DOLPHIN: A PROGRAMMABLE FRAMEWORK FOR SCALABLE NEUROSYMBOLIC LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Neurosymbolic learning has emerged as a promising paradigm to incorporate symbolic reasoning into deep learning models. However, existing frameworks are limited in scalability with respect to both the training data and the complexity of symbolic programs. We propose DOLPHIN, a framework to scale neurosymbolic learning at a fundamental level by mapping both forward chaining and backward gradient propagation in symbolic programs to vectorized computations. For this purpose, DOLPHIN introduces a set of abstractions and primitives built directly on top of a high-performance deep learning framework like PyTorch, effectively enabling symbolic programs to be written as PyTorch modules. It thereby enables neurosymbolic programs to be written in a language like Python that is familiar to developers and compile them to computation graphs that are amenable to end-to-end differentiation on GPUs. We evaluate DOLPHIN on a suite of 13 benchmarks across 5 neurosymbolic tasks that combine deep learning models for text, image, or video processing with symbolic programs that involve multi-hop reasoning, recursion, and even black-box functions like Python `eval()`. DOLPHIN only takes 0.33%-37.17% of the time (and 2.77% on average) to train these models on the largest input per task compared to baselines Scallop, ISED, and IndeCateR+, which time out on most of these inputs. Models written in DOLPHIN also achieve state-of-the-art accuracies even on the largest benchmarks.

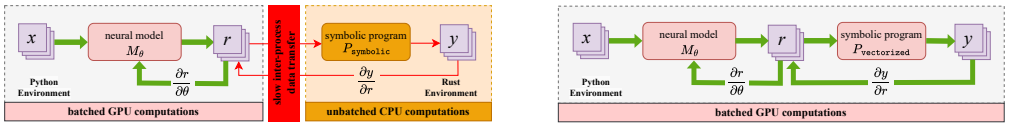
1 INTRODUCTION

Deep learning has achieved remarkable success on a wide range of tasks such as image classification, speech recognition, and natural language processing. With the recent emergence of foundation models like GPT-4 and CLIP, deep learning models are now increasingly being used in more challenging tasks. While such models work well for prediction and generation tasks, they are limited in their ability to perform reasoning required for tasks involving structure, logic, and planning, where symbolic reasoning traditionally excels (Kambhampati et al., 2024). Neurosymbolic programming (Chaudhuri et al., 2021) has emerged as a promising paradigm to incorporate symbolic reasoning into deep learning models, providing the best of both worlds.

Various frameworks have been developed to improve the programmability and accessibility of neurosymbolic applications (Manhaeve et al., 2018; Li et al., 2023; Solko-Breslin et al., 2024). These frameworks support complex symbolic reasoning features like recursion and black-box functions, implement efficient differentiable reasoning algorithms, and provide bindings for deep learning frameworks like PyTorch. Despite significant strides, however, these frameworks incur significant overhead when training neurosymbolic programs.

Consider a typical workflow of such a framework depicted in Figure 1(a). In this case, we have a supervised learning task with labeled data (x, y), a neural network M_θ that processes input x, and a symbolic program $P_{\texttt{symbolic}}$ that takes the network's output r and produces final output y. Existing frameworks, such as Scallop (Li et al., 2023), execute the neural model on GPU but use a separate CPU-based backend (implemented in Rust in Scallop's case) for the symbolic program. Moreover, such systems suffer from slow inter-process data transfers between the two environments.

Together, these issues hinder the scalability of neurosymbolic learning with respect to *problem complexity* and *data complexity*. First, the symbolic computation engine must derive a set of all possible results and their associated probabilities in a manner that is differentiable with respect to the net-



(a) Typical neurosymbolic framework (e.g. Scallop). (b) Our neurosymbolic framework DOLPHIN.

Figure 1: Comparison of system architectures of neurosymbolic frameworks.

work's parameters θ. As the complexity of the symbolic program increases, the number of possible results and their associated weights also grows exponentially, leading to a combinatorial explosion in the number of required computations. However, the symbolic computations are discrete and not easily parallelizable on modern hardware like GPUs. Second, larger datasets also compound the computational cost of neurosymbolic learning. Deep learning typically addresses this challenge by batching computations across multiple data samples. However, in neurosymbolic learning, the computations may differ across data samples, making it difficult to batch them effectively.

To address these challenges, we need to fundamentally rethink the design of a neurosymbolic framework. One approach is to develop specialized and low-level primitives that scale specific benchmarks but make it time-intensive for developers to write neurosymbolic programs tailored to particular tasks. Alternatively, providing high-level primitives—such as a logic programming language like Scallop (Li et al., 2023) or DeepProbLog (Manhaeve et al., 2018)—simplifies the development of symbolic programs but limits the fine-grained control needed to scale specific applications. Finally, to truly democratize neurosymbolic programming, it is crucial to develop a framework that seamlessly integrates into the everyday deep learning workflows that developers already use.

In this work, we propose DOLPHIN, a novel framework for scalable neurosymbolic learning. In DOLPHIN, we build three key components that effectively tackle the scalability and programmability challenges described above. First, we develop a general symbolic representation that efficiently captures the relationships between neural network outputs and associated discrete symbols. Second, we introduce a set of primitives to map forward chaining in symbolic programs to vectorized computations over these representations. Third, we develop a set of vectorized *provenance semirings* (Green et al., 2007) that are easily pluggable into DOLPHIN and enable to efficiently compute symbolic gradients. As illustrated in Figure 1b, these components together allow DOLPHIN to build a computation graph that spans both symbolic and neural operations, is highly parallelizable, and end-to-end differentiable on GPUs. Finally, DOLPHIN is implemented as a library that is integrated with PyTorch, allowing users to easily incorporate it into their existing deep learning pipelines.

We evaluate DOLPHIN on a diverse set of neurosymbolic tasks that involve text, image, video, and multi-modal data, and use rich reasoning features such as recursion and black-box Python functions. Neurosymbolic programs written using DOLPHIN only require 0.33%-37.17% (2.77% on average) of the time to train compared to state-of-the-art baselines including differentiable reasoning frameworks like Scallop, and sampling-based frameworks like ISED and IndeCateR+ while maintaining similar levels of accuracy. We also observe that DOLPHIN efficiently scales to more complex benchmarks and larger datasets whereas the baselines either time out after 10 hours or fail to converge.

We make the following contributions in this work:

- We propose DOLPHIN, a novel neurosymbolic programming framework for end-to-end differentiable symbolic reasoning in a scalable manner.
- We develop novel abstractions to represent symbolic and neural computations and introduce vectorized primitives for neurosymbolic programs.
- We develop vectorized provenances that can be plugged into DOLPHIN for efficient computation of symbolic gradients on parallelizable hardware such as GPUs.
- We evaluate DOLPHIN on a diverse range of challenging neurosymbolic tasks across different domains and show that it effectively scales with increasing problem complexity and dataset size.

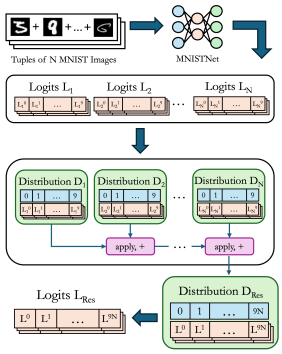
2 OVERVIEW

We illustrate our approach using the MNIST Sum-N task from (De Smet et al., 2024). The goal is to train a model that takes as input N images of MNIST digits and returns the sum of the digits

represented by the images. During learning, supervision is provided only on the sum instead of the labels of the digits. The difficulty of the problem scales exponentially as there are 10^N states in the input space. Further, there are only $9N + 1$ possible labels, resulting in very sparse supervision.

```
1  class MNISTNet(nn.Module):
2    def __init__(self):
3      super(MNISTNet, self).__init__()
4      ...
5
6  class SumNNet(nn.Module):
7    def __init__(self):
8      super(SumNNet, self).__init__()
9      self.CNN = MNISTNet()
10
11   def forward(self, imgs):
12     digits = range(10)
13     D_res = Distribution(self.CNN(imgs[0]), digits)
14     for i in range(1, len(imgs)):
15       D_i = Distribution(self.CNN(imgs[i]), digits)
16       D_res = apply(D_res, D_i, lambda x,y: x + y)
17     l_res = get_logits(D_res)
18     return l_res
```

(a) Code using DOLPHIN primitives with PyTorch.



(b) Components of the `SumNNet` model.

Figure 2: A neurosymbolic program for the MNIST Sum-N task written using DOLPHIN.

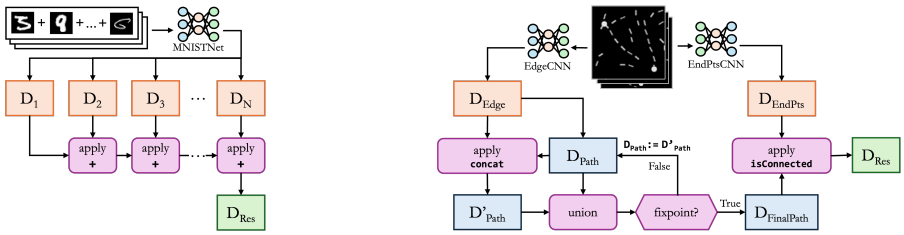
Figure 2a shows the code for this task using DOLPHIN with PyTorch. The neural module is a convolutional neural network (CNN) called `MNISTNet`. It takes in a batch of image tuples `imgs` where each sample contains N MNIST images. `MNISTNet` classifies each image into one of 10 classes representing the digits 0-9. The logits produced by `MNISTNet`, representing probability distributions over the digits, are then passed as inputs to the symbolic program. Lines 13-17 depict a symbolic program written in Python using DOLPHIN primitives.

In order to support training, the symbolic program must reason over all the outputs of the CNN, and return probability distributions over all the possible results (0 to $9N$). This involves tracking the probabilities of individual *symbols* (here, digits or numbers), combinatorially evaluating the results of complex symbolic functions, and calculating the probabilities of each intermediate result, all while tracking their gradients to allow for accurate backpropagation while optimizing the training objective. The batched nature of data in machine learning further complicates these calculations since the probabilities of symbols can be different across samples within the same batch. As a result, writing neurosymbolic programs in native PyTorch is tedious even for simple tasks.

To address these issues, DOLPHIN provides primitives that allow programmers to express symbolic programs without worrying about the underlying computations. Lines 13 and 15 of Figure 2a show how the CNN's output can be captured within `Distribution` objects. Each `Distribution` associates the `digits` with the corresponding batched logits produced by the CNN, along with any gradients and associated metadata. Figure 2b shows the internal structure of these objects.

The programmer can now express the symbolic program in terms of operations that manipulate `Distributions`. For instance, in line 16, the `apply` function is used to perform an operation on two distributions. Here, the `apply` function takes two `Distributions` as arguments, along with a lambda function that specifies the addition operation. Under the hood, `apply` combinatorially explores all the possible sums of the symbols from `D_res` and `D_i` and calculates their associated probabilities using an appropriate provenance. The result of `apply` is a new `Distribution` over the calculated sums, and is stored back into `D_res`. This is repeated iteratively until all the outputs of the CNN are summed appropriately. Once the final `Distribution` is calculated, it is simply a matter of getting its logits which can be used to calculate the loss of the predictions.

DOLPHIN provides additional primitives to support more complex symbolic programs. Figure 3b, for instance, shows the computation graph for the PathFinder task (Tay et al., 2020), which involves recursively building paths to identify if two points in a maze are connected. The `union` primitive is