type(1).
       ⋮                        ⋮
2 is_process(M).          4 is_type(N).
5 :- embed(world, 8).
```

DATASET	$|\mathcal{K}|$	# 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 1. Statistics of each dataset.

```
6  :- embed(is_event, 8).
7  :- event(e, 0).
8  is_event(M,N)
       :- is_process(M), is_type(N)
       :: emb(M,N).
9  e(M,N)
       :- world, is_process(M), is_type(N)
       :: dep(M,N).
10 world <- init.
11 world <- e(M,N), is_event(M,N), world.
```

F.3. IPTV Dataset Details

We used the events of days 1–200, days 201–220, and days 221–240 as training, dev and test data respectively, and saved the rest for future experiments. For dev and test, we evaluate the model's predictive power on the held-out dev and test events respectively. However, when predicting an event, the model is still allowed to condition on the *full* history of that event (starting from day 1). This is needed to determine the facts in the database, their embeddings, and the event intensities.

The time unit in this domain is 1 minute and thus in the graph for time prediction, an error of 1.5 (for example) means an error of 1.5 minutes.

Each event type has the form `watch(U,P)` meaning that user U `watch`es TV program P. The relational facts and rules about users and programs are specified as follows:

```
1  is_user(u1).
      .
      .
      .
2  is_user(u1000).
3  is_tag(adventure).
      .
      .
      .
4  is_tag(war).
5  is_program(p1).
6  has_tag(p1,comedy).
      .
      .
      .
      .
      .
      .
7  is_program(p49).
8  has_tag(p49,romance).
      .
      .
      .
```

In principle, the atoms are allowed to have different dimen-

sions. But we make all of them to be the same in our experiments so as not to have too many hyperparameters to tune: see Appendix F.5 for tuning details. Supposing that the dimension is 8, our Datalog program declares it as follows:

```
9  :- embed(user, 8).
10 :- embed(availprog, 8).
11 :- embed(profile, 8).
12 :- embed(released, 0).
13 :- embed(watchhistory, 8).
14 :- embed(tag, 8).
15 :- event(watch, 8).
```

where `watch` has an extra dimension for its intensity.

Here are the `:-` rules:

```
16 user(U) :- is_user(U).
17 availprog(P)
       :- profile(P), released(P),
          watchhistory(P).
18 profile(P) :- tag(T), has_tag(P,T).
19 profile(P) :- is_program(P).
20 release(P) :- is_program(P).
21 watch(U,P) :- user(U), availprog(P).
```

and `<-` rules:

```
22 user(U) <- init, is_user(U).
23 tag(T) <- init, is_tag(T).
24 released(P) <- release(P).
25 watchhistory(P) <- release(P).
26 user(U) <- watch(U,P), user(U), availprog(P).
27 watchhistory(P)
       <- watch(U,P), user(U), availprog(P).
28 tag(T)
       <- watch(U,P), user(U), availprog(P),
          tag(T), has_tag(P,T).
```

We also implement the architectures of Know-Evolve and DyRep as Datalog programs. The relational facts and rules (lines are the same for all three models. The Know-Evolve program continues as follows:

```
29 :- embed(user, 8).
30 :- embed(availprog, 8).
31 :- embed(watch_emb, 8).
32 :- event(watch, 0).
```

Here are the `:-` rules:

```
33 watch_emb(U,P) :- is_user(U), is_program(P).
34 watch(U,P) :- user(U), availprog(P).
35 proj_local(U,P)
       :- user(U), is_program(P) :: proj.
36 proj_local(U,P)
```

```
             :- availprog(P), is_user(U) :: proj.
37 | proj_obj(U,P)
             :- user(U), is_program(P) :: proj.
38 | proj_obj(U,P)
             :- availprog(P), is_user(U) :: proj.
```

Note that the static embedding is independent of U or P because the parameter name doesn't consider U or P and functors is_user and is_program have dimension 0.

Here are the ← rules:

