Ripple: A Language for Neurosymbolic Programming

ZIYANG LI, University of Pennsylvania
JIANI HUANG, University of Pennsylvania
MAYUR NAIK, University of Pennsylvania

We present Ripple, a language which combines the benefits of deep learning and logical reasoning. Ripple enables users to write a wide range of neurosymbolic applications and train them in a data and compute efficient manner. It achieves these goals through three key features: 1) a flexible symbolic representation that is based on the relational data model; 2) a declarative logic programming language that is based on Datalog and supports recursion, aggregation, and negation; and 3) a framework for automatic and efficient differentiable reasoning that is based on the theory of provenance semirings. We evaluate Ripple on a suite of eight neurosymbolic applications from the literature. Our evaluation demonstrates that Ripple is capable of expressing algorithmic reasoning in diverse and challenging AI tasks, provides a succinct interface for machine learning programmers to integrate logical domain knowledge, and yields solutions that are comparable or superior to state-of-the-art models in terms of accuracy. Furthermore, Ripple's solutions outperform these models in aspects such as runtime and data efficiency, interpretability, and generalizability.

1 INTRODUCTION

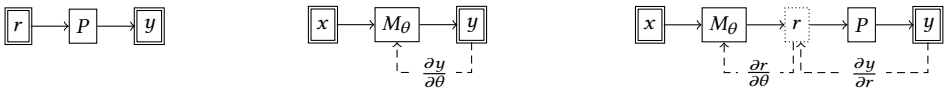
Classical algorithms and deep learning embody two prevalent paradigms of modern programming. Classical algorithms are well suited for exactly defined tasks, such as sorting a list of numbers or finding a shortest path in a graph. Deep learning, on the other hand, is well suited for tasks that are not tractable or feasible to perform using classical algorithms, such as detecting objects in an image or parsing natural language text. These tasks are typically specified using a set of input-output training data, and solving them involves learning the parameters of a deep neural network to fit the data using gradient-descent based methods.

The two paradigms are complementary in nature. For instance, a classical algorithm such as the logic program P depicted in Figure 1a is interpretable but operates on limited (e.g., structured) input r. In contrast, a deep neural network such as M_θ depicted in Figure 1b can operate on rich (e.g., unstructured) input x but is not interpretable. Modern applications demand the capabilities of both paradigms. Examples include question answering [43], code completion [10], and mathematical problem solving [34], among many others. For instance, code completion requires deep learning to comprehend programmer intent from the code context, and classical algorithms to ensure that the generated code is correct. A natural and fundamental question then is how to program such applications by integrating the two paradigms.

Neurosymbolic programming is an emerging paradigm that aims to fulfill this goal [9]. It seeks to integrate symbolic knowledge and reasoning with neural architectures for better efficiency, interpretability, and generalizability than the neural or symbolic counterparts alone. Consider the task of handwritten formula evaluation [35], which takes as input a formula as an image, and outputs a number corresponding to the result of evaluating it. An input-output example for this task is $\langle x = \text{<image>}, y = 1.6 \rangle$ A neurosymbolic program for this task, such as the one depicted in Figure 1c, might first apply a convolutional neural network M_θ to the input image to obtain a structured intermediate form r as a sequence of symbols ['1', '+', '3', '/', '5'], followed by a classical algorithm P to parse the sequence, evaluate the parsed formula, and output the final result 1.6.

Despite significant strides in individual neurosymbolic applications [11, 35, 39, 40, 53, 55], there is a lack of a language with compiler support to make the benefits of the neurosymbolic paradigm

PLDI'23, January 01–03, 2018, New York, NY, USA
2018.



(a) Logic program. (b) Neural model. (c) A basic neurosymbolic program.

Fig. 1. Comparison of different paradigms. Logic program P accepts only structured input r whereas neural model M_θ with parameter θ can operate on unstructured input x. Supervision is provided on data indicated in double boxes. Under *algorithmic supervision*, a neurosymbolic program must learn θ without supervision on r.

more widely accessible. We set out to develop such a language and identified five key criteria that it should satisfy in order to be practical. These criteria, annotated by the components of the neurosymbolic program in Figure 1c, are as follows:

(1) A symbolic data representation for r that supports diverse kinds of data, such as image, video, natural language text, tabular data, and their combinations.
(2) A symbolic reasoning language for P that allows to express common reasoning patterns such as recursion, negation, and aggregation.
(3) An automatic and efficient differentiable reasoning engine for learning ($\frac{\partial y}{\partial r}$) under *algorithmic supervision*, i.e., supervision on observable input-output data (x, y) but not r.
(4) The ability to tailor learning ($\frac{\partial y}{\partial r}$) to individual applications' characteristics, since non-continuous loss landscapes of logic programs hinder learning using a one-size-fits-all method.
(5) A mechanism to leverage and integrate with existing training pipelines ($\frac{\partial r}{\partial \theta}$), implementations of neural architectures and models M_θ, and hardware (e.g., GPU) optimizations.

In this paper, we present Ripple[1], a language which satisfies the above criteria. The key insight underlying Ripple is its choice of three inter-dependent design decisions: a relational model for symbolic data representation, a declarative language for symbolic reasoning, and a provenance framework for differentiable reasoning. We elaborate upon each of these decisions.

Relations can represent arbitrary graphs and are therefore well-suited for representing symbolic data in Ripple applications. For instance, they can represent *scene graphs*, a canonical symbolic representation of images [31], or abstract syntax trees, a symbolic representation of natural language text. Further, they can be combined in a *relational database* to represent multi-modal data. Relations are also a natural fit for probabilistic reasoning [18], which is necessary since symbols in r produced by neural model M_θ can have associated probabilities.

Next, the symbolic reasoning program P in Ripple is specified in a declarative logic programming language. The language extends Datalog [2] and is expressive enough for programmers to specify complex domain knowledge patterns that the neural model M_θ would struggle with. Datalog implementations can take advantage of optimizations from the literature on relational database systems. This in turn enables efficient inference and learning since the logical domain knowledge specifications of the task at hand help reduce the burden of M_θ, whose responsibilities are now less complex and more modular. Finally, Datalog is rule-based, which makes programs easier to write, debug, and verify. It also facilitates inferring them by leveraging techniques from program synthesis [26] and inductive logic programming [14].

While symbolic reasoning offers many aforementioned benefits, it poses a fundamental challenge for learning parameter θ. Deep learning relies on gradient-descent based methods, enabled by the differentiable nature of the low-level activation functions that comprise M_θ, to obtain $\frac{\partial r}{\partial \theta}$. The key challenge then concerns how to support automatic and efficient differentiation of the high-level logic program P to obtain $\frac{\partial y}{\partial r}$, which can be used in conjunction with $\frac{\partial r}{\partial \theta}$ to compute $\frac{\partial y}{\partial \theta}$. Ripple addresses this problem by leveraging the framework of *provenance semirings* [23]. The

[1]The system name has been anonymized for review

framework proposes a common algebraic structure for applications that define annotations (i.e., tags) for tuples and propagate the annotations from inputs to outputs of relational algebra (RA) queries. One of our primary contributions is a novel adaptation of the framework for differentiable reasoning for an extended fragment of RA that includes recursion, negation, and aggregation. Ripple implements an extensible library of provenance structures including the extended max-min semiring and the top-k proofs semiring [28]. We further demonstrate that different provenance structures enable different heuristics for the gradient calculations, providing an effective mechanism to tailor the learning process to individual applications' characteristics.

We have implemented a comprehensive and open-source toolchain for Ripple in 45K LoC of Rust. It includes a compiler, an interpreter, and PyTorch bindings to integrate Ripple programs with existing machine learning pipelines. We evaluate Ripple using a suite of eight neurosymbolic applications that span the domains of image and video processing, natural language processing, planning, and knowledge graph querying, in a variety of learning settings such as supervised learning, reinforcement learning, rule learning, and contrastive learning. Our evaluation demonstrates that Ripple is expressive and yields solutions of comparable, and often times superior, accuracy than state-of-the-art models. We show additional benefits of Ripple's solutions in terms of runtime and data efficiency, interpretability, and generalizability.

The rest of the paper is organized as follows. Section 2 presents an illustrative overview of Ripple. Section 3 describes Ripple's language for symbolic reasoning. Section 4 presents the differentiable reasoning framework. Section 5 describes our implementation of Ripple. Section 6 empirically evaluates Ripple on a benchmark suite. Section 7 surveys related work and Section 8 concludes.

2 ILLUSTRATIVE OVERVIEW

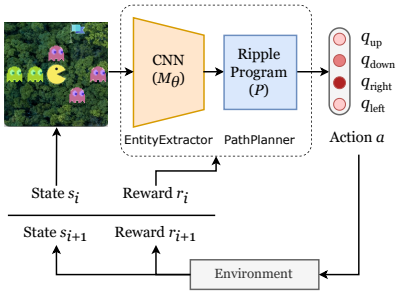
We illustrate Ripple using an reinforcement learning (RL) based planning application which we call PacMan-Maze. The application, depicted in Fig. 2a, concerns an intelligent agent realizing a sequence of actions in a simplified version of the PacMan maze game. The maze is an implicit 5×5 grid of cells. Each cell is either empty or has an entity, which can be either the *actor* (PacMan), the *goal* (flag), or an *enemy* (ghost). At each step, the agent moves the actor in one of four directions: up, down, right, or left. The game ends when the actor reaches the goal or hits an enemy. The maze is provided to the agent as a raw image that is updated at each step, requiring the agent to process sensory inputs, extract relevant features, and logically plan the path to take. Additionally, each session of the game has randomized initial positions of the actor, the goal, and the enemies.

More concretely, the game is modeled as a sequence of interactions between the agent and an environment, as depicted in Fig. 2b. The game state $s_i \in S$ at step i is a 200×200 colored image ($S = \mathbb{R}^{200\times200\times3}$). The agent proposes an action $a_i \in A = \{\texttt{up}, \texttt{down}, \texttt{right}, \texttt{left}\}$ to the environment, which generates a new image s_{i+1} as the next state. The environment also returns a reward r_i to the agent: 1 upon reaching the goal, and 0 otherwise. This procedure repeats until the game ends or reaches a predefined limit on the number of steps.

A popular RL method to realize our application is Q-Learning. Its goal is to learn a function $Q : S \times A \rightarrow \mathbb{R}$ that returns the expected reward of taking action a_i in state s_i.[2] Since the game states are images, we employ a variant called Deep Q-Learning [41], which approximates the Q function using a convolutional neural network (CNN) model with learned parameter θ. An end-to-end deep learning based approach for our application involves training the model to predict the Q-value of each action for a given game state. This approach, however, takes 50K training episodes to achieve a 84.9% test success rate, where a single episode is one gameplay session from start to end.

[2]We elide the Q-Learning algorithm as it is not needed to illustrate the neurosymbolic programming aspects of our example.

(a) Three states of one gameplay session.



(b) Architecture of application with Ripple.

```
1  class PacManAgent(torch.nn.Module):
2   def __init__(self, grid_dim, cell_size):
3    # initializations...
4    self.extract_entities =
5     EntityExtractor(grid_dim, cell_size)
6    self.path_planner = RippleModule(
7     file="path_planner.rpl",
8     provenance="diff-top-k-proofs", k=1,
9     input_mappings={"actor": cells,
10     "goal": cells, "enemy": cells},
11    output_mappings={"next_action": actions})
12
13  def forward(self, game_state_image):
14   actor, goal, enemy =
15    self.extract_entities(game_state_image)
16   next_action = self.path_planner(
17    actor=actor, goal=goal, enemy=enemy)
18   return next_action
```

(c) Snippet of implementation in Python.

Fig. 2. Illustration of a planning application PacMan-Maze in Ripple.

In contrast, a neurosymbolic solution using Ripple only needs 50 training episodes to attain a 99.4% test success rate. Ripple enables to realize these benefits of the neurosymbolic paradigm by decomposing the agent's task into separate neural and symbolic components, as shown in Fig. 2b. These components perform sub-tasks that are ideally suited for their respective paradigms: the neural component perceives pixels of individual cells of the image at each step to identify the entities in them, while the symbolic component reasons about enemy-free paths from the actor to the goal to determine the optimal next action. Fig. 2c shows an outline of this architecture's implementation using the popular PyTorch framework.

The neural component is still a convolutional neural network model, but it now takes the pixels of a single cell in the input image at a time, and classifies the entity in it. The implementation of the neural component (`EntityExtractor`) is standard and elided for brevity. It is invoked on lines 14-15 with input `game_state_image`, a tensor in $\mathbb{R}^{200\times200\times3}$, and returns three $\mathbb{R}^{5\times5}$ tensors of entities. For example, `actor` is an $\mathbb{R}^{5\times5}$ tensor and $\texttt{actor}_{ij}$ is the probability of the actor being in cell (i, j). A representation of the entities is then passed to the symbolic component on lines 16-17, which derives the Q-value of each action. The symbolic component, which is configured on lines 6-11, comprises the Ripple program shown in Fig. 3. We next illustrate the three key design decisions of Ripple (outlined in Section 1) with respect to this program.

Relational Model. In Ripple, the primary data structure for representing symbols is a *relation*. In our example, the game state can be symbolically described by the kinds of entities that occur in the discrete cells of a 5×5 grid. We can therefore represent the input to the symbolic component using binary relations for the three kinds of entities: `actor`, `goal`, and `enemy`. For instance, the fact `actor(2,3)` indicates that the actor is in cell (2,3). Likewise, since there are four possible actions, the output of the symbolic component is represented by a unary relation `next_action`.

Symbols extracted from unstructured inputs by neural networks have associated probabilities, such as the $\mathbb{R}^{5\times5}$ tensor `actor` produced by the neural component in our example (line 14 of Fig. 2c). Ripple therefore allows to associate tuples with probabilities, e.g. 0.96 :: `actor(2,3)`, to indicate that the actor is in cell (2,3) with probability 0.96. More generally, Ripple enables the conversion of

```
1  // File path_planner.rpl
2  type actor(x: i32, y: i32), goal(x: i32, y: i32), enemy(x: i32, y: i32)
3
4  const UP = 0, DOWN = 1, RIGHT = 2, LEFT = 3
5  rel safe_cell(x, y) = range(0, 5, x), range(0, 5, y), not enemy(x, y)
6  rel edge(x, y, x, yp, UP) = safe_cell(x, y), safe_cell(x, yp), yp == y + 1
7  // Rules for DOWN, RIGHT, and LEFT edges are omitted...
8
9  rel next_pos(p, q, a) = actor(x, y), edge(x, y, p, q, a)
10 rel path(x, y, x, y) = next_pos(x, y, _)
11 rel path(x1, y1, x3, y3) = path(x1, y1, x2, y2), edge(x2, y2, x3, y3, _)
12 rel next_action(a) = next_pos(p, q, a), path(p, q, r, s), goal(r, s)
```

Fig. 3. The logic program of the PacMan-Maze application in Ripple.

tensors in the neural component to and from relations in the symbolic component via input-output mappings (lines 9-11 in Fig. 2c), allowing the two components to exchange information seamlessly.

Declarative Language. Another key consideration in a neurosymbolic language concerns what constructs to provide for symbolic reasoning. Ripple uses a declarative language based on Datalog, which we present in Section 3 and illustrate here using the program in Fig. 3. The program realizes the symbolic component of our example using a set of logic rules. Following Datalog syntax, they are "if-then" rules, read right to left, with commas denoting conjunction.

Recall that we wish to determine an action a (up, down, right, or left) to a cell (p, q) that is adjacent to the actor's cell (x, y) such that there is an enemy-free path from (p, q) to the goal's cell (r, s). The nine depicted rules succinctly compute this sophisticated reasoning pattern by building successively complex relations, with the final rule (on line 14) computing all such actions.[3]

The arguably most complex concept is the `path` relation which is recursively defined (on lines 10-11). Recursion allows to define the pattern succinctly, enables the trained application to generalize to grids arbitrarily larger than 5×5 unlike the purely neural version, and makes the pattern more amenable to synthesis from input-output examples. Besides recursion, Ripple also supports negation and aggregation; together, these features render the language adequate for specifying common high-level reasoning patterns in practice.