```
39 | user(U) ← init, is_user(U).
40 | availprog(P) ← init, is_program(P).
41 | user(U)
           ← watch(U,P), watch_emb(U,P),
             proj_local(U,P).
42 | availprog(P)
           ← watch(U,P), watch_emb(U,P),
             proj_obj(U,P).
```

Note that Know-Evolve doesn't consider the release of new TV programs but only behaves as if all the TV programs are available to watch at any time. So it doesn't impose any temporal hard constraints on the intensities.

The DyRep program also makes the intensity of each event depend on its participants, so the :- rules about intensities are the same as in Know-Evolve. The standard DyRep model doesn't have embeddings for event types but our reimplementation followed their appendices to assign a static embedding to each event type. The way these static embeddings are computed is defined to be the same as in Know-Evolve. However, for the ← rules, DyRep considers the neighbors of participants while updating them. In this domain, we consider the tags of a program P as its neighbors. This design is meaningful because it offers a way for DyRep to consider the tags of each program P when updating P. The declarations, :- rules, and ← rules of DyRep are as follows:

```
43 | :- embed(user, 8).
44 | :- embed(availprog, 8).
45 | :- embed(tag, 8).
46 | :- embed(watch_emb, 8).
47 | :- event(watch, 0).
48 | watch_emb(U,P) :- is_user(U), is_program(P).
49 | watch(U,P) :- user(U), availprog(P).
50 | user(U) ← init, is_user(U).
51 | tag(T) ← init, is_tag(T).
52 | availprog(P) ← release(P).
53 | user(U) ← watch(U,P), watch_emb(U,P), user(U).
54 | availprog(P)
           ← watch(U,P), watch_emb(U,P),
             availprog(P).
55 | availprog(P)
           ← watch(U,P), tag(T), has_tag(P,T).
```

Note that DyRep handles the addition of facts—i.e., new TV programs being released—thus can have dynamic hard logic constraints on intensities as our structured model does.

F.4. RoboCup Dataset Details

We use Final 2001 and 2002, Final 2003 and Final 2004 as training, dev and test data respectively. The time unit in this domain is 1 second, and thus in the graph for time prediction, an error of 1.5 (for example) means an error of 1.5 seconds.

Each event type has one of the forms:

- kickoff(P) means that player P kicks the ball off which usually happens at the beginning of the game, after a pause or after someone scores;
- kick(P) means that player P kicks the ball and still keeps its possession;
- goal(P) means that player P scores;
- pass(P,Q) means that P passes ball to teammate Q;
- steal(Q,P) means that the ball is turned over from P by opponent Q.

The relational facts and rules about players and teams are specified as follows:

```
1  | is_team(a).    % team a
2  | is_team(b).    % team b
3  | is_player(a1).
4  | in_team(a1,a).   % player a1 is in team a
       ⋮
5  | is_player(b11).
6  | in_team(b11,b).   % player b11 is in team b
7  | not_eq(a,b).
8  | not_eq(b,a).   % teams are different
9  | not_eq(a1,a2).   % a1 and a2 are different
       ⋮
10 | not_eq(b11,b10).
11 | teammate(P,Q) :- in_team(P,T), in_team(Q,T),
        is_team(T), not_eq(P,Q).
12 | opponent(P,Q) :- in_team(P,T), in_team(Q,S),
        is_team(T), is_team(S), not_eq(T,S).
```

In principle, the functors are allowed to have different dimensions. But we make all of them to be the same in our experiments so as not to have too many hyperparameters to tune: see Appendix F.5 for details. Suppose that the dimension is 8, our Datalog program declares it as follows:

```
13 | :- embed(player, 8).
14 | :- embed(team, 8).
15 | :- event(kickoff, 8).
16 | :- event(kick, 8).
17 | :- event(goal, 8).
18 | :- event(pass, 8).
19 | :- event(steal, 8).
```

Here are the :- rules:

```
20 | kickoff(P)
           :- player(P), team(T), team(S),
              in_team(P,T), not_eq(T,S), has_ball(T).
            % team T has ball
21 | kick(P)
           :- player(P), team(T), team(S),
```

```
       in_team(P,T), not_eq(T,S), has_ball(P).
        % player P has ball
22 | goal(P)
       :- player(P), team(T), team(S),
       in_team(P,T), not_eq(T,S), has_ball(P).
        % player P has ball
23 | pass(P,Q)
       :- player(P), player(Q), team(T), team(S),
       has_ball(P), teammate(P,Q), in_team(P,T),
       not_eq(T,S).  % player P has ball
24 | steal(Q,P)
       :- player(P), player(Q), team(T), team(S),
       has_ball(P), opponent(P,Q), in_team(P,T),
       in_team(Q,S).  % player P has ball
```

Note that any P in team T can kick the ball off when team T possesses the ball; that is, many events <u>kickoff</u>(P) are all possible. Once a specific player P takes possession of the ball (by <u>kickoff</u>, receiving a <u>pass</u>, or <u>steal</u>ing the ball), it is only P who can <u>kick</u>, <u>goal</u>, <u>pass</u> or be the target of who <u>steal</u>s the ball. How the possession gets transferred between players and teams is specified as follows:

```
25 | has_ball(a)
       ← init.   % team a has ball at beginning
26 | !has_ball(T) ← kickoff(P), in_team(P,T).
27 | !has_ball(P) ← goal(P).
28 | has_ball(S)
       ← goal(P), in_team(P,T), not_eq(T,S).
29 | !has_ball(P) ← pass(P,Q).
30 | has_ball(Q) ← pass(P,Q).
31 | !has_ball(P) ← steal(Q,P).
32 | has_ball(Q) ← steal(Q,P).
```

Each event updates not only who does it but also who observes it according to the ← rules below:

```
33 | player(P)
       ← init, is_player(P).
34 | team(T)
       ← init, is_team(T).
35 | player(P)
       ← kickoff(P), player(P), team(T),
       team(S), in_team(P,T), not_eq(T,S)
       :: individual.
36 | team(T)
       ← kickoff(P), player(P), team(T),
       team(S), in_team(P,T), not_eq(T,S)
       :: team_local.
37 | team(S)
       ← kickoff(P), player(P), team(T),
       team(S), in_team(P,T), not_eq(T,S)
       :: team_obj.
38 | player(P)
       ← kick(P), player(P), team(T), team(S),
       in_team(P,T), not_eq(T,S)
       :: individual.
39 | team(T)
       ← kick(P), player(P), team(T), team(S),
       in_team(P,T), not_eq(T,S)
       :: team_local.
40 | team(S)
       ← kick(P), player(P), team(T), team(S),
       in_team(P,T), not_eq(T,S)
       :: team_obj.
```

```
41 | player(P)
       ← goal(P), player(P), team(T), team(S),
       in_team(P,T), not_eq(T,S)
       :: individual.
42 | team(T)
       ← goal(P), player(P), team(T), team(S),
       in_team(P,T), not_eq(T,S)
       :: team_local.
43 | team(S)
       ← goal(P), player(P), team(T), team(S),
       in_team(P,T), not_eq(T,S)
       :: team_obj.
44 | player(P)
       ← pass(P,Q), player(P), player(Q),
       team(T), team(S), in_team(P,T), not_eq(T,S)
       :: individual_local.
45 | player(Q)
       ← pass(P,Q), player(P), player(Q),
       team(T), team(S), in_team(P,T), not_eq(T,S)
       :: individual_obj.
46 | team(T)
       ← pass(P,Q), player(P), player(Q),
       team(T), team(S), in_team(P,T), not_eq(T,S)
       :: team_local_inter.
47 | team(S)
       ← pass(P,Q), player(P), player(Q),
       team(T), team(S), in_team(P,T), not_eq(T,S)
       :: team_obj_inter.
48 | player(Q)
       ← steal(Q,P), player(P), player(Q),
       team(T), team(S), in_team(P,T), not_eq(T,S)
       :: individual_local.
49 | player(P)
       ← steal(Q,P), player(P), player(Q),
       team(T), team(S), in_team(P,T), not_eq(T,S)
       :: individual_obj.
50 | team(S)
       ← steal(Q,P), player(P), player(Q),
       team(T), team(S), in_team(P,T), not_eq(T,S)
       :: team_local_inter.
51 | team(T)
       ← steal(Q,P), player(P), player(Q),
       team(T), team(S), in_team(P,T), not_eq(T,S)
       :: team_obj_inter.
```

We also implement the architectures of Know-Evolve and DyRep as Datalog programs. The relational facts and rules are the same for all three models. The Know-Evolve program continues as follows:

```
52 | :- embed(player, 8).
53 | :- embed(kickoff_emb, 8).
54 | :- embed(kick_emb, 8).
55 | :- embed(goal_emg, 8).
56 | :- embed(pass_emb, 8).
57 | :- embed(steal_emb, 8).
58 | :- event(kickoff, 0).
59 | :- event(kick, 0).
60 | :- event(goal, 0).
61 | :- event(pass, 0).
62 | :- event(steal, 0).
```

Here are the :- rules:

```
63 | kickoff(P) :- player(P).
64 | kick(P) :- player(P).
```

(a) IPTV Dataset

(b) RoboCup Dataset
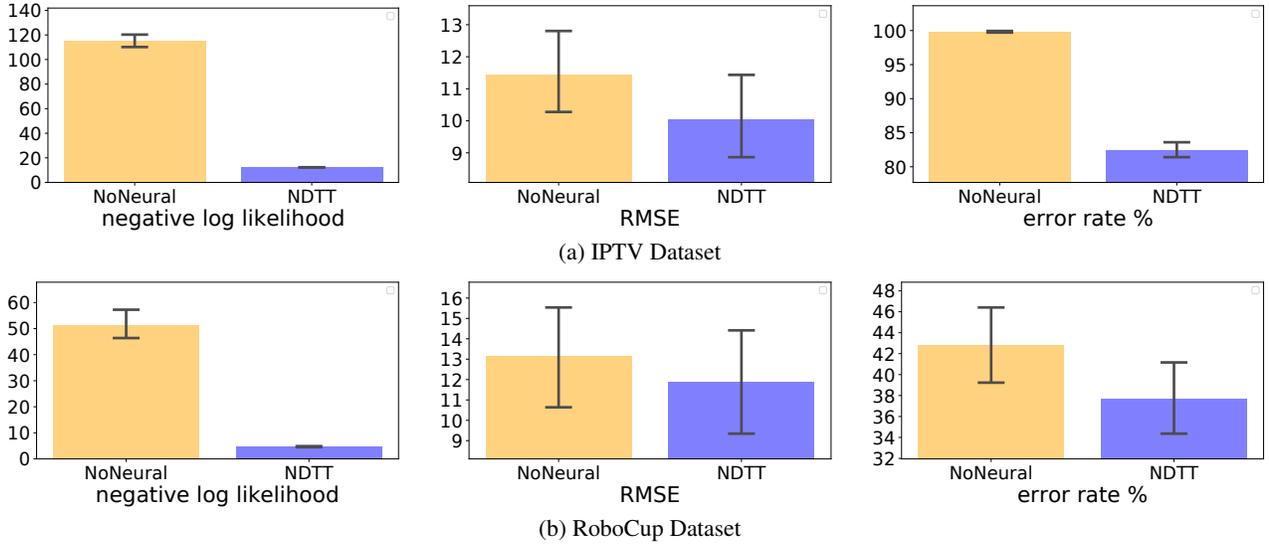
Figure 4. Ablation study of taking away neural networks from our Datalog programs in the real-world domains. The format of the graphs is the same as in Figure 2. The results imply that neural networks have been learning useful representations that are not explicitly specified in the Datalog programs.

```
65 | goal(P) :- player(P).
66 | pass(P,Q)
        :- player(P), player(Q), teammate(P,Q).
67 | steal(Q,P)
        :- player(P), player(Q), opponent(P,Q).
68 | kickoff_emb(P) :- is_player(P).
69 | kick_emb(P) :- is_player(P).
70 | goal_emb(P) :- is_player(P).
71 | pass_emb(P,Q) :- teammate(P,Q).
72 | steal_emb(Q,P) :- opponent(P,Q).
73 | proj_local(P,Q)
        :- player(P), is_player(Q) :: proj.
74 | proj_local(P,Q)
        :- player(Q), is_player(P) :: proj.
75 | proj_obj(P,Q)
        :- player(P), is_player(Q) :: proj.
76 | proj_obj(P,Q)
        :- player(Q), is_player(P) :: proj.
```

Note that each static event embedding such as `pass_emb(P,Q)` has its embedding independent of P or Q because the parameter names of each rule don't consider P or Q and functors `is_player`, `teammate` and `opponent` have dimension 0.

Here are the ← rules:

```
77 | player(P)
        ← init, is_player(P).
78 | player(P)
        ← kickoff(P), kickoff_emb(P), player(P)
        :: individual.
79 | player(P)
        ← kick(P), kick_emb(P), player(P)
        :: individual.
80 | player(P)
        ← goal(P), goal_emb(P), player(P)
        :: individual.
81 | player(P)
```

```
        ← pass(P,Q), pass_emb(P,Q),
        proj_local(P,Q)
        :: individual_local.
82 | player(Q)
        ← pass(P,Q), pass_emb(P,Q), proj_obj(P,Q)
        :: individual_obj.
83 | player(Q)
        ← steal(Q,P), steal_emb(Q,P),
        proj_obj(P,Q)
        :: individual_local.
84 | player(P)
        ← steal(Q,P), steal_emb(Q,P),
        proj_local(P,Q)
        :: individual_obj.
```

The DyRep program also makes the intensity of each event depend on its participants, so the :- rules about intensities are the same as in Know-Evolve. As in Appendix F.3, the standard DyRep model doesn't have embeddings for event types but our reimplementation followed their appendices to assign a static embedding to each event type. The way these static embeddings are computed is defined to be the same as in Know-Evolve. However, for the ← rules, DyRep considers the neighbors of participants while updating them. Although it is not obvious how neighbors are defined in this domain, we consider the ball as a player P's neighbor if P possesses the ball. This design is meaningful because it offers a way for DyRep to consider the dynamics of who possesses the ball at different times. The dimension declaration, :- rules and ← rules of DyRep are as follows:

```
85 | :- embed(player, 8).
86 | :- embed(ball, 8).
87 | :- embed(kickoff_emb, 8).
88 | :- embed(kick_emb, 8).
89 | :- embed(goal_emg, 8).
90 | :- embed(pass_emb, 8).
```

```
91  :- embed(steal_emb, 8).
92  :- event(kickoff, 0).
93  :- event(kick, 0).
94  :- event(goal, 0).
95  :- event(pass, 0).
96  :- event(steal, 0).
97  kickoff(P) :- player(P).
98  kick(P) :- player(P).
99  goal(P) :- player(P).
100 pass(P,Q) :- player(P), player(Q),
         teammate(P,Q).
101 steal(Q,P) :- player(P), player(Q),
         opponent(P,Q).
102 kickoff_emb(P) :- is_player(P).
103 kick_emb(P) :- is_player(P).
104 goal_emb(P) :- is_player(P).
105 pass_emb(P,Q) :- teammate(P,Q).
106 steal_emb(Q,P) :- opponent(P,Q).
107 player(P) <- init, is_player(P).
108 ball <- init.
109 player(P)
         <- kickoff(P), kickoff_emb(P), player(P)
         :: individual.
110 player(P)
         <- kickoff(P), ball :: ball.
111 player(P)
         <- kick(P), kick_emb(P), player(P)
         :: individual.
112 player(P)
         <- kick(P), ball :: ball.
113 player(P)
         <- goal(P), goal_emb(P), player(P)
         :: individual.
114 player(P)
         <- goal(P), ball :: ball.
115 player(P)
         <- pass(P,Q), pass_emb(P,Q), player(P)
         :: individual.
116 player(P)
         <- pass(P,Q), ball :: ball.
117 player(Q)
         <- pass(P,Q), pass_emb(P,Q), player(Q)
         :: individual.
118 player(P)
         <- steal(Q,P), steal_emb(Q,P), player(P)
         :: individual.
119 player(P)
         <- steal(Q,P), ball :: ball.
120 player(Q)
         <- steal(Q,P), steal_emb(Q,P), player(Q)
         :: individual.
```

F.5. Training Details

We had to choose the dimension D that is specified in the `embed` and `event` declarations of the programs in Appendices F.3 and F.4. For simplicity, all declarations within a program used the same dimension D, so we only had a single hyperparameter to tune. We tuned this separately for each domain, for each competing architecture, and each training size (e.g., each point in Figure 1 and each bar in Figures 2 and 3), 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 was usually 32 or 64.

To train the parameters for a given D, we used the Adam algorithm (Kingma & 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.

F.6. Ablation Study II Details

In the final experiment of §6.2, each event type still has an extra dimension for its intensity (see §3.2). The set of possible events at any time is unchanged. 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 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).

Our experimental results in Figure 4 show that the neural networks have really been learning representations that are actually helpful for prediction.