Differentiable Reasoning. With the neural and symbolic components defined, the last major consideration concerns how to train the neural component using only end-to-end supervision. In our example, supervision is provided in the form of a reward of 1 or 0 per gameplay session, depending upon whether or not the sequence of actions by the agent successfully led the actor to the goal without hitting any enemy. This form of supervision, called algorithmic or weak supervision, alleviates the need to label intermediate relations at the interface of the neural and symbolic components, such as the `actor`, `goal`, and `enemy` relations. However, this also makes it challenging to learn the gradients for the tensors of these relations, which in turn are needed to train the neural component using gradient-descent techniques.

The key insight in Ripple is to exploit the structure of the logic program to guide the gradient calculations. The best heuristic for such calculations depends on several application characteristics such as the amount of available data, reasoning patterns, and the learning setup. Ripple provides a convenient interface for the user to select from a library of built-in heuristics. Furthermore, since all of these heuristics follow the structure of the logic program, Ripple implements them uniformly as instances of a general and extensible *provenance framework*, described in Section 4. For our

[3]We elide showing an auxiliary relation of all grid cells tagged with probability 0.99 which serves as the penalty for taking a step. Thus, longer paths are penalized more, driving the model to choose an action that moves the actor closer to the goal.

example, line 8 in Fig. 2c specifies `diff-top-k-proofs` with `k=1` as the heuristic to use, which is the default in Ripple that works best for many applications, as we demonstrate in Section 6.

3 LANGUAGE

We provide an overview of Ripple's language for symbolic reasoning which we previously illustrated in the program shown in Fig. 3. Appendix A provides the formal syntax of the language. Here, we illustrate each of the key constructs using examples of inferring kinship relations.

3.1 Data Types

The fundamental data type in Ripple is set-valued relations comprising tuples of statically-typed primitive values. The primitive data types include signed and unsigned integers of various sizes (e.g. `i32`, `usize`), single- and double-precision floating point numbers (`f32`, `f64`), boolean (`bool`), character (`char`), and string (`String`). The following example declares two binary relations, `mother` and `father`:

```
type mother(c: String, m: String), father(c: String, f: String)
```

Values of relations can be specified via individual tuples or a set of tuples of constant literals:

```
rel mother("Bob", "Christine")                    // Christine is Bob's mother
rel father = {("Alice", "Bob"), ("John", "Bob")} // Bob is father of two kids
```

As a shorthand, primitive values can be named and used as constant variables:

```
const FATHER = 0, MOTHER = 1, GRANDMOTHER = 2, ... // other relationships
rel transitive(FATHER, MOTHER, GRANDMOTHER) // father's mother is grandmother
```

Type declarations are optional since Ripple supports type inference. The type of the `transitive` relation is inferred as `(usize, usize, usize)` since the default type of unsigned integers is `usize`.

3.2 (Horn) Rules

Since Ripple's language is based on Datalog, it supports "if-then" rule-like Horn clauses. Each rule is composed of a head atom and a body, connected by the symbol `:-` or `=`. The following code shows two rules defining the `grandmother` relation. Conjunction is specified using "`,`"-separated atoms within the rule body whereas disjunction is specified by multiple rules with the same head predicate. Each variable appearing in the head must also appear in some positive atom in the body (we introduce negative atoms below).

```
rel grandmother(a, c) :- father(a, b), mother(b, c) // father's mother
rel grandmother(a, c) :- mother(a, b), mother(b, c) // mother's mother
```

Conjunctions and disjunctions can also be expressed using logical connectives like `and`, `or`, and `implies`. For instance, the following rule is equivalent to the above two rules combined.

```
rel grandmother(a, c) = (mother(a, b) or father(a, b)) and mother(b, c)
```

Ripple supports value creation by means of foreign functions (FFs). FFs are polymorphic and include arithmetic operators such as `+` and `-`, comparison operators such as `!=` and `>=`, type conversions such as `[i32] as String`, and built-in functions like `$hash` and `$string_concat`. They only operate on primitive values but not relational tuples or atoms. The following example shows how strings are concatenated together using FF, producing the result `full_name("Alice Lee")`.

```
rel first_name("Alice"), last_name("Lee")
rel full_name($string_concat(x, "␣", y)) = first_name(x), last_name(y)
```

Note that FFs can fail due to runtime errors such as division-by-zero and integer overflow, in which case the computation for that single fact is omitted. The purpose of this semantics is to support probabilistic extensions (Section 3.3) rather than silent suppression of runtime errors.

Recursion. A relation r is dependent on s if an atom s appears in the body of a rule with head atom r. A *recursive* relation is one that depends on itself, directly or transitively. The following rule derives additional `kinship` facts by combine existing `kinship` facts using the `transitive` relation.

```
rel kinship(r3,a,c) = kinship(r1,a,b), kinship(r2,b,c), transitive(r1,r2,r3)
```

Negation. Ripple supports stratified negation using the `not` operator on atoms in the rule body. The following example shows a rule defining the `has_no_children` relation as any person `p` who is neither a father nor a mother. Note that we need to bound `p` by a positive atom `person` in order for the rule to be well-formed.

```
rel person = {"Alice", "Bob", "Christine"} // can omit () since arity is 1
rel has_no_children(p) = person(p) and not father(_, p) and not mother(_, p)
```

A relation r is *negatively* dependent on s if a negated atom s appears in the body of a rule with head atom r. In the above example, `has_no_children` negatively depends on `father`. A relation cannot be negatively dependent on itself, directly or transitively, as Ripple supports only stratified negation.

Aggregation. Ripple also supports stratified aggregation. The set of built-in aggregators include common ones such as `count`, `sum`, `max`, and first-order quantifiers `forall` and `exists`. Besides the operator, the aggregation specifies the binding variables, the aggregation body to bound those variables, and the result variable(s) to assign the result. The aggregation in the example below reads, "variable `n` is assigned the count of `p`, such that `p` is a person":

```
rel num_people(n) = n := count(p: person(p))
```

In the rule, `p` is the binding variable and `n` is the result variable. Depending on the aggregator, there could be multiple binding variables or multiple result variables. Further, Ripple supports SQL-style group-by using a `where` clause in the aggregation. In the following example, we compute the number of children of each person `p`, which serves as the group-by variable.

```
rel parent(a, b) = father(a, b) or mother(a, b)
rel num_child(p, n) = n := count(c: parent(c, p) where p: person(p))
```

Finally, quantifier aggregators such as `forall` and `exists` return one boolean variable. For instance, in the aggregation below, variable `sat` is assigned the truthfulness (`true` or `false`) of the following statement: "for all `a` and `b`, if `b` is `a`'s father, then `a` is `b`'s son or daughter".

```
rel integrity_constraint(sat) =
    sat := forall(a, b: father(a, b) implies (son(b, a) or daughter(b, a)))
```

There are a couple of syntactic checks on aggregations. First, similar to negation, aggregation also needs to be stratified—a relation cannot be dependent on itself through an aggregation. Second, the binding variables must be bounded by a positive atom in the body of the aggregation.

3.3 Probabilistic Extensions

Although Ripple is designed primarily for neurosymbolic programming, its syntax also supports probabilistic programming. This is especially useful when debugging Ripple code before integrating it with a neural network. Consider a machine learning programmer who wishes to extract structured relations from a natural language sentence "Bob takes his daughter Alice to the beach". The programmer could imitate a neural network producing a probability distribution of kinship relations between Alice (`A`) and Bob (`B`) as follows:

```
rel kinship = {0.95::(FATHER, A, B); 0.01::(MOTHER, A, B); ... }
```

Here, we list out all possible kinship relations between Alice and Bob. For each of them, we use the syntax `[PROB]::[TUPLE]` to tag the kinship tuples with probabilities. The semicolon ";" separating them specifies that they are mutually exclusive—Bob cannot be both the mother and father of Alice.

(Tag)	t	$\in$	T
(False)	$\mathbf{0}$	$\in$	T
(True)	$\mathbf{1}$	$\in$	T
(Disjunction)	$\oplus$	:	$T \times T \to T$
(Conjunction)	$\otimes$	:	$T \times T \to T$
(Negation)	$\ominus$	:	$T \to T$
(Saturation)	$\Subset$	:	$T \times T \to \text{Bool}$

Fig. 4. Algebraic interface for provenance.

(Predicate)	p		
(Aggregator)	g	::=	`count` | `sum` | `max` | `exists` | ...
(Expression)	e	::=	$p \mid \gamma_g(e) \mid \pi_\alpha(e) \mid \sigma_\beta(e)$
		|	$e_1 \cup e_2 \mid e_1 \times e_2 \mid e_1 - e_2$
(Rule)	r	::=	$p \leftarrow e$
(Stratum)	s	::=	$\{r_1, \ldots, r_n\}$
(Program)	$\overline{s}$	::=	$s_1; \ldots; s_n$

Fig. 5. Abstract syntax of core fragment of RplRam.

Ripple also supports operators to sample from probability distributions. They share the same surface syntax as aggregations, allowing sampling with group-by. The following rule deterministically picks the most likely kinship relation between a given pair of people `a` and `b`, which are implicit group-by variables in this aggregation. As the end, only one fact, `0.95::top_1_kinship(FATHER, A, B)`, will be derived according to the above probabilities.

```
rel top_1_kinship(r, a, b) = r := top<1>(rp: kinship(rp, a, b))
```

Other types of sampling are also supported, including categorical sampling (`categorical<K>`) and uniform sampling (`uniform<K>`), where a static constant `K` denotes the number of trials.

Finally, rules can also be tagged by probabilities which can reflect their confidence. The following rule states that a grandmother's daughter is one's mother with 90% confidence.

```
rel 0.9::mother(a, c) = grandmother(a, b) and daughter(b, c)
```

Probabilistic rules are syntactic sugar. They are implemented by introducing in the rule's body an auxiliary 0-ary (i.e., boolean) fact that is regarded true with the tagged probability.

4 REASONING FRAMEWORK

The preceding section presented Ripple's surface language for use by programmers to express discrete reasoning. However, the language must also support differentiable reasoning to enable end-to-end training. In this section, we formally define the semantics of the language by means of a provenance framework. We show how Ripple uniformly supports different reasoning modes—discrete, probabilistic, and differentiable—simply by defining different provenances.

We start by presenting the basics of our provenance framework (Section 4.1). We then present a low-level representation RplRam, its operational semantics, and its interface to the rest of a Ripple application (Sections 4.2-4.4). We next present differentiation and three different provenances for differentiable reasoning (Section 4.5). Lastly, we discuss practical considerations (Section 4.6).

4.1 Provenance Framework

Ripple's provenance framework enables to tag and propagate additional information alongside relational tuples in the logic program's execution. The framework is based on the theory of *provenance semirings* [23]. Fig. 4 defines Ripple's algebraic interface for provenance. We call the additional information a *tag* t from a *tag space* T. There are two distinguished tags, $\mathbf{0}$ and $\mathbf{1}$, representing unconditionally *false* and *true* tags. Tags are propagated through operations of binary *disjunction* $\oplus$, binary *conjunction* $\otimes$, and unary *negation* $\ominus$ resembling logical *or*, *and*, and *not*. Lastly, a *saturation* check $\Subset$ serves as a customizable stopping mechanism for fixed-point iteration.

All of the above components combined form a 7-tuple $(T, \mathbf{0}, \mathbf{1}, \oplus, \otimes, \ominus, \Subset)$ which we call a *provenance* T. Ripple provides a built-in library of provenances and users can add custom provenances simply by implementing this interface.

Example 4.1. `max-min-prob` (`mmp`) $\triangleq$ $([0,1], 0, 1, \max, \min, \lambda x.(1-x), ==)$, is a built-in *probabilistic provenance*, where tags are numbers between 0 and 1 that are propagated with max and min.

$$
\begin{array}{rrcrcl}
\text{(Constant)} & \mathbb{C} & \ni & c & ::= & int \mid bool \mid str \mid \ldots \\
\text{(Tuple)} & \mathbb{U} & \ni & u & ::= & c \mid (u_1, \ldots, u_n) \\
\text{(Tagged-Tuple)} & \mathbb{U}_T & \ni & u_t & ::= & t :: u \\
\text{(Fact)} & \mathbb{F} & \ni & f & ::= & p(u) \\
\text{(Tagged-Fact)} & \mathbb{F}_T & \ni & f_t & ::= & t :: p(u)
\end{array}
\qquad
\begin{array}{rrcrcl}
\text{(Tuples)} & U & \in & \mathcal{U} & \triangleq & \mathcal{P}(\mathbb{U}) \\
\text{(Tagged-Tuples)} & U_T & \in & \mathcal{U}_T & \triangleq & \mathcal{P}(\mathbb{U}_T) \\
\text{(Facts)} & F & \in & \mathcal{F} & \triangleq & \mathcal{P}(\mathbb{F}) \\
\text{(Database)} & F_T & \in & \mathcal{F}_T & \triangleq & \mathcal{P}(\mathbb{F}_T)
\end{array}
$$

Fig. 6. Semantic domains for RplRam.

The tags do not represent true probabilities but are merely an approximation. We discuss richer provenances for more accurate probability calculations later in this section.

A provenance must satisfy a few properties. First, the 5-tuple $(T, \mathbf{0}, \mathbf{1}, \oplus, \otimes)$ should form a semiring. That is, $\mathbf{0}$ is the additive identity and annihilates under multiplication, $\mathbf{1}$ is the multiplicative identity, $\oplus$ and $\otimes$ are associative and commutative, and $\otimes$ distributes over $\oplus$. To guarantee the existence of fixed points (which are discussed in Section 4.3), it must also be *absorptive*, i.e., $t_1 \oplus (t_1 \otimes t_2) = t_1$ [15]. Moreover, we need $\ominus\, \mathbf{0} = \mathbf{1}$, $\ominus\, \mathbf{1} = \mathbf{0}$, $\mathbf{0} \not\ominus \mathbf{1}$, $\mathbf{0} \ominus \mathbf{0}$, and $\mathbf{1} \ominus \mathbf{1}$. A provenance which violates an individual property (e.g. absorptive) is still useful, for instance, to applications that do not use the affected features (e.g. recursion), or if the user simply wishes to bypass the restrictions.

4.2 RplRam Intermediate Language

Ripple programs are compiled to a low-level representation called RplRam. Fig. 5 shows the abstract syntax of a core fragment of RplRam. Expressions resemble queries in an extended relational algebra. They operate over relational predicates (p) using unary operations for aggregation (γ_g with aggregator g), projection (π_α with mapping α), and selection (σ_β with condition β), and binary operations union ($\cup$), product ($\times$), and difference ($-$).

A rule r in RplRam is denoted $p \leftarrow e$, where predicate p is the rule head and expression e is the rule body. An unordered set of rules combined form a stratum s, and a sequence of strata $s_1; \ldots; s_n$ constitutes a RplRam program. Rules in the same stratum have distinct head predicates. Denoting the set of head predicates in stratum s by P_s, we also require $P_{s_i} \cap P_{s_j} = \emptyset$ forall $i \neq j$ in a program. Stratified negation and aggregation from the surface language is enforced as syntax restrictions in RplRam: within a rule in stratum s_i, if a relational predicate p is used under aggregation (γ) or right-hand-side of difference ($-$), that predicate p cannot appear in P_{s_j} if $j \geq i$.

We next define the semantic domains in Fig. 6 which are used subsequently to define the semantics of RplRam. A tuple u is either a constant or a sequence of tuples. A fact $p(u) \in \mathbb{F}$ is a tuple u named under a relational predicate p. Tuples and facts can be tagged, forming *tagged tuples* $(t :: u)$ and *tagged facts* $(t :: p(u))$. Given a set of tagged tuples U_T, we say $U_T \vDash u$ iff there exists a t such that $t :: u \in U_T$. A set of tagged facts form a database F_T. We use bracket notation $F_T[p]$ to denote the set of tagged facts in F_T under predicate p.

4.3 Operational Semantics of RplRam

We now present the operational semantics for our core fragment of RplRam in Fig. 7. A RplRam program $\bar{s}$ takes as input an *extensional database* (EDB) F_T, and returns an *intentional database* (IDB) $F'_T = [\![\bar{s}]\!](F_T)$. The semantics is conditioned on the underlying provenance T. We call this *tagged semantics*, as opposed to the *untagged semantics* found in traditional Datalog.

Basic Relational Algebra. Evaluating an expression in RplRam yields a set of tagged tuples according to the rules defined at the top of Fig. 7. A predicate p evaluates to all facts under that predicate in the database. Selection filters tuples that satisfy condition β, and projection transforms tuples according to mapping α. The mapping function α is partial: it may fail since it can apply foreign functions to values. A tuple in a union $e_1 \cup e_2$ can come from either e_1 or e_2. In (cartesian) product, each pair of incoming tuples combine and we use $\otimes$ to compute the tag conjunction.

Expression semantics $\alpha : \mathbb{U} \to \mathbb{U}, \quad \beta : \mathbb{U} \to \text{Bool}, \quad g : \mathcal{U} \to \mathcal{U}, \quad [\![e]\!] : \mathcal{F}_T \to \mathcal{U}_T$

$$\frac{t :: p(u) \in F_T}{t :: u \in [\![p]\!](F_T)}\ (\textsc{Predicate}) \qquad \frac{t :: u \in [\![e]\!](F_T) \qquad \beta(u) = \text{true}}{t :: u \in [\![\sigma_\beta(e)]\!](F_T)}\ (\textsc{Select}) \qquad \frac{t :: u \in [\![e]\!](F_T) \qquad u' = \alpha(u)}{t :: u' \in [\![\pi_\alpha(e)]\!](F_T)}\ (\textsc{Project})$$

$$\frac{t :: u \in [\![e_1]\!](F_T) \cup [\![e_2]\!](F_T)}{t :: u \in [\![e_1 \cup e_2]\!](F_T)}\ (\textsc{Union}) \qquad \frac{t_1 :: u_1 \in [\![e_1]\!](F_T) \qquad t_2 :: u_2 \in [\![e_2]\!](F_T)}{(t_1 \otimes t_2) :: (u_1, u_2) \in [\![e_1 \times e_2]\!](F_T)}\ (\textsc{Product})$$

$$\frac{t :: u \in [\![e_1]\!](F_T) \qquad [\![e_2]\!](F_T) \not\vdash u}{t :: u \in [\![e_1 - e_2]\!](F_T)}\ (\textsc{Diff-1}) \qquad \frac{t_1 :: u \in [\![e_1]\!](F_T) \qquad t_2 :: u \in [\![e_2]\!](F_T)}{(t_1 \otimes (\ominus\, t_2)) :: u \in [\![e_1 - e_2]\!](F_T)}\ (\textsc{Diff-2})$$

$$\frac{X_T \subseteq [\![e]\!](F_T) \qquad \{t_i :: u_i\}_{i=1}^n = X_T \qquad \{\overline{t}_j :: \overline{u}_j\}_{j=1}^m = [\![e]\!](F_T) - X_T \qquad u \in g(\{u_i\}_{i=1}^n)}{(\bigotimes_{i=1}^n t_i) \otimes (\bigotimes_{j=1}^m (\ominus\, \overline{t}_j)) :: u \in [\![\gamma_g(e)]\!](F_T)}\ (\textsc{Aggregate})$$

Rule semantics $\langle . \rangle : \mathcal{U}_T \to \mathcal{U}_T, \quad [\![r]\!] : \mathcal{F}_T \to \mathcal{F}_T$

(Normalize) $\langle U_T \rangle = \{(\bigoplus_{i=1}^n t_i) :: u \mid t_1 :: u, \ldots, t_n :: u \text{ are all tagged-tuples in } U_T \text{ with the same tuple } u\}$

$$\frac{t^{\text{old}} :: u \in [\![p]\!](F_T) \qquad \langle [\![e]\!](F_T) \rangle \not\vdash u}{t^{\text{old}} :: p(u) \in [\![p \leftarrow e]\!](F_T)}\ (\textsc{Rule-1}) \qquad \frac{t^{\text{new}} :: u \in \langle [\![e]\!](F_T) \rangle \qquad [\![p]\!](F_T) \not\vdash u}{t^{\text{new}} :: p(u) \in [\![p \leftarrow e]\!](F_T)}\ (\textsc{Rule-2})$$

$$\frac{t^{\text{old}} :: u \in [\![p]\!](F_T) \qquad t^{\text{new}} :: u \in \langle [\![e]\!](F_T) \rangle}{(t^{\text{old}} \oplus t^{\text{new}}) :: p(u) \in [\![p \leftarrow e]\!](F_T)}\ (\textsc{Rule-3})$$

Stratum and Program semantics $\text{lfp}^\circ : (\mathcal{F}_T \to \mathcal{F}_T) \to (\mathcal{F}_T \to \mathcal{F}_T), \quad [\![s]\!], [\![\overline{s}]\!] : \mathcal{F}_T \to \mathcal{F}_T$

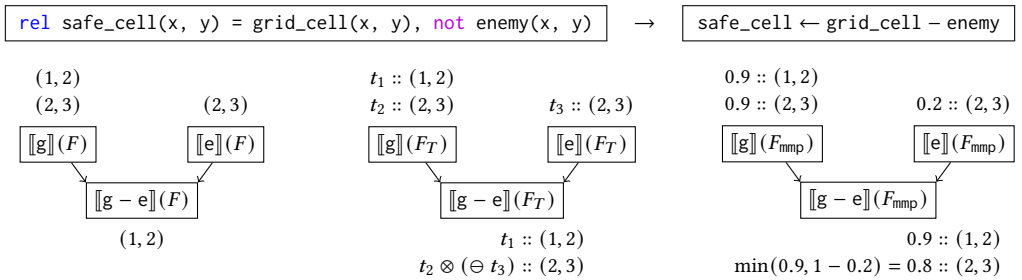
(Saturation) $F_T^{\text{old}} \doteq F_T^{\text{new}}$ iff $\forall t^{\text{new}} :: p(u) \in F_T^{\text{new}}, \exists t^{\text{old}} :: p(u) \in F_T^{\text{old}}$ such that $t^{\text{old}} \ominus t^{\text{new}}$

(Fixpoint) $\text{lfp}^\circ(h) = h \circ \cdots \circ h = h^n$ if there exists a minimum $n > 0$, such that $h^n(F_T) \doteq h^{n+1}(F_T)$

(Stratum) $[\![s]\!] = \text{lfp}^\circ(\lambda F_T . (F_T - \bigcup_{p \in P_s} F_T[p]) \cup (\bigcup_{r \in s} [\![r]\!](F_T)))$

(Program) $[\![\overline{s}]\!] = [\![s_n]\!] \circ \cdots \circ [\![s_1]\!]$, where $\overline{s} = s_1; \ldots; s_n$.

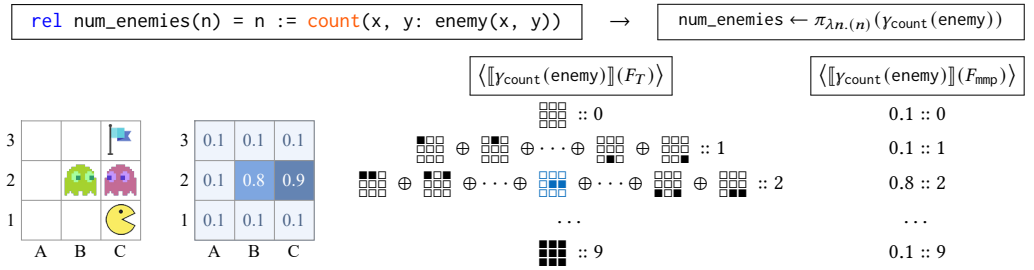
Fig. 7. Operational semantics of core fragment of RplRam.



(a) Untagged semantics (b) RplRam tagged semantics (c) RplRam with `max-min-prob`

Fig. 8. An example rule adapted from Section 2 is compiled to a RplRam rule with difference. g and e are abbreviated relation names. The graphs illustrate the evaluation of expression g − e under different semantics.

Difference and Negation. To evaluate a difference expression $e_1 - e_2$, there are two cases depending on whether a tuple u evaluated from e_1 appears in the result of e_2. If it does not, we simply propagate the tuple and its tag to the result (Diff-1); otherwise, we get $t_1 :: u$ from e_1 and $t_2 :: u$ from e_2. Instead of erasing the tuple u from the result as in untagged semantics, we propagate a tag $t_1 \otimes (\ominus\, t_2)$ with u (Diff-2). In this manner, information is not lost during negation. Fig. 8 compares the evaluations of an example difference expression under different semantics. While the tuple $(2, 3)$ is removed in the outcome with untagged semantics, it remains with the tagged semantics.



(a) Maze image (b) enemy (c) Evaluation of the aggregation expression

Fig. 9. An example counting enemies in a Pac-Man maze. Shown above are the Ripple rule and compiled RplRam rule with aggregation. (a) and (b) visualize the maze and the content of a probabilistic enemy relation. For example, we have t_{B2} :: enemy$(B, 2)$ where $t_{B2} = 0.8$. In (c), we show two normalized ($\langle . \rangle$ defined in Fig. 7) evaluation results under abstract tagged semantics and with max-min-prob provenance. Each symbol such as ▦ represents a world corresponding to our arena (■: enemy; □: no enemy). A world is a conjunction of 9 tags, e.g., ▦ $= t_{A3} \otimes (\ominus t_{A2}) \otimes \cdots \otimes (\ominus t_{C1})$. We mark the correct world ▦ which yields the answer 2.

Aggregation. Aggregators in RplRam are discrete functions g operating on set of (untagged) tuples $U \in \mathcal{U}$. They return a *set* of aggregated tuples to account for aggregators like argmax which can produce multiple outcomes. For example, we have $\text{count}(U) = \{|U|\}$. However, in the probabilistic domain, discrete symbols do not suffice. Given n tagged tuples to aggregate over, each tagged tuple can be turned on or off, resulting in 2^n distinct *worlds*. Each world is a partition of the input set U_T ($|U_T| = n$). Denoting the positive part as X_T and the negative part as $\overline{X}_T = U_T - X_T$, the tag associated with this world is a conjunction of tags in X_T and negated tags in $\overline{X}_T$. Aggregating on this world then involves applying aggregator g on tuples in the positive part X_T. This is inherently exponential if we enumerate all worlds. However, we can optimize over each aggregator and each provenance to achieve better performance. For instance, counting over max-min-prob tagged tuples can be implemented by an $O(n \log(n))$ algorithm (Appendix B), much faster than exponential. Fig. 9 demonstrates a running example and an evaluation of a counting expression under max-min-prob provenance. The resulting count can be 0-9, each derivable by multiple worlds.

Rules and Fixed-Point Iteration. Evaluating a rule $p \leftarrow e$ on database F_T concerns evaluating the expression e and merging the result with the existing facts under predicate p in F_T. The result of evaluating e may contain duplicated tuples tagged by distinct tags, owing to expressions such as union, projection, or aggregation. Therefore, we perform *normalization* on the set to join ($\oplus$) the distinct tags. From here, there are three cases to merge the newly derived tuples ($\langle\langle [\![e]\!](F_T) \rangle\rangle$) with the previously derived tuples ($[\![p]\!](F_T)$). If a fact is present only in the old or the new, we simply propagate the fact to the output. When a tuple u appears in both the old and the new, we propagate the disjunction of the old and new tag ($t^{\text{old}} \oplus t^{\text{new}}$). Combining all cases, we obtain a set of newly tagged facts under predicate p.

Recursion in RplRam is realized similar to least fixed point iteration in Datalog [2]. The iteration happens on a per-stratum basis to enforce stratified negation and aggregation. Evaluating a single step of stratum s means evaluating all the rules in s and returning the updated database. Note that we define a specialized least fixed point operator $\textbf{lfp}^{\circ}$, which stops the iteration once the whole database is *saturated*. Fig. 10 illustrates an evaluation involving recursion and database saturation. The whole database saturates on the 7th iteration, and finds the tag representing the optimal path for the PacMan to reach the goal. Termination is not universally guaranteed in RplRam due to the presence features such as value creation. But its existence can be proven in a per-provenance basis.

```
rel path(x1,y1,x3,y3) = (edge(x1,y1,x3,y3) or path(x1,y1,x2,y2) and edge(x2,y2,x3,y3)) and not enemy(x3,y3)
```

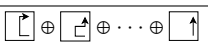
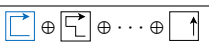
Iteration count i	1	2	3	4	5	6	7
$t^{(i)}_{\texttt{C1-C3}}$ in $F^{(i)}_T$	–		(same)		(same)		(same)
$t^{(i)}_{\texttt{C1-C3}}$ in $F^{(i)}_{\texttt{mmp}}$	–	0.1	0.1	0.2	0.2	0.9	0.9
$t^{(i)}_{\texttt{C1-C3}}$ saturated?	–	false	true	false	true	false	true
$F^{(i)}_{\texttt{mmp}}$ saturated?	false	false	false	false	false	false	true

Fig. 10. A demonstration of the fixed-point iteration to check whether actor at C1 can reach C3 without hitting an enemy (Fig. 9a). The Ripple rule to derive this is defined on the top, and we assume bidirectional edges are populated and tagged by **1**. Let $t_{\texttt{C1-C3}}$ be the tag associated with `path(C,1,C,3)`. We use a symbol like  to represent a conjunction of negated tags of `enemy` along the illustrated path, e.g. $= (\ominus t_{\texttt{C2}}) \otimes (\ominus t_{\texttt{C3}})$. 2nd iter is the first time $t_{\texttt{C1-C3}}$ is derived, but the path is blocked by an enemy. On 6th iter, the best path is derived in the tag. After that, under the `max-min-prob` provenance, both the tag $t_{\texttt{C1-C3}}$ and the database $F_{\texttt{mmp}}$ are saturated, causing the iteration to stop. Compared to untagged semantics in Datalog which will stop after 4 iterations, RplRam with `mmp` saturates slower but allowing to explore better reasoning chains.

For example, it is easy to show that if a program terminates under untagged semantics, then it terminates under tagged semantics with `max-min-prob` provenance.

4.4 External Interface and Execution Pipeline

Thus far, we have only illustrated the `max-min-prob` provenance, in which the tags are probabilities. There are other probabilistic provenances with more complex tags such as proof trees or boolean formulae. We therefore introduce, for each provenance T, an *input tag* space I, an *output tag* space O, a *tagging function* $\tau : I \to T$, and a *recover function* $\rho : T \to O$. For instance, all probabilistic provenances share the same input and output tag spaces $I = O = [0, 1]$ for a unified interface, while the internal tag spaces T could be different. We call the 4-tuple (I, O, τ, ρ) the *external interface* for a provenance T. The whole execution pipeline is then illustrated below:

In the context of a Ripple application, an EDB is provided in the form $F_{\texttt{option<I>}}$. During the *tagging phase*, τ is applied to each input tag to obtain F_T, following which the RplRam program operates on F_T. For convenience, not all input facts need to be tagged—untagged input facts are simply associated by the tag **1** in F_T. In the *recovery phase*, ρ is applied to obtain F_O, the IDB that the whole pipeline returns. Ripple allows the user to specify a set of *output relations* and ρ is only applied to tags under such relations to avoid redundant computations.

4.5 Differentiable Reasoning with Provenance

We now elucidate how provenance also supports differentiable reasoning. Let all the probabilities in the EDB form a vector $\vec{r} \in \mathbb{R}^n$, and the probabilities in the resulting IDB form a vector $\vec{y} \in \mathbb{R}^m$. Differentiation concerns deriving output probabilities $\vec{y}$ as well as the derivative $\nabla\vec{y} = \frac{\partial \vec{y}}{\partial \vec{r}} \in \mathbb{R}^{m \times n}$.

In Ripple, one can obtain these using a *differentiable provenance* (DP). DPs share the same external interface—let the input tag space $I = [0, 1]$ and output tag space O be the space of *dual-numbers* $\mathbb{D}$ (Fig. 12). Now, each input tag $r_i \in [0, 1]$ is a probability, and each output tag $\hat{y}_j = (y_j, \nabla y_j)$ encapsulates the output probability y_j and its derivative w.r.t. inputs, ∇y_j. From here, we can obtain our expected output $\vec{y}$ and $\nabla\vec{y}$ by stacking together y_j-s and ∇y_j-s respectively.

Ripple provides 8 configurable built-in DPs with different empirical advantages in terms of runtime efficiency, reasoning granularity, and performance. In this section, we elaborate upon 3 simple but versatile DPs, whose definitions are shown in Fig. 11. In the following discussion, we

Provenance	T	$\mathbf{0}$	$\mathbf{1}$	$t_1 \oplus t_2$	$t_1 \otimes t_2$	$\ominus\, t$	$t_1 \ominus t_2$	$\tau(r_i)$	$\rho(t)$
`diff-max-min-prob`	$\mathbb{D}$	$\hat{0}$	$\hat{1}$	$\max(t_1, t_2)$	$\min(t_1, t_2)$	$\hat{1} - t$	$t_1^{\text{fst}} == t_2^{\text{fst}}$	$(r_i, \vec{\mathbf{e}}_i)$	t
`diff-add-mult-prob`	$\mathbb{D}$	$\hat{0}$	$\hat{1}$	$\text{clamp}(t_1 + t_2)$	$t_1 \cdot t_2$	$\hat{1} - t$	true	$(r_i, \vec{\mathbf{e}}_i)$	t
`diff-top-k-proofs`	Φ	$\emptyset$	$\{\emptyset\}$	$t_1 \vee_k t_2$	$t_1 \wedge_k t_2$	$\neg_k\, t$	$t_1 == t_2$	$\{\{\text{pos}(i)\}\}$	$\text{WMC}(t, \Gamma)$

Fig. 11. Definitions of three differentiable provenances.

$$\hat{a}_i = (a_i, \nabla a_i) \in \mathbb{D} \qquad \hat{a}_1 + \hat{a}_2 = (a_1 + a_2, \nabla a_1 + \nabla a_2) \qquad \min(\hat{a}_1, \hat{a}_2) = \hat{a}_i, \text{ where } i = \text{argmin}_i(a_i)$$
$$\hat{0} = (0, \vec{0}) \qquad \hat{a}_1 \cdot \hat{a}_2 = (a_1 \cdot a_2, a_2 \cdot \nabla a_1 + a_1 \cdot \nabla a_2) \qquad \max(\hat{a}_1, \hat{a}_2) = \hat{a}_i, \text{ where } i = \text{argmax}_i(a_i)$$
$$\hat{1} = (1, \vec{0}) \qquad -\hat{a}_1 = (-a_1, -\nabla a_1) \qquad \text{clamp}(\hat{a}_1) = (\text{clamp}_0^1(a_1), \nabla a_1)$$

Fig. 12. Operations on dual-number $\mathbb{D} \triangleq [0, 1] \times \mathbb{R}^n$, where n is the number of input probabilities.

$$\text{(Variable)} \quad i \in 1 \ldots n \qquad \varphi_1 \vee_k \varphi_2 = \text{top}_k(\varphi_1 \cup \varphi_2)$$
$$\text{(Literal)} \quad \nu ::= \text{pos}(i) \mid \text{neg}(i) \qquad \varphi_1 \wedge_k \varphi_2 = \text{top}_k(\{\, \eta \mid (\eta_1, \eta_2) \in \varphi_1 \times \varphi_2, \eta = \eta_1 \cup \eta_2, \eta \text{ no conflict} \,\})$$
$$\text{(Proof)} \quad \eta ::= \{\nu_1, \nu_2, \ldots\} \qquad \neg_k\, \varphi = \text{top}_k(\text{cnf2dnf}(\{\{\neg \nu \mid \nu \in \eta\} \mid \eta \in \varphi\}))$$
$$\text{(Formula)} \quad \Phi \ni \varphi ::= \{\eta_1, \eta_2, \ldots\} \qquad \Gamma(i) = (r_i, \vec{\mathbf{e}}_i)$$

Fig. 13. Definitions used for `diff-top-k-proofs` provenance.

use r_i to denote the i-th element of $\vec{r}$, where i is called a *variable* (ID). Vector $\vec{\mathbf{e}}_i \in \mathbb{R}^n$ is the standard basis vector where all entries are 0 except the i-th entry.

4.5.1 `diff-max-min-prob` *(*`dmmp`*).* This provenance is the differentiable version of `mmp`. When obtaining r_i from an input tag, we transform it into a dual-number by attaching $\vec{\mathbf{e}}_i$ as its derivative. Note that throughout the execution, the derivative will always have at most one entry being non-zero and, specifically, 1 or -1. The saturation check is based on equality of the probability part only, so that the derivative does not affect termination. All of its operations can be implemented by algorithms with time complexity $O(1)$, making it extremely runtime-efficient.

4.5.2 `diff-add-mult-prob` *(*`damp`*).* This provenance has the same internal tag space, tagging function, and recover function as `dmmp`. As suggested by its name, its disjunction and conjunction operations are just + and · for dual-numbers. When performing disjunction, we clamp the real part of the dual-number obtained from performing +, while keeping the derivative the same. The saturation function for `damp` is designed to always returns true to avoid non-termination. But this decision makes it less suitable for complex recursive programs. The time complexity of operations in `damp` is $O(n)$, which is slower than `dmmp` is but still very efficient in practice.

4.5.3 `diff-top-k-proofs` *(*`dtkp`*).* This is the extended *top-k proofs* semiring from [28]. Shown in Fig. 11 and 13, the tags of `dtkp` are boolean formulas $\varphi \in \Phi$ in *disjunctive normal form* (DNF). Each conjunctive clause in the DNF is called a *proof* η. A formula can contain at most k proofs, where k is a tunable hyper-parameter. Throughout execution, boolean formulas are propagated with $\vee_k$, $\wedge_k$, and $\neg_k$, which resemble *or*, *and*, and *not* on DNF formulas. At the end of these computations, top_k is applied to keep only k proofs with the highest *proof probability*:

$$\Pr(\eta) = \textstyle\prod_{\nu \in \eta} \Pr(\nu), \quad \Pr(\text{pos}(i)) = r_i, \quad \Pr(\text{neg}(i)) = 1 - r_i. \tag{1}$$

When merging two proofs during $\wedge_k$, there might be conflicting literals, e.g. $\text{pos}(i)$ and $\text{neg}(i)$, in which case we remove the whole proof. To take negation $\neg_k$ on φ, we first negate all the literals to obtain a *conjunctive normal form* (CNF) equivalent to $\neg\varphi$. Then we perform cnf2dnf operation (conflict check included) to convert it back to a DNF. To obtain the output dual-number $\hat{y}_j$ from a DNF formula φ_j, the tag for j-th output tuple, we adapt a differentiable *weighted-model-counting* (WMC) procedure [38]. WMC computes the weight of a boolean formula φ_j given the weights of individual variables. Concretely, $\hat{y}_j = \text{WMC}(\varphi_j, \Gamma)$ where $\Gamma(i) = (r_i, \vec{\mathbf{e}}_i)$ is the mapping from

variables to their dual-numbers. Note that WMC is #P-complete, and is the main contributor to the runtime when using this provenance. The tunable k enables the user to balance between runtime and reasoning granularity. Detailed runtime analysis is shown in Appendix B.

4.6 Practical Considerations

We finally discuss some practical aspects concerning RplRam extensions and provenance selection.

Additional Features. We only presented the syntax and semantics of the core fragment of RplRam. RplRam additionally supports the following useful features: 1) sampling operations, and the provenance extension supporting them, 2) group-by aggregations, 3) tag-based early removal and its extension in provenance, and 4) mutually exclusive tags in `dtkp`. We formalize these in Appendix B.

Provenance Selection. A natural question is how to select a differentiable provenance for a given Ripple application. Based on our empirical evaluation in Section 6, `dtkp` is often the best performing one, and setting $k = 3$ is usually a good choice for both runtime efficiency and learning performance. This suggests that a user can start with `dtkp` as the default. In general, the provenance selection process is similar to the process of hyperparameter tuning common in machine learning.

5 IMPLEMENTATION

We implement the core Ripple system with 45K LoC of Rust. The LoC of individual modules is shown in Table 1. Within the compiler, there are two levels of intermediate representations, front-IR and back-IR, between the surface language and the RplRam language. In front-IR, we perform analyses such as type inference and transformations such as desugaring. In back-IR, we generate query plans and apply optimizations. The runtime operates directly on RplRam and is based on semi-naive evaluation specialized for tagged semantics. There are *dynamic* and *static* runtimes for interpreted and compiled RplRam programs. Currently, all computation is on CPU only, but can be parallelized per-batch for machine learning.

Module	LoC
Compiler	19K
Runtime	16K
Interpreter	2K
`rippley`	4K

Table 1. LoC of core Ripple modules.

Ripple can be used through different interfaces such as interpreter, compiler, and interactive terminal. It also provides language bindings such as `rippley` for Python and `ripple-wasm` for WebAssembly and JavaScript. With `rippley`, Ripple can be seamlessly integrated with machine learning frameworks such as PyTorch, wherein the `rippley` module is treated just like any other PyTorch module. When `jit` is specified in `rippley`, Ripple programs can be *just-in-time* (JIT) compiled to Rust, turned into binaries, and dynamically loaded into the Python environment.

Ripple provides 18 built-in provenances (4 for discrete reasoning, 6 for probabilistic, and 8 for differentiable). The WMC algorithm is implemented using *sentential decision diagram* (SDD) [16] with naive bottom-up compilation. We allow each provenance to provide its own implementation for operations such as aggregation and sampling. This gives substantial freedom and enables optimizations for complex operations. Our provenance framework is also interfaced with `rippley`, allowing to quickly create and test new provenances in Python.

6 EVALUATION

We evaluate the Ripple language and framework on a benchmark suite comprising eight neurosymbolic applications. Our evaluation aims to answer the following research questions:

RQ1 How expressive is Ripple for solving diverse neurosymbolic tasks?
RQ2 How do Ripple's solutions compare to state-of-the-art baselines in terms of accuracy?
RQ3 Is the differentiable reasoning module of Ripple runtime efficient?
RQ4 Is Ripple effective at improving generalizability, interpretability, and data-efficiency?
RQ5 What are the failure modes of Ripple solutions and how can we mitigate them?

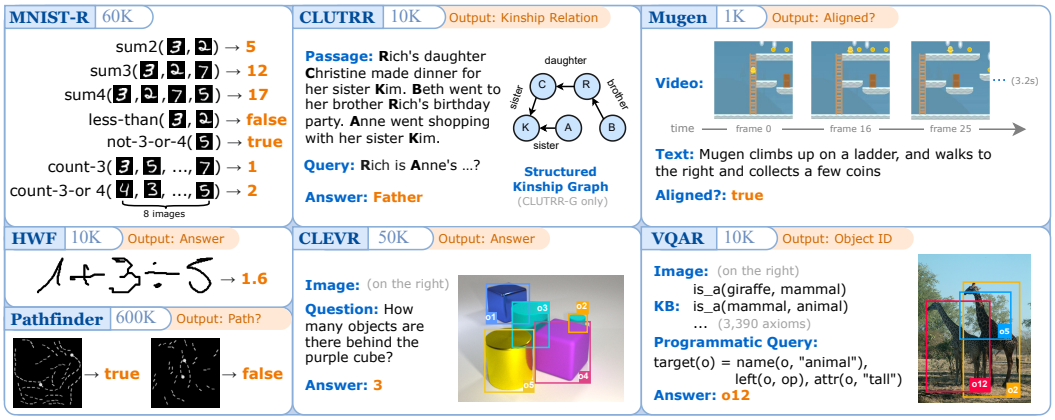


Fig. 14. Visualization of benchmark tasks. Beside the name of each task we specify the size of the training dataset and the output domain. PacMan-Maze is omitted since it has been shown in Section 2.

In the following sections, we first introduce the benchmark tasks and the chosen baselines for each task (Section 6.1). Then, we answer **RQ1** to **RQ5** in Section 6.2 to Section 6.6 respectively. All Ripple related and runtime related experiments were conducted on a machine with two 20-core Intel Xeon CPUs, four GeForce RTX 2080 Ti GPUs, and 768 GB RAM.

6.1 Benchmarks and Baselines

We present an overview of our benchmarks in Fig. 14. They cover a wide spectrum of tasks involving perception and reasoning. The input data modality ranges from images and videos to natural language texts and knowledge bases (KB). The size of training dataset is also presented in the figure. We next elaborate on the benchmark tasks and their corresponding baselines.

MNIST-R: *A Synthetic MNIST Test Suite.* This synthetic benchmark is curated by augmenting the MNIST hand-written digit dataset [33], inspired by DeepProbLog (DPL) [38]. It is designed to test various features provided by Ripple, such as negation and aggregation. Each task takes as input one or multiple MNIST images, and requires performing simple arithmetic (sum2, sum3, sum4), comparison (less-than), negation (not-3-or-4), or counting (count-3, count-3-or-4) over depicted digits. For count-3 and count-3-or-4, we count digits from a set of 8 images. For this test suite, we pick DPL [38] and a convolutional neural network (CNN) based model as the baselines.

HWF: *Hand-Written Formula Parsing and Evaluation.* HWF, proposed in [35], concerns parsing and evaluating hand-written formulas. The formula is provided in the form of a sequence of images, where each image represents either a digit (0-9) or an arithmetic symbol ($+, -, \times, \div$). Formulas are well-formed according to a grammar, and there is no formula with divide-by-zero. The size of the formulas ranges from 1 to 7, and is indicated as part of the input. The goal is to evaluate the formula to obtain a rational number as the result. We choose from [35] the baselines NGS-m-BS, NGS-RL, and NGS-MAPO, which are *neurosymbolic methods* designed specifically for this task.

Pathfinder: *Image Classification with Long-Range Dependency.* In this task from [49], the input is an image containing two dots that are possibly connected by curved and dashed lines. The goal is to tell whether the dots are connected. There are two subtasks, Path and Path-X, where Path contains 32×32 images and Path-X contains 128×128 ones. We pick as baselines CNN and Transformer based models, and we include state-of-the-art models S4 [24], S4* [25], and SGConv [36].

PacMan-Maze: *Playing PacMan Maze Game.* As presented in Section 2, this task tests an agent's ability to recognize entities in an image and plan the path for the PacMan to reach the goal. An

RL environment provides the game state image as input and the agent must plan the optimal action {`up`, `down`, `left`, `right`} to take at each step. There is no "training dataset" as the environment is randomized for every session. We pick as baseline a CNN based Deep-Q-Network (DQN). Unlike other tasks, here we use the "success rate" metric for evaluation, i.e., among 1000 game sessions, we measure the number of times the PacMan reach the goal within a certain time-budget.

CLUTRR: *Kinship Reasoning from Natural Language Context.* In this task from [47], the input contains a natural language (NL) passage about a set of characters. Each sentence in the passage hints at kinship relations. The goal is to infer the relationship between a given pair of characters. The target relation must be deduced through a reasoning chain and is not stated explicitly in the passage. Our baseline models include RoBERTa [37], BiLSTM [22], GPT-3-FT (fine-tuned), GPT-3-ZS (zero-shot), and GPT-3-FS (5-shot) [8]. In an alternative setup, CLUTRR-G, instead of the NL passage, the structured kinship graph corresponding to the NL passage is provided, making it a *Knowledge Graph Reasoning* problem. For CLUTRR-G, we pick GAT [52] and CTP [40] as baselines.

Mugen: *Video-Text Alignment and Retrieval.* Mugen [27] is based on a game called CoinRun [12]. In the video-text alignment task, the input contains a 3.2 second long video of gameplay footage and a short NL paragraph describing events happening in the video. The goal is to compute a similarity score representing how "aligned" they are. There are two subsequent tasks, Video-to-Text Retrieval (VTR) and Text-to-Video Retrieval (TVR). In TVR, the input is a piece of text and a set of 16 videos, and the goal is to retrieve the video that aligns with the text the most. In VTR, the goal is to retrieve text from video. We compare our method with SDSC [27].

CLEVR: *Compositional Language and Elementary Visual Reasoning [30].* In this visual question answering (VQA) task, the input contains a rendered image of geometric objects and a NL question that asks about counts, attributes, and relationships of objects. The goal is to answer the question based on the scene shown in the image. We pick as baselines NS-VQA [55] and NSCL [39], which are *neurosymbolic methods* designed specifically for this task.

VQAR: *Visual-Question-Answering with Common-Sense Reasoning.* This task, like CLEVR, also concerns VQA but with important differences. First, it contains real-life images from the GQA dataset [29]. Secondly, we assume that the queries are in their programmatic form, asking to retrieve objects in the image. Lastly, there is an additional input in the form of a common-sense knowledge base (KB) [19] containing triplets such as (giraffe, is-a, animal), in order to perform common-sense reasoning. The baselines for this task are NMNs [3] and LXMERT [48].

6.2 RQ1: Our Solutions and Expressivity

To answer **RQ1**, we demonstrate our Ripple solutions to the benchmark tasks (Table 2). For each task, we specify the interface relations which serve as the bridge between the neural and symbolic components. The neural modules process the perceptual input and their outputs are mapped to (probabilistic) facts in the interface relations. Our Ripple programs subsequently take these facts as input and perform the described reasoning to produce the final output. As shown by the *features* column, our solutions use all of the core features provided by Ripple.

The complete Ripple program for each task is provided in Appendix C. These programs are succinct, as indicated by the LoCs in the last column of Table 2. We highlight three tasks, HWF, Mugen, and CLEVR, to demonstrate Ripple's expressivity. For HWF, the Ripple program consists of a formula parser. It is capable of parsing probabilistic input symbols according to a context free grammar for simple arithmetic expressions. For Mugen, the Ripple program is a *temporal specification checker*, where the specification is extracted from NL text to match the sequential events excerpted from the video. For CLEVR, the Ripple program is an interpreter for CLEVR-DSL, a domain specific functional language introduced in the CLEVR dataset [30].

Task	Input	Neural Net	Interface Relation(s)	Ripple Program	Features			LoC
					R	N	A	
MNIST-R	Images	CNN	`digit(`*id*, *digit*`)`	Arithmetic, comparison, negation, and counting.		✔	✔	2†
HWF	Images	CNN	`symbol(`*id*, *symbol*`)` `length(`*len*`)`	Parses and evaluates formula over recognized symbols.	✔			39
Pathfinder	Image	CNN	`dot(`*id*`)` `dash(`*from_id*, *to_id*`)`	Checks if the dots are connected by dashes.	✔			4

PacMan-Maze	Image	CNN	`actor(`*x*, *y*`)`
			`enemy(`*x*, *y*`)`
			`goal(`*x*, *y*`)`
				Plans the optimal action by finding an enemy-free path from actor to goal.	✔	✔	✔	31

CLUTRR (-G)	NL	RoBERTa	`kinship(`*rela*, *sub*, *obj*`)`	Deduces queried relationship by recursively applying learnt transitivity rules.	✔			8
	Query*	–	`question(`*sub*, *obj*`)`					
	Rule	–	`transitive(`r_1, r_2, r_3`)`					
Mugen	Video	S3D	`action(`*frame*, *action*, *mod*`)`	Checks if events specified in NL text match the actions recognized from the video.	✔	✔	✔	46
	NL	DistilBERT	`expr(`*expr_id*, *action*`)` `mod(`*expr_id*, *mod*`)`					
CLEVR	Image	FastRCNN	`obj_attr(`*obj_id*, *attr*, *val*`)` `obj_rela(`*rela*, o_1, o_2`)`	Interprets CLEVR-DSL program (extracted from question) on scene graph (extracted from image).	✔	✔	✔	51
	NL	BiLSTM	`filter_expr(`*e*, *ce*, *attr*, *val*`)` `count_expr(`*e*, *ce*`)`, …					

VQAR	Image	FastRCNN	`obj_name(`*obj_id*, *name*`)`
			`obj_attr(`*obj_id*, *val*`)`
			`obj_rela(`*rela*, o_1, o_2`)`
				Evaluates query over scene graphs (extracted from image) with the aid of common-sense knowledge base (KB).	✔			42

	KB*	–	`is_a(`*name1*, *name2*`)`, …					

Table 2. Characteristics of Ripple solutions for each task. Structured input which is not learnt is denoted by *. Neural models used are RoBERTa [37], DistilBERT [44], and BiLSTM [22] for natural language (NL), CNN and FastRCNN [21] for images, and S3D [54] for video. We show the three key features of Ripple used by each solution: (R)ecursion, (N)egation, and (A)ggregation. †: For MNIST-R, the LoC is 2 for every subtask.

In addition to diverse kinds of perceptual data and reasoning patterns, the Ripple programs are applied in a variety of learning settings. As shown in Section 2, the program for PacMan-Maze is used in a *reinforcement learning* environment. For CLUTRR(-G), learnable weights are attached to `transitive` facts such as `transitive(FATHER, MOTHER, GRANDMOTHER)`. In this case, the numerous transitivity rules for kinship reasoning are not specified by the users, but are learnt from the data. This is akin to *rule learning* in inductive logic programming. For Mugen, our program is trained in a *contrastive learning* setup, since it requires to maximize similarity scores between aligned video-text pairs but minimize that for un-aligned ones.

6.3 RQ2: Performance and Accuracy

To answer **RQ2**, we evaluate the performance and accuracy of our methods in terms of two aspects: 1) the best performance of our solutions compared to existing baselines, and 2) the performance of our solutions with different provenance structures (`dmmp`, `damp`, `dtkp` with different k).

We start with comparing our solutions against selected baselines on all the benchmark tasks, as shown in Fig. 15, Table 3, and Fig. 17. First, we highlight two particular applications, PacMan-Maze and CLUTRR, which benefit the most from our solution. For PacMan-Maze, compared to DQN, we obtain a 1,000× speed-up in terms of training episodes, and a near perfect success rate of 99.4%. For CLUTRR, we obtain a 20% improvement over selected baselines, which includes GPT-3-FT, the state-of-the-art large language model fine-tuned on the CLUTRR dataset. Next, for tasks such as HWF and CLEVR, our solutions attain comparable performance, even when compared against neurosymbolic baselines NGS-m-BS, NSCL, and NS-VQA, specifically designed for each task. On

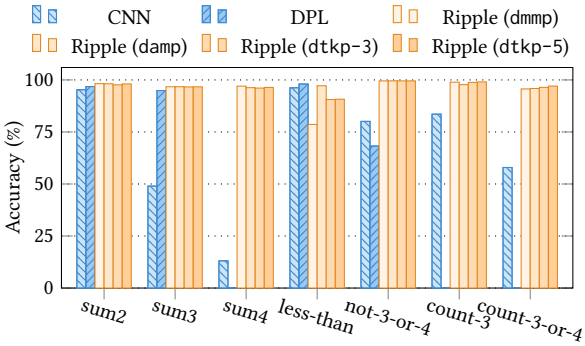


Fig. 15. MNIST-R suite accuracy comparison.

Method	Ripple			DQN
	dmmp	damp	dtkp-1	
Succ Rate	8.80%	7.84%	**99.40%**	84.90%
#Episodes	50	50	**50**	50K

Table 3. PacMan-Maze performance.

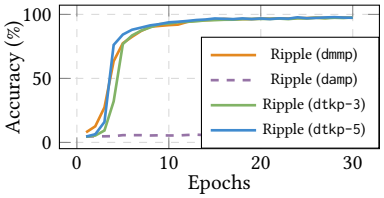


Fig. 16. HWF learning curve.

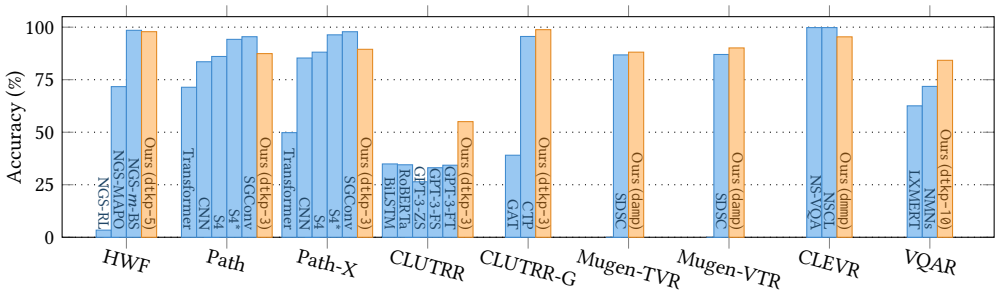


Fig. 17. Overall benchmark accuracy comparison. The best-performing provenance structure for our solution is indicated for each task. Among the shown tasks, `dtkp` performs the best on 6 tasks, `damp` on 2, and `dmmp` on 1.

Path and Path-X, our solution obtains a 4% accuracy gain over our underlying model CNN and even outperforms a carefully designed transformer based model S4.

The performance of the Ripple solution for each task is dependent on the chosen provenance structure. As can be seen from Table 3 and Figs. 15–17, although `dtkp` is generally the best-performing one, each presented provenance finds its special place, e.g., PacMan-Maze and VQAR for `dtkp`, less-than (MNIST-R) and Mugen for `damp`, and HWF and CLEVR for `dmmp`. In conclusion, allowing configurable provenance helps tailor our methods towards different applications.

6.4 RQ3: Runtime Efficiency

We evaluate the runtime efficiency of Ripple solutions with different provenance structures and compare it against baselines with logical reasoning module. As shown in Table 4, Ripple achieves substantial speed-up over DeepProbLog (DPL) on MNIST-R tasks. DPL is a probabilistic programming system based on Prolog using exact probabilistic reasoning. As an example, on sum4, DPL takes 40 days to finish only 4K training samples, showing that it is prohibitively slow to use in practice. On the contrary, Ripple solutions can finish a training epoch (15K samples) in minutes without sacrificing testing accuracy (according to Fig. 15). For HWF, Ripple achieves comparable runtime efficiency, even when compared against the hand-crafted and specialized NGS-m-BS method.

Comparing among provenance structures, we see significant runtime blowup when increasing k for `dtkp`. This is expected as increasing k results in larger boolean formula tags, making the WMC procedure exponentially slower. In practice, we find $k = 3$ for `dtkp` to be a good balance point between runtime efficiency and reasoning granularity. In fact, `dtkp` generalizes DPL, as one can set an extremely large $k \geq 2^n$ (n is the total number of input facts) for exact probabilistic reasoning.

Task	Ripple				Baseline
	dmmp	damp	dtkp-3	dtkp-10	
sum2	**34**	88	72	185	21,430 (DPL)
sum3	**34**	**119**	71	1,430	30,898 (DPL)
sum4	**34**	154	77	4,329	timeout (DPL)
less-than	35	**42**	34	43	2,540 (DPL)
not-3-or-4	37	**33**	**33**	**34**	3,218 (DPL)
HWF	89	107	**120**	8,435	79 (NGS-*m*-BS)
CLEVR	**1,964**	1,618	2,325	timeout	–

Table 4. Runtime efficiency comparison on selected benchmark tasks. Numbers shown are average training time (sec.) per epoch. Our variants attaining the best accuracy are indicated in bold.

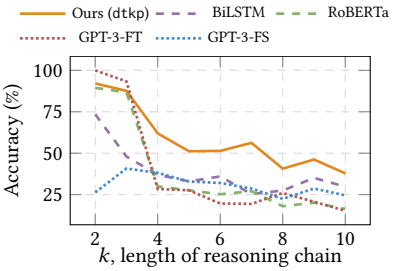


Fig. 18. Systematic generalizability on CLUTRR dataset.

Fig. 19. The predicted most likely (action, mod) pair for example video segments from Mugen dataset.

%Train	Ripple	NGS		
	dtkp-5	RL	MAPO	*m*-BS
100%	97.85	3.4	71.7	98.5
50%	95.7	3.6	9.5	95.7
25%	92.95	3.5	5.1	93.3

Table 5. Testing accuracy of various methods on HWF when trained with only a portion of the data. Numbers are in percentage (%).

6.5 RQ4: Generalizability, Interpretability, and Data-Efficiency

We now consider other important desirable aspects of machine learning models besides accuracy and runtime, such as generalizability on unseen inputs, interpretability of the outputs, and data-efficiency of the training process. For brevity, we focus on a single benchmark task in each case.

We evaluate Ripple's generalization ability for the CLUTRR task. Each data-point in CLUTRR is annotated with a parameter k denoting the length of the reasoning chain to infer the target kinship relation. To test different solutions' *systematic generalizability*, we train them on data-points with $k \in \{2, 3\}$ and test on data-points with $k \in \{2, \ldots, 10\}$. As shown in Fig. 18, the neural baselines suffer a steep drop in accuracy on the more complex unseen instances, whereas the accuracy of Ripple's solution degrades more slowly, indicating that it is able to generalize better.

Next, we demonstrate Ripple's interpretability on the Mugen task. Although the goal of the task is to see whether a video-text pair is aligned, the perceptual model in our method extracts interpretable symbols, i.e., the action of the controlled character at a certain frame. Fig. 19 shows that the predicted (action, mod) pairs perfectly match the events in the video. This indicates that our solution not only tells "how aligned", but also "why" they are aligned.

Lastly, we evaluate Ripple's data-efficiency on the HWF task, using lesser training data (Table 5). While methods such as NGS-MAPO suffer significantly when trained on less data, Ripple's testing accuracy decreases slowly, and is comparable to the data-efficiency of the state-of-the-art neurosymbolic NGS-*m*-BS method. PacMan-Maze task also demonstrates Ripple's data-efficiency, as it takes much less training episodes than DQN does, while achieving much higher success rate.

6.6 RQ5: Analysis of Failure Modes

Compared to purely neural models, Ripple solutions provide more transparency, allowing programmers to debug effectively. By manually checking the interface relations, we observed that the main source of error lies in inaccurate predictions from the neural components. For example, the RoBERTa model for CLUTRR correctly extracts only 84.69% of kinship relations. There are two potential causes—either the neural component is not powerful enough, or our solution is not

providing adequate supervision to train it. The former can be addressed by employing better neural architectures or more data. The latter can be mitigated in different ways, such as tuning the selected provenance or incorporating integrity constraints (discussed in Section 3.2) into training and/or inference. For instance, in PacMan-Maze, the constraint that "there should be no more than one *goal* in the arena" (shown in Fig. 28 in Appendix C) helps to improve robustness.

In conclusion, we compared Ripple applications to state-of-the-art solutions for diverse and challenging tasks. Ripple enables to improve upon a number of valuable metrics, including accuracy, data-efficiency, interpretability, and generalizability. Ripple is thus able to leverage important concepts in each field while simultaneously generalizing across all of them. Lastly, akin to analogous approaches in other domains (e.g., virtual machines and SAT solvers), further improvements in Ripple serve to benefit all of the applications to which it is applied.

7 RELATED WORK

We survey related work in four different but overlapping domains: Datalog and logic programming, probabilistic programming, differentiable programming, and neurosymbolic methods.

Datalog, Provenance, and Logic Programming. A variety of Datalog based systems [4, 45] have been built for program analysis and enterprise database applications. The provenance semiring framework is theorized in [15, 23], and is deployed to variants of Datalog [32], supporting applications such as program synthesis [46]. Ripple is also a Datalog based language and employs an extended provenance semiring framework for configurable differentiable reasoning.

Probabilistic Programming is a paradigm for programmers to model distributions and perform probabilistic sampling and inference. Such systems include ProbLog [18], Pyro [6], Turing [20], and PPL [50]. When integrated with modern ML systems, they are well suited for statistical modeling and building generative models. Ripple also supports ProbLog style exact probabilistic inference as one instantiation of our provenance framework. But advanced statistical sampling and generative modeling is not yet supported and left for future work.

Differentiable Programming (DP) systems seek to empower programmers to write code that is differentiable. Common practices for DP include symbolic differentiation and automatic differentiation (auto-diff) [5], resulting in popular machine learning frameworks such as PyTorch [42], TensorFlow [1], and JAX [7]. However, most of the differentiable programming systems are designed for real-valued functions. Ripple is also a differentiable programming system, but with a focus on programming with discrete, logical, and relational symbols.

Neurosymbolic Methods have emerged to incorporate symbolic reasoning into existing learning systems. Their success has been demonstrated in a number of individual problems [11, 35, 39, 40, 53, 55]. Such methods usually build their specialized symbolic reasoning components, some of which are instantiations of our general provenance framework. DeepProbLog (DPL) [38] and TensorLog [13] can be seen as unified neurosymbolic frameworks. Ripple draws many inspirations from DPL, and in addition, offers a much more scalable, customizable, and easy-to-use language and framework to solve a wide range of learning tasks.

8 CONCLUSION

We presented Ripple, a neurosymbolic programming language for integrating deep learning and logical reasoning. We introduced a declarative language and a reasoning framework based on provenance computations. We showed that our framework is practical by applying it to a variety of machine learning tasks. In particular, our experiments show that Ripple solutions are comparable and even supersede many existing baselines.

In the future, we plan to extend Ripple in three aspects: 1) Supporting more machine learning paradigms, such as generative modeling, open domain reasoning, in-context learning, and adversarial learning. 2) Further enhancing the usability, efficiency, and expressiveness of Ripple's language and framework. We intend to provide bindings to other machine learning frameworks such as TensorFlow [1] and JAX [7], leverage hardware such as GPUs to accelerate computation, and implement programming constructs such as algebraic data-types to make Ripple more expressive. 3) Applying Ripple to real-world and safety-critical domains. For instance, we intend to integrate it with the CARLA driving simulator [17] to specify soft temporal constraints for autonomous driving systems. We also intend to apply Ripple in the medical domain for explainable disease diagnosis from electronic health records (EHR) data.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases: the logical level.* Addison-Wesley Longman Publishing Co., Inc.

[3] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 39–48.

[4] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1371–1382.

[5] Atilim Gunes Baydin, Barak A. Pearlmutter, and Alexey Andreyevich Radul. 2015. Automatic differentiation in machine learning: a survey. *CoRR* abs/1502.05767 (2015). arXiv:1502.05767

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

[7] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.*

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[9] Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, Yisong Yue, et al. 2021. Neurosymbolic Programming. *Foundations and Trends® in Programming Languages* 7, 3 (2021).

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv:2107.03374* (2021).

[11] Xinyun Chen, Chen Liang, Adams Wei Yu, Denny Zhou, Dawn Song, and Quoc V. Le. 2020. Neural Symbolic Reader: Scalable Integration of Distributed and Symbolic Representations for Reading Comprehension. In *International Conference on Learning Representations (ICLR)*.

[12] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. 2019. Quantifying generalization in reinforcement learning. In *International Conference on Machine Learning (ICML)*. PMLR, 1282–1289.

[13] William W. Cohen, Fan Yang, and Kathryn Rivard Mazaitis. 2017. TensorLog: Deep Learning Meets Probabilistic DBs.

[14] Andrew Cropper, Sebastijan Dumancic, Richard Evans, and Stephen H. Muggleton. 2022. Inductive logic programming at 30. *Mach. Learn.* 111, 1 (2022).

[15] Katrin M. Dannert, Erich Grädel, Matthias Naaf, and Val Tannen. 2021. Semiring Provenance for Fixed-Point Logic. In *29th EACSL Annual Conference on Computer Science Logic (CSL 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 183)*, Christel Baier and Jean Goubault-Larrecq (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 17:1–17:22.

[16] Adnan Darwiche. 2011. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two* (Barcelona, Catalonia, Spain) *(IJCAI'11)*. AAAI Press, 819–826.

[17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator.

[18] Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van den Broeck, Jonas Vlasselaer, and Luc De Raedt. 2015. ProbLog2: Probabilistic Logic Programming. In *Machine Learning and Knowledge Discovery in Databases*, Albert Bifet, Michael May, Bianca Zadrozny, Ricard Gavalda, Dino Pedreschi, Francesco Bonchi, Jaime Cardoso, and Myra Spiliopoulou (Eds.). Springer International Publishing, Cham, 312–315.

[19] Difei Gao, Ruiping Wang, Shiguang Shan, and Xilin Chen. 2019. From Two Graphs to N Questions: A VQA Dataset for Compositional Reasoning on Vision and Commonsense. *CoRR* abs/1908.02962 (2019). arXiv:1908.02962

[20] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: a language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*. 1682–1690.

[21] Ross B. Girshick. 2015. Fast R-CNN. *CoRR* abs/1504.08083 (2015). arXiv:1504.08083

[22] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. Ieee, 6645–6649.

[23] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*.

[24] Albert Gu, Karan Goel, and Christopher Ré. 2021. Efficiently Modeling Long Sequences with Structured State Spaces. *CoRR* abs/2111.00396 (2021). arXiv:2111.00396

[25] Albert Gu, Isys Johnson, Aman Timalsina, Atri Rudra, and Christopher Ré. 2022. How to train your hippo: State space models with generalized orthogonal basis projections. *arXiv preprint arXiv:2206.12037* (2022).

[26] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017).

[27] Thomas Hayes, Songyang Zhang, Xi Yin, Guan Pang, Sasha Sheng, Harry Yang, Songwei Ge, Qiyuan Hu, and Devi Parikh. 2022. MUGEN: A Playground for Video-Audio-Text Multimodal Understanding and GENeration.

[28] Jiani Huang, Ziyang Li, Binghong Chen, Karan Samel, Mayur Naik, Le Song, and Xujie Si. 2021. Scallop: From probabilistic deductive databases to scalable differentiable reasoning. *Advances in Neural Information Processing Systems* 34 (2021), 25134–25145.

[29] Drew A Hudson and Christopher D Manning. 2019. Gqa: A new dataset for real-world visual reasoning and compositional question answering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 6700–6709.

[30] Justin Johnson, Bharath Hariharan, Laurens Van Der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. 2017. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2901–2910.

[31] Justin Johnson, Ranjay Krishna, Michael Stark, Li-Jia Li, David Shamma, Michael Bernstein, and Li Fei-Fei. 2015. Image Retrieval Using Scene Graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[32] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2021. Convergence of Datalog over (Pre-) Semirings.

[33] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[34] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. 2022. Solving Quantitative Reasoning Problems with Language Models. *arXiv:2206.14858* (2022).

[35] Qing Li, Siyuan Huang, Yining Hong, Yixin Chen, Ying Nian Wu, and Song-Chun Zhu. 2020. Closed loop neural-symbolic learning via integrating neural perception, grammar parsing, and symbolic reasoning. In *International Conference on Machine Learning (ICML)*. PMLR, 5884–5894.

[36] Yuhong Li, Tianle Cai, Yi Zhang, Deming Chen, and Debadeepta Dey. 2022. What Makes Convolutional Models Great on Long Sequence Modeling?

[37] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[38] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. 2018. Deepproblog: Neural probabilistic logic programming. *Advances in Neural Information Processing Systems* 31 (2018).

[39] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. 2019. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. *arXiv preprint arXiv:1904.12584* (2019).

[40] Pasquale Minervini, Sebastian Riedel, Pontus Stenetorp, Edward Grefenstette, and Tim Rocktäschel. 2020. Learning reasoning strategies in end-to-end differentiable proving. In *International Conference on Machine Learning (ICML)*. PMLR, 6938–6949.

[41] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems* 32 (2019).

[43] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[44] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).

[45] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-Scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 196–206.

[46] Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. 2019. Synthesizing Datalog Programs using Numerical Relaxation. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 6117–6124.

[47] Koustuv Sinha, Shagun Sodhani, Jin Dong, Joelle Pineau, and William L. Hamilton. 2019. CLUTRR: A Diagnostic Benchmark for Inductive Reasoning from Text. *CoRR* abs/1908.06177 (2019). arXiv:1908.06177

[48] Hao Tan and Mohit Bansal. 2019. LXMERT: Learning Cross-Modality Encoder Representations from Transformers. *CoRR* abs/1908.07490 (2019). arXiv:1908.07490

[49] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. 2020. Long Range Arena: A Benchmark for Efficient Transformers. *CoRR* abs/2011.04006 (2020). arXiv:2011.04006

[50] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming.

[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* 30 (2017).

[52] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[53] Po-Wei Wang, Priya L. Donti, Bryan Wilder, and Zico Kolter. 2019. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver.

[54] Saining Xie, Chen Sun, Jonathan Huang, Zhuowen Tu, and Kevin Murphy. 2018. Rethinking spatiotemporal feature learning: Speed-accuracy trade-offs in video classification. In *Proceedings of the European conference on computer vision (ECCV)*. 305–321.

[55] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. 2018. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. *Advances in Neural Information Processing Systems* 31 (2018).

$$
\begin{array}{rcl}
item & := & importDef \mid typeDef \mid constDef \mid relaDef \mid queryDef \\
importDef & := & \texttt{import}\ file \\
typeDef & := & \texttt{type}\ typeName = aliasName \mid \texttt{type}\ typeName <: superTypeName \\
 & \mid & \texttt{type}\ relationName([name:]type^*) \\
constDef & := & \texttt{const}\ constName[:type] = constant \\
relaDef & := & \texttt{rel}\ [tag::]\ relaName(expr^*) \mid \texttt{rel}\ relaName = \{taggedTuple^*\} \\
 & \mid & \texttt{rel}\ [tag::]\ atom\ \texttt{:-}\ formula \\
taggedTuple & := & [tag::]\ (constant^*) \\
atom & := & relaName(expr^*) \\
formula & := & formula\ \texttt{and}\ formula \mid formula\ \texttt{or}\ formula \mid formula\ \texttt{implies}\ formula \\
 & \mid & atom \mid \neg atom \mid constraint \mid reduce \\
reduce & := & var^* = aggregator(var^*: formula\ [\texttt{where}\ var^*: formula]) \\
 & \mid & var^* = sampler(var^*: formula\ [\texttt{where}\ var^*: formula]) \\
type & := & \texttt{u8} \mid \texttt{u16} \mid \texttt{u32} \mid \texttt{u64} \mid \texttt{usize} \mid \texttt{i8} \mid \texttt{i16} \mid \texttt{i32} \mid \texttt{i64} \mid \texttt{isize} \mid \texttt{f32} \mid \texttt{f64} \mid \texttt{bool} \mid \texttt{char} \mid \texttt{String} \\
expr & := & expr\ binOp\ expr \mid unaryOp\ expr \mid expr\ \texttt{as}\ type \mid \texttt{if}\ expr\ \texttt{then}\ expr\ \texttt{else}\ expr \\
 & \mid & \$func(expr^*) \mid variable \mid constant \\
binOp & := & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
unaryOp & := & \texttt{!} \mid \texttt{-} \\
aggregator & := & \texttt{min} \mid \texttt{argmin<}var^*\texttt{>} \mid \texttt{max} \mid \texttt{argmax<}var^*\texttt{>} \mid \texttt{count} \mid \texttt{sum} \mid \texttt{prod} \mid \texttt{exists} \mid \texttt{forall} \\
sampler & := & \texttt{top<K>} \mid \texttt{categorical<K>} \mid \texttt{uniform<K>}
\end{array}
$$

Fig. 20. Language grammar ($var : variable$ and $f : formula$).

$$
\begin{array}{rccl}
\text{(Tag)} & t & \in & T \\
\text{(False)} & \mathbf{0} & \in & T \\
\text{(True)} & \mathbf{1} & \in & T \\
\text{(Disjunction)} & \oplus & : & T \times T \to T \\
\text{(Conjunction)} & \otimes & : & T \times T \to T \\
\text{(Negation)} & \ominus & : & T \to T \\
\text{(Saturation)} & \Subset & : & T \times T \to \text{Bool} \\
\text{(Early Removal)} & \text{discard} & : & T \to \text{Bool} \\
\text{(Weight)} & \text{weight} & : & T \to \mathbb{R}
\end{array}
$$

Fig. 21. Algebraic interface for provenance.

A RIPPLE LANGUAGE SYNTAX

In this appendix, we provide the abstract syntax of the full language of Ripple in Fig. 20.

B REASONING FRAMEWORK

In this appendix, we provide the extensions to our provenance framework, and illustrate the additional features provided by full RplRam.

B.1 Provenance Extensions

We present the full provenance interface including two additional features, early removal and weight for sampling, in Fig. 21. For early removal, we introduce a function *discard* where it returns a boolean value based on a tag. When returned true, the fact associated with this tag will be early removed from the computation. For sampling, we allow each provenance to implement a function *weight* that returns a real number in $\mathbb{R}$ representing the weight of the tag. These features will be used later when we introduce the full RplRam syntax and semantics.

B.2 Abstract Syntax of RplRam

We revisit the abstract syntax of RplRam, and add extensions including:

$$
\begin{array}{rrcl}
\text{(Predicate)} & p & & \\
\text{(Aggregator)} & g & ::= & \texttt{count} \mid \texttt{sum} \mid \texttt{prod} \mid \texttt{exists} \mid \texttt{max} \mid \texttt{min} \mid \texttt{argmax} \mid \texttt{argmin} \\
\text{(Sampler)} & \mu & ::= & \texttt{top<K>} \mid \texttt{categorical<K>} \mid \texttt{uniform<K>} \\
\text{(Expression)} & e & ::= & \varnothing \mid p \mid \mathbb{1}(e) \mid \emptyset(e) \mid \gamma_g(e) \mid \hat{\gamma}_g(e_1, e_2) \mid \psi_\mu(e) \mid \hat{\psi}_\mu(e_1, e_2) \mid \pi_\alpha(e) \mid \sigma_\beta(e) \\
& & \mid & e_1 \cup e_2 \mid e_1 \times e_2 \mid e_1 \bowtie e_2 \mid e_1 \cap e_2 \mid e_1 - e_2 \\
\text{(Rule)} & r & ::= & p \leftarrow e \\
\text{(Stratum)} & s & ::= & \{r_1, \ldots, r_n\} \\
\text{(Program)} & \overline{s} & ::= & s_1; \ldots; s_n
\end{array}
$$

Fig. 22. Full abstract syntax of RplRam.

(1) Empty-set ($\varnothing$), One-overwrite ($\mathbb{1}(e)$) and zero-overwrite ($\emptyset(e)$),
(2) Intersection ($\cap$), Natural-Join ($\bowtie$), and Anti-join ($\triangleright$),
(3) Group-by aggregate ($\hat{\gamma}_g(e_1, e_2)$), where e_1 is the expression finding groups, and e_2 is the main expression to aggregate over, conditioned on the groups found by evaluating e_1,
(4) Sampling ($\psi_\mu(e)$) where μ denotes a sampler,
(5) Group-by sampling ($\hat{\psi}_\mu(e_1, e_2)$), which is sampling but similar to aggregate, we have e_1 being the expression finding groups.

Remark. The One-overwrite $\mathbb{1}(e)$ is particularly useful for defining magic-set transformation under tagged semantics. Overwriting **1** will make the tuples in magic-set predicate serve as pure demand fact for optimization, not conditioned on some given tags.

B.3 Operational Semantics of RplRam

We show the full semantics in Fig. 23, with the addition of group-by, sampling, and early-removal. Note that the sampler $\mu : \mathcal{U}_{\mathbb{R}} \to \mathcal{U}$ takes in facts tagged by weights, and samples according to the weight. The early removal happens in the stage of normalization before each rule update.

Expression semantics $\alpha : \mathbb{U} \to \mathbb{U}, \quad \beta : \mathbb{U} \to \text{Bool}, \quad g : \mathcal{U} \to \mathcal{U}, \quad \mu : \mathcal{U}_{\mathbb{R}} \to \mathcal{U}, \quad [\![e]\!] : \mathcal{F}_T \to \mathcal{U}_T$

$$\frac{}{[\![\varnothing]\!](F_T) = \emptyset}\ (\textsc{Empty Set}) \qquad \frac{t :: p(u) \in F_T}{t :: u \in [\![p]\!](F_T)}\ (\textsc{Predicate})$$

$$\frac{t :: u \in [\![e]\!](F_T)}{\mathbf{0} :: u \in [\![\emptyset(e)]\!](F_T)}\ (\mathbf{0}\text{-Overwrite}) \qquad \frac{t :: u \in [\![e]\!](F_T)}{\mathbf{1} :: u \in [\![\mathbb{1}(e)]\!](F_T)}\ (\mathbf{1}\text{-Overwrite})$$

$$\frac{t :: u \in [\![e]\!](F_T) \qquad \beta(u) = \text{true}}{t :: u \in [\![\sigma_\beta(e)]\!](F_T)}\ (\textsc{Select}) \qquad \frac{t :: u \in [\![e]\!](F_T) \qquad u' = \alpha(u)}{t :: u' \in [\![\pi_\alpha(e)]\!](F_T)}\ (\textsc{Project})$$

$$\frac{t :: u \in [\![e_1]\!](F_T) \cup [\![e_2]\!](F_T)}{t :: u \in [\![e_1 \cup e_2]\!](F_T)}\ (\textsc{Union}) \qquad \frac{t_1 :: u_1 \in [\![e_1]\!](F_T) \qquad t_2 :: u_2 \in [\![e_2]\!](F_T)}{(t_1 \otimes t_2) :: (u_1, u_2) \in [\![e_1 \times e_2]\!](F_T)}\ (\textsc{Product})$$

$$\frac{t_1 :: u \in [\![e_1]\!](F_T) \qquad t_2 :: u \in [\![e_2]\!](F_T)}{(t_1 \otimes t_2) :: u \in [\![e_1 \cap e_2]\!](F_T)}\ (\textsc{Intersect})$$

$$\frac{t_1 :: (u, u_1) \in [\![e_1]\!](F_T) \qquad t_2 :: (u, u_2) \in [\![e_2]\!](F_T)}{(t_1 \otimes t_2) :: (u, u_1, u_2) \in [\![e_1 \bowtie e_2]\!](F_T)}\ (\textsc{Natural-Join})$$

$$\frac{t :: u \in [\![e_1]\!](F_T) \qquad [\![e_2]\!](F_T) \not\vdash u}{t :: u \in [\![e_1 - e_2]\!](F_T)}\ (\textsc{Diff-1}) \qquad \frac{t_1 :: u \in [\![e_1]\!](F_T) \qquad t_2 :: u \in [\![e_2]\!](F_T)}{(t_1 \otimes (\ominus\, t_2)) :: u \in [\![e_1 - e_2]\!](F_T)}\ (\textsc{Diff-2})$$

$$\frac{t :: (u_1, u_2) \in [\![e_1]\!](F_T) \qquad [\![e_2]\!](F_T) \not\vdash u_1}{t :: (u_1, u_2) \in [\![e_1 \triangleright e_2]\!](F_T)}\ (\textsc{AntiJoin-1}) \qquad \frac{t_1 :: (u_1, u_2) \in [\![e_1]\!](F_T) \qquad t_2 :: u_1 \in [\![e_2]\!](F_T)}{(t_1 \otimes (\ominus\, t_2)) :: (u_1, u_2) \in [\![e_1 \triangleright e_2]\!](F_T)}\ (\textsc{AntiJoin-2})$$

$$\frac{X_T \subseteq [\![e]\!](F_T) \qquad \{t_i :: u_i\}_{i=1}^n = X_T \qquad \{\bar{t}_j :: \bar{u}_j\}_{j=1}^m = [\![e]\!](F_T) - X_T \qquad u \in g(\{u_i\}_{i=1}^n)}{(\bigotimes_{i=1}^n t_i) \otimes (\bigotimes_{j=1}^m (\ominus\, \bar{t}_j)) :: u \in [\![\gamma_g(e)]\!](F_T)}\ (\textsc{Aggregate})$$

$$\frac{\begin{array}{c} t' :: (u_g, u'_g) \in [\![e_1]\!](F_T) \qquad X_T \subseteq [\![e_2]\!](F_T) \\ \{t_i :: (u_g, u_i)\}_{i=1}^n = X_T \qquad \{\bar{t}_j :: (u_g, \bar{u}_j)\}_{j=1}^m = [\![e_2]\!](F_T) - X_T \qquad u \in g(\{u_i\}_{i=1}^n) \end{array}}{t' \otimes (\bigotimes_{i=1}^n t_i) \otimes (\bigotimes_{j=1}^m (\ominus\, \bar{t}_j)) :: (u_g, u'_g, u) \in [\![\hat{\gamma}_g(e_1, e_2)]\!](F_T)}\ (\textsc{Group-By Aggregate})$$

$$\frac{\{t_i :: u_i\}_{i=1}^n = [\![e]\!](F_T) \qquad u_j \in \mu(\{\text{weight}(t_i) :: u_i\}_{i=1}^n)}{t_j :: u_j \in [\![\psi_\mu(e)]\!](F_T)}\ (\textsc{Sample})$$

$$\frac{t' :: (u_g, u'_g) \in [\![e_1]\!](F_T) \qquad \{t_i :: (u_g, u_i)\}_{i=1}^n \subseteq [\![e]\!](F_T) \qquad u_j \in \mu(\{\text{weight}(t_i) :: u_i\}_{i=1}^n)}{(t' \otimes t_j) :: (u_g, u'_g, u_j) \in [\![\hat{\psi}_\mu(e_1, e_2)]\!](F_T)}\ (\textsc{Group-By Sample})$$

Rule semantics $\langle . \rangle : \mathcal{U}_T \to \mathcal{U}_T, \quad [\![r]\!] : \mathcal{F}_T \to \mathcal{F}_T$

$$\frac{t_1 :: u, \ldots, t_n :: u \text{ are all tagged-tuples in } U_T \qquad t = \bigoplus_{i=1}^n t_i \qquad \text{discard}(t) = \text{false}}{t :: u \in \langle U_T \rangle}\ (\textsc{Normalize})$$

$$\frac{t^{\text{old}} :: u \in [\![p]\!](F_T) \qquad \langle [\![e]\!](F_T) \rangle \not\vdash u}{t^{\text{old}} :: p(u) \in [\![p \leftarrow e]\!](F_T)}\ (\textsc{Rule-1}) \qquad \frac{t^{\text{new}} :: u \in \langle [\![e]\!](F_T) \rangle \qquad [\![p]\!](F_T) \not\vdash u}{t^{\text{new}} :: p(u) \in [\![p \leftarrow e]\!](F_T)}\ (\textsc{Rule-2})$$

$$\frac{t^{\text{old}} :: u \in [\![p]\!](F_T) \qquad t^{\text{new}} :: u \in \langle [\![e]\!](F_T) \rangle}{(t^{\text{old}} \oplus t^{\text{new}}) :: p(u) \in [\![p \leftarrow e]\!](F_T)}\ (\textsc{Rule-3})$$

Stratum and Program semantics $\text{lfp}^\circ : (\mathcal{F}_T \to \mathcal{F}_T) \to (\mathcal{F}_T \to \mathcal{F}_T), \quad [\![s]\!], [\![\bar{s}]\!] : \mathcal{F}_T \to \mathcal{F}_T$

(Remained same)

Fig. 23. Operational semantics of RplRam.

B.4 Provenance Structures

We provide detailed runtime analysis for each presented provenance structure in the following sections. Additionally, we discuss the extended provenance `diff-top-k-proofs-me` which can handle mutual exclusive facts.

B.4.1 `diff-max-min-prob`.

Operation	Algorithm	Time Complexity
$t_1 \oplus t_2$	$\max(t_1, t_2)$	$O(1)$
$t_1 \otimes t_2$	$\min(t_1, t_2)$	$O(1)$
$\ominus t$	$\hat{1} - t$	$O(1)$
$t_1 \oslash t_2$	$t_1 == t_2$	$O(1)$
$\rho(t)$	t	$O(1)$

Table 6. `diff-max-min-prob` provenance structure run time analysis

Operation	Algorithm	Time Complexity
$t_1 \oplus t_2$	$clamp(t_1 + t_2)$	$O(n)$
$t_1 \otimes t_2$	$t_1 \cdot t_2$	$O(n)$
$\ominus t$	$\hat{1} - t$	$O(n)$
$t_1 \oslash t_2$	true	$O(1)$
$\rho(t)$	t	$O(1)$

Table 7. `diff-add-mult-prob` provenance structure run time analysis

Operation	Algorithm	Time Complexity
$t_1 \oplus t_2$	$\text{top}_k(t_1 \cup t_2)$	$O(nk)$
$t_1 \otimes t_2$	$\text{top}_k(\{\ \eta \mid (\eta_1, \eta_2) \in t_1 \times t_2, \eta = \eta_1 \cup \eta_2, \eta \text{ no conflict}\ \})$	$O(n^2k^2)$
$\ominus t$	$\text{top}_k(\text{cnf2dnf}(\{\{\neg v \mid v \in \eta\} \mid \eta \in t\}$	$O(2^n)$
$t_1 \oslash t_2$	$t_1 == t_2$	$O(nk)$
$\rho(t)$	$\text{WMC}(t, \Gamma)$	$O(2^n)$

Table 8. `diff-top-k-proofs` provenance structure run time analysis. We assume the term number for each tag is of $O(n)$, and the number of clauses is of $O(k)$. Note that we show the time complexity for the naive implementation of cnf2dnf and WMC algorithms.

Runtime Analysis. We showcase the time complexity for each operator in the max-min-prob provenance structure in Table 6. We next present our optimized counting algorithm in Fig. 1 which performs counting over a set of `max-min-prob` tagged-tuples. Note that we only showed the algorithm for `mmp` for simplicity. But it easily extends to `dmmp`. It can be easily shown that its runtime complexity is $O(n \log(n))$.

Algorithm 1: Counting over max-min-prob tagged tuples

Data: $U_{\text{mmp}} = \{t_1 :: u_1, t_2 :: u_2, \ldots, t_n :: u_n\}$: $\mathcal{U}_{\text{mmp}}$, set of tagged-tuples
Result: U'_{mmp}: $\mathcal{U}_{\text{mmp}}$
/* sort all positive tuples according to their tags from small to large. O(nlog(n)) */
$\mathbf{t}^{\text{pos}} = sorted([t_i \mid i = 1 \ldots n])$;
$\mathbf{t}^{\text{neg}} = [1 - t^{\text{pos}}_{n-i+1} \mid i = 1 \ldots n]$;
/* Iterate through all possible partitions between positive and negative tags. O(n) */
$U'_{\text{mmp}} = \{t^{\text{neg}}_n :: 0, t^{\text{pos}}_1 :: n\}$;
for $i = 1 \ldots (n-1)$ **do**
  Add $\min(t^{\text{pos}}_{i+1}, t^{\text{neg}}_i) :: (n-i)$ to U'_{mmp};
return U'_{mmp}

B.4.2 `diff-add-mult-prob`.

Runtime Analysis. We showcase the time complexity for the diff-add-mult-prob provenance structure in Table 7. We denote the number of total variables in the input as n. Since we are performing calculations over the dual-numbers, the time complexity is decided by the gradient calculation over vectors of dimension n.

B.4.3 `diff-top-k-proofs`.

Runtime Analysis. We showcase the time complexity for the diff-top-k-proofs provenance structure in Table 8. Note that we show the runtime analysis for the naive implementation of the cnf2dnf function and weighted model counting process. We leave the optimizations for future work.

B.4.4 `diff-top-k-proofs-me`.

Support for Mutually Exclusive Facts. We keep the original `diff-top-k-proofs` and extend it to support mutal exclusive tags (hence the name `-me`). Most of the operations remains, and we start by giving it a new input tag space:

$$I_{\texttt{dtkp-me}} = [0, 1] \times \texttt{option<}\mathbb{N}\texttt{>}.$$

The probability remains the same, and the second part is an optional identifier of one mutual exclusion. Not providing that means the fact does not belong to a mutual exclusion. If presented, we denote such a tag

$$t_i = (r_i, \texttt{some}(\mathbf{me}_i))$$

Example B.1. Consider the sum2 task, we have two distributions specified by

```
rel digit = {0.90::(A, 0); 0.01::(A, 1); ...; 0.02::(A, 9)}
rel digit = {0.01::(B, 0); 0.91::(B, 1); ...; 0.01::(B, 9)}
```

Then we would have the input facts being

$$[\![\texttt{digit}]\!](F_{\texttt{option<}I\texttt{>}}) = \left\{ \begin{array}{lll} (0.90, \texttt{some}(1)) & :: & (A, 0), \\ (0.01, \texttt{some}(1)) & :: & (A, 1), \\ & \vdots & \\ (0.02, \texttt{some}(1)) & :: & (A, 9), \\ \hline (0.01, \texttt{some}(2)) & :: & (B, 0), \\ (0.91, \texttt{some}(2)) & :: & (B, 1), \\ & \vdots & \\ (0.01, \texttt{some}(2)) & :: & (B, 9) \end{array} \right\}.$$

There are two **me**-s, namely 1 and 2, suggesting there are two mutual exclusions. The first 10 facts for digit A belong to mutual exclusion 1, and the next 10 for digit B belong to mutual exclusion 2.

Putting it in action, two facts f_i and f_j with the same **me** ($\mathbf{me}_i = \mathbf{me}_j$) cannot appear together in the same proof. This is enforced inside of the conflict check during `dtkp`'s $\wedge_k$ and $\neg_k$ operations. If the proof violates the mutual exclusive assumption, it will be deemed invalid proof and discarded. For example, executing the following rule

```
rel not_possible() = digit(A, x), digit(A, y), x != y
```

shall yield no valid output due to this mechanism, i.e., the internal tag associated with `not_possible()` is going to be $\emptyset$ and be early-removed from the computation.

```
type digit_1(u32), digit_2(u32)
rel sum_2(a + b) = digit_1(a), digit_2(b)
```

(a) Ripple code for MNIST sum-2.

```
type digit_1(u32), digit_2(u32), digit_3(u32)
rel sum_3(a + b + c) = digit_1(a), digit_2(b), digit_3(c)
```

(b) Ripple code for MNIST sum-3.

```
type digit_1(u32), digit_2(u32), digit_3(u32), digit_4(u32)
rel sum_4(a + b + c + d) = digit_1(a), digit_2(b), digit_3(c), digit_4(d)
```

(c) Ripple code for MNIST sum-4.

```
type digit_1(u32), digit_2(u32)
rel less_than(a < b) = digit_1(a), digit_2(b)
```

(d) Ripple code for MNIST less-than.

```
type digit(u32)
rel not_3_or_4() = not digit(3) and not digit(4)
```

(e) Ripple code for MNIST not-3-or-4.

```
type digit(digit_id: u32, digit_value: u32)
rel count_3(x) :- x = count(o: digit(o, 3))
```

(f) Ripple code for MNIST count-3.

```
type digit(digit_id: u32, digit_value: u32)
rel count_3_or_4(x) = x = count(o: digit(o, 3) or digit(o, 4))
```

(g) Ripple code for MNIST count-3-or-4.

Fig. 24. Ripple code for MNIST-R.

C EVALUATION

C.1 MNIST-R

Experimental Setup. The total MNIST[33] dataset contains 60K handwritten digits. According to the number of digits used in each task, we further split the MNIST dataset into downstream task data points. For example, we have 30K training pairs for sum-2, 20K training pairs for sum-3, and 15K training pairs for sum-4. As the counting task consumes 8 images per datapoint, we have 7.5K training pairs for the counting tasks. The test and training splits follow the original MNIST task setup. Our neural module consists of a 2-layer CNN connected by a 2-layer MLP, with a hidden layer size of 1024. We trained the Ripple module 10 epochs on the training dataset, the learning rate is set to 0.001, and the batch size is 64, with a BCE-loss closing the training loop.

Ripple code. We present the Ripple code that is used in the MNIST-R benchmark in Fig. 24.

C.2 Hand Written Formula

Experimental Setup. The HWF [35] dataset takes in a sequence of images, where each image can be either a digit or an arithmetic symbol, and aims to calculate the outcome of the formula. We apply the same CNN-based symbol recognition network as for the MNIST-based tasks, with

```
// Input from neural networks
type symbol(usize, String)
type length(usize)

// Facts for lexing
rel digit = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"}
rel mult_div = {"*", "/"}
rel plus_minus = {"+", "-"}

// Parsing
type value_node(id: u64, string: String, begin: usize, end: usize)
rel value_node($hash(x, d), d, x, x + 1) = symbol(x, d), digit(d)
rel value_node($hash(joint, b - 1, e), joint, b - 1, e) =
  symbol(b - 1, dh), digit(dh), value_node(x, dr, b, e), joint
  == $string_concat(dh, dr)

type mult_div_node(id: u64, string: String, left_node: u64,
right_node: u64, begin: usize, end: usize)
rel mult_div_node(id, string, 0, 0, b, e) = value_node(id, string, b, e)
rel mult_div_node($hash(id, s, l, r), s, l, r, b, e) =
  symbol(id, s), mult_div(s), mult_div_node(l, _, _, _, b, id),
  value_node(r, _, id + 1, e)

type plus_minus_node(id: u64, string: String, left_node: u64, right_node: u64,
begin: usize, end: usize)
rel plus_minus_node(id, string, l, r, b, e) =
                      mult_div_node(id, string, l, r, b, e)
rel plus_minus_node($hash(id, s, l, r), s, l, r, b, e) =
                      symbol(id, s), plus_minus(s),
                      plus_minus_node(l, _, _, _, b, id),
                      mult_div_node(r, _, _, _, id + 1, e)

type root_node(id: u64)
rel root_node(id) = plus_minus_node(id, _, _, _, 0, l), length(l)

// Evaluate AST
@demand("bf")
rel eval(x, s as f64) = value_node(x, s, _, _)
rel eval(x, y1 + y2) = plus_minus_node(x, "+", l, r, _, _),
                       eval(l, y1), eval(r, y2)
rel eval(x, y1 - y2) = plus_minus_node(x, "-", l, r, _, _),
                       eval(l, y1), eval(r, y2)
rel eval(x, y1 * y2) = mult_div_node(x, "*", l, r, _, _),
                       eval(l, y1), eval(r, y2)
rel eval(x, y1 / y2) = mult_div_node(x, "/", l, r, _, _),
                       eval(l, y1), eval(r, y2), y2 != 0.0

// Compute result
rel result(y) = eval(e, y), root_node(e)
```

Fig. 25. Ripple code for HWF.

```
// Input from neural networks
type dash(i8, i8)
type dot(i8)

// Connectivity check
rel path(x, y) = dash(x, y) or path(x, z) and dash(z, y)
rel connected() = dot(x), dot(y), path(x, y), x != y
```

Fig. 27. Ripple code for Pathfinder.

the only difference being that the model now performs a 14-way classification. We then generate a probabilistic database containing two relations: 1) `length(usize)`, a non-probabilistic relation storing the length of the formula, and 2) `symbol(id: usize, σ: String)`, a probabilistic relation mapping index IDs to symbols $\sigma \in \Sigma$. For the reasoning component, we write a formula parser in Ripple (as shown in 25). Additionally, Ripple is used to perform calculations based on the parsed abstract syntax tree (AST) to obtain the final rational number output. The training loop is closed by a BCE loss at the end, forming an end-to-end training pipeline where gradients can back-propagate all the way from the output to the symbol recognition network. To prune the enormous output domain if we enumerate every possible parse tree, we sample only a subset of symbol predictions to be sent to Ripple. There are 10K images answer pairs for the training dataset. We set the batch size to 16, the learning rate to 0.0001, and the training epochs to 100. To enhance the learning efficiency, we adopt a sampling strategy, which only preserves the 7 most likely classification result for each image classification task, and sends it to Ripple.

C.3 Pathfinder

Problem Definition. Pathfinder is a task that initially set out to test the long-range reasoning ability of neural networks. It is adopted into a benchmark called Long Range Arena (LRA) [49] to evaluate the long-range reasoning ability of neural networks, or more specifically the Transformers [51]. In this task, we are provided with images with dark backgrounds and two white dots possibly connected by dashes. The model is then required to make a binary prediction on whether the two dots are connected or not. There are multiple versions of the dataset: one consisting of only 32×32 images (Path) and the other one with 128×128 (Path-X). Both versions have 3 difficulties, easy, normal, and hard, depending on the length of the dashes connecting the two dots – longer means harder. Table 26 shows the 3 difficulties each with one positive and one negative sample from Path-X.

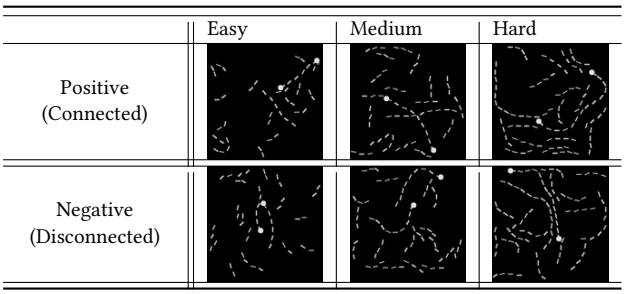


Fig. 26. Examples of positive and negative for each difficulty level.

Architecture. Going to the extreme of symbolism one would use the individual symbol to represent each small dash and dot. However, doing so would mean that the perception network, presumably a CNN, needs to segment the image precisely to extract all dashes and dots as entities. At the same time, the reasoning module bears an extra burden when processing individual connectivities.

```
// Input from neural networks
type grid_node(x: usize, y: usize) // 0.99 for each grid cell
type actor(x: usize, y: usize)
type goal(x: usize, y: usize)
type enemy(x: usize, y: usize)

// Basic connectivity
const UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3
rel safe_node(x, y) = grid_node(x, y), not enemy(x, y)
rel edge(x, y, x, yp, UP) = safe_node(x, y), safe_node(x, yp), yp == y + 1
rel edge(x, y, xp, y, RIGHT) = safe_node(x, y), safe_node(xp, y), xp == x + 1
rel edge(x, y, x, yp, DOWN) = safe_node(x, y), safe_node(x, yp), yp == y - 1
rel edge(x, y, xp, y, LEFT) = safe_node(x, y), safe_node(xp, y), xp == x - 1

// Get the next position
rel next_pos(xp, yp, a) = actor(x, y), edge(x, y, xp, yp, a)

// Path for connectivity; will condition on no enemy on the path
rel path(x, y, x, y) = next_pos(x, y, _)
rel path(x1, y1, x3, y3) = path(x1, y1, x2, y2), edge(x2, y2, x3, y3, _)

// Get the next action
rel next_action(a) = next_pos(x, y, a), goal(gx, gy), path(x, y, gx, gy)

// Constraint violation
type too_many_goal(), too_many_actor(), too_many_enemy()
rel too_many_goal() = n := count(x, y: goal(x, y)), n > 1
rel too_many_actor() = n := count(x, y: actor(x, y)), n > 1
rel too_many_enemy() = n := count(x, y: enemy(x, y)), n > 5
rel violation() = too_many_goal() or too_many_enemy() or too_many_actor()
```

Fig. 28. Ripple code for PacMan-Maze.

Instead, we employ a simple grid-based connectivity graph: each node represents a conceptual "dot" and each edge represents a conceptual "dash". We ask neural networks to take in the image and output a feature vector with the size of the number of dots plus the number of edges. Each logit in this vector is then treated as a probability of that "dot" or "dash". The Ripple program we use for this task, as shown in Fig. 27, is simply a transitive closure that checks whether there are two distinct dots connected by dashes.

Experimental Setup. There are 600K training data points for both PATH and PATH-X, which are images of 32×32 and 128×128 pixels respectively, and the y-label represents whether there exists a path that connects the two dots on the image. We recognize the "dash" and "dots" relations through a 4-layer CNN, and the local connectivity is recognized through a 2-layer MLP classifier. We set the learning rate to 0.0001, the batch size to 64, and the number of epochs to 100.

C.4 PacMan-Maze

PacMan-Maze is a reinforcement learning based planning application, which aims to lead the actor to the flag without running into any enemy. We show the Ripple code for this task in Fig. 28.

```
// Define kinships belong to the Relation type
type Relation = usize

//  Input from neural networks
type question(sub: String, obj: String)
type kinship(rela: Relation, sub: String, obj: String)
type transitive(Relation, Relation, Relation)

// Rules to derive the final answer
rel kinship(r3, x, z) = transitive(r1, r2, r3), kinship(r1, x, y),
                        kinship(r2, y, z), x != z
rel answer(r) = question(s, o), kinship(r, s, o)
```

Fig. 29. Ripple code for CLUTRR.

In addition to the code shown in Section 2, we added additional "integrity constraint violation" terms that need to be minimized during learning.

Experimental Setup. We set the size of the maze to be a 5×5 matrix, and the maximum number of enemies that exists in the arena to be 5. In the training process, we set the batch size to be 24, the memory replay buffer size to be 3000, and the maximum number of actions in an episode to be 30. We update the target net every 10 epochs, we set the exploration rate to be 0.9, its falloff to be 0.98, and the learning rate is set to 0.0001.

C.5 CLUTRR

With CLUTRR [47] we step into the domain of NLP on kinship reasoning. Each data point in the CLUTRR dataset consists of a natural language passage, a query, and its answer. The passage tells a story about a family's daily activities; the query is a tuple of two names, denoting the target pair of persons that we want to predict the relation for; the answer is among the 20 basic relations such as "mother", "uncle", and "daughter-in-law". The dataset is further categorized into 10 difficulty levels according to the number of facts (k) required to derive the final result. Note that the knowledge base for kinship transitivity, such as "father's father is grandfather" is not included in the dataset. We, therefore, designed three experiments to help alleviate the issue of missing knowledge base, a) learning with manually specified rules, b) rule learning, and c) learning without rules. We showcase our Ripple code in Fig. 29.

Learn with Manually Specified Rules. A straightforward strategy to overcome the missing required database issue, is to manually add it. We created an external knowledge base with 92 kinship transitivity triplets and 3 rules. To define how two known relations can be connected to derive the third relation, we design a high-order predicate, `transitive`. As an example, the rule "father's mother is grandmother" is encoded as `transitive(FATHER, MOTHER, GRANDMOTHER)`. The full Ripple program connecting the knowledge base into the reasoning process is shown in Fig. 29 Given the knowledge base, context, and question-answer pairs, we are ready to perform learning over the new dataset. As the context has varies lengths, we first separate the context into multiple windows to ensure the perception model processes similar-length-context across the dataset. Then a large language model, such as RoBERTa [37], is adopted to extract relations from these windowed texts. The kinship extracted from the context goes into `kinship(rela, sub, obj)`. For example, `kinship(SON, "Alice", "Bob")` indicates that Bob is Alice's son. The query, on the other hand, is stored in the relation `question(sub, obj)`. Next, we will combine the predicted relations from all the context windows

into one probabilistic database, together with the knowledge base, and the query to perform logical reasoning. Last, we compare the probabilistic query result with the ground truth with BCE loss.

Rule Learning. Specifying 92 transitive rules manually could be time consuming. With Ripple, it is possible to do this in a smarter way. Since there are 20 basic kinship relations, we have a finite space containing $20^3 = 8$K possible transitive facts. We treat all 8K transitivity facts as probabilistic and to be learnt, and therefore we store them inside a tensor of size $20 \times 20 \times 20$. We can initialize this tensor randomly but with a low maximum weight (i.e. 0.1). During training, we will allow gradients to be back-propagated to this transitivity tensor so that these weights can also be learnt. In this case, the user does not specify any transitivity fact manually and everything is learnt on the fly. Note that, 8K transitive facts can slow down the training time. Therefore, we use multinomial sampling to pick 150 transitive facts for training. During testing, we simply select the top 150 for inference.

Learn without Rules. We can also jointly extract entity relations and learn the transitive rules without the human specified rules. This is a more challenging task where one needs to figure out the way to combine inferred relations and derive the desired answer without intermediate supervision.

Experimental Setup. We have 10K training data points, which are triplets of passage, query, and answers. We use RoBERTa as the backbone language model. We further train a 2-layer MLP-based relation extractor which takes in the embedding of three components, a global embedding of the passage, and the embedding of the queried subject and the object, and returns the result for a 21-way classification. We set the batch size to 32, the learning rate to 0.00001, and train the models for 100 epochs.

C.6 Mugen

Experimental Setup. We sample 1K Mugen[27] video and text pairs as the training dataset, and another 1K for testing. Our neural module consists of the S3D image embedding and the distillBert text embedding. Then we pass the embeddings through a 2-layer MLP, with a hidden layer size of 256. We trained the Ripple module 1000 epochs on the training dataset, the learning rate is set to 0.0001, and the batch size is 3, with BCE-loss closing the training loop. The Ripple code is shown in Fig. 30

C.7 CLEVR

CLEVR [30] stands for **C**ompositional **L**anguage and **E**lementary **V**isual **R**easoning, is a synthetic dataset testing model's ability to perform *Visual Question Answering* (VQA). Each data point of the dataset contains an image and a question-answer pair with regard to the image. The images are randomly generated from a dictionary of 3 shapes, 8 colors, 2 materials, and 2 sizes and the programmatic queries are also randomly generated. Given a rendered image with simple objects such as cubes and spheres, we aim to answer a question about the image.

Architecture. The object feature vectors are obtained by a 4-layer convolutional neural network. The feature vectors are then passed to four 2-layer MLP classifiers and another 3-layer MLP classifier to obtain the attributes and the spacial relations respectively. The learning rate is set to 0.0001, the batch size is 32, and the latent dimension is 1024. We show our Ripple code in Fig. 31.

```
// Input from neural networks
type action(usize, String)
type expr(usize, String)
type expr_start(usize)
type expr_end(usize)
type action_start(usize)
type action_end(usize)

type match_single(usize, usize, usize)
type match_sub(usize, usize, usize, usize)

// Check whether does a subsection of text specification matches the video content
rel match_single(tid, vid, vid + 1) = expr(tid, a), action(vid, a)
rel match_sub(tid, tid, vid_start, vid_end) =
    match_single(tid, vid_start, vid_end)
rel match_sub(tid, tid, vid_start, vid_end) =
    match_sub(tid, tid, vid_start, vid_mid),
    match_single(tid, vid_mid, vid_end)
rel match_sub(tid_start, tid_end, vid_start, vid_end) =
    match_sub(tid_start, tid_end - 1, vid_start, vid_mid),
    match_single(tid_end, vid_mid, vid_end)

// Check whether does the whole text specification matches the video content
rel match() = expr_start(tid_start),
    expr_end(tid_end), action_start(vid_start), action_end(vid_end),
    match_sub(tid_start, tid_end, vid_start, vid_end)

// Integrity violation when too many identical text expressions
// occurs consecutively
rel too_many_consecutive_expr() = expr(tid, a),
    expr(tid + 1, a), expr(tid + 2, a), expr(tid + 3, a)
```

Fig. 30. Ripple code for Mugen.

```
// yes/no question
rel eval_yn(e, x > y) = greater_than_expr(e, a, b),
                        eval_num(a, x), eval_num(b, y)
rel eval_yn(e, x < y) = less_than_expr(e, a, b),
                        eval_num(a, x), eval_num(b, y)
rel eval_yn(e, x == y) = equal_expr(e, a, b),
                         eval_num(a, x), eval_num(b, y)
rel eval_yn(e, x == y) = equal_color_expr(e, a, b),
                         eval_query(a, x), eval_query(b, y)
rel eval_yn(e, x == y) = equal_material_expr(e, a, b),
                         eval_query(a, x), eval_query(b, y)
rel eval_yn(e, x == y) = equal_shape_expr(e, a, b),
                         eval_query(a, x), eval_query(b, y)
rel eval_yn(e, x == y) = equal_size_expr(e, a, b),
                         eval_query(a, x), eval_query(b, y)
rel eval_yn(e, b) = b := exists(o: eval_objs(f, o)
                         where e: exists_expr(e, f))

// count number of objects
rel eval_num(e, n) = n := count(o: eval_objs(f, o) where e: count_expr(e, f))

// objects filter
rel eval_objs(e, o) = scene_expr(e), obj(o)
rel eval_objs(e, o) = unique_expr(e, f), eval_objs(f, o)
rel eval_objs(e, o) = filter_size_expr(e, f, s), eval_objs(f, o), size(o, s)
rel eval_objs(e, o) = filter_color_expr(e, f, c), eval_objs(f, o), color(o, c)
rel eval_objs(e, o) = filter_material_expr(e, f, m), eval_objs(f, o), material(o, m)
rel eval_objs(e, o) = filter_shape_expr(e, f, s), eval_objs(f, o), shape(o, s)
rel eval_objs(e, o) = and_expr(e, f1, f2), eval_objs(f1, o), eval_objs(f2, o)
rel eval_objs(e, o) = or_expr(e, f1, f2) and (eval_objs(f1, o) or eval_objs(f2, o))

// same attribute
rel eval_objs(e, o) = same_size_expr(e, f),
            eval_objs(f, p), size(p, c), size(o, c), o != p
rel eval_objs(e, o) = same_color_expr(e, f),
            eval_objs(f, p), color(p, c), color(o, c), o != p
rel eval_objs(e, o) = same_material_expr(e, f),
            eval_objs(f, p), material(p, m), material(o, m), o != p
rel eval_objs(e, o) = same_shape_expr(e, f),
            eval_objs(f, p), shape(p, s), shape(o, s), o != p

// relation
rel relate("right", p, o) = relate("left", o, p)
rel relate("front", p, o) = relate("behind", o, p)
rel eval_objs(e, o) = relate_expr(e, f, r),
                      eval_objs(f, p), relate(r, p, o), o != p

// eval query
rel eval_query(e, s) = query_size_expr(e, f), eval_objs(f, o), size(o, s)
rel eval_query(e, c) = query_color_expr(e, f), eval_objs(f, o), color(o, c)
rel eval_query(e, m) = query_material_expr(e, f), eval_objs(f, o), material(o, m)
rel eval_query(e, s) = query_shape_expr(e, f), eval_objs(f, o), shape(o, s)
```

```
54  // Final result
55  rel result(y as String) = root_expr(e), eval_yn(e, y)
56  rel result(y as String) = root_expr(e), eval_num(e, y)
57  rel result(y) = root_expr(e), eval_query(e, y)
```

Fig. 31. Ripple code for CLEVR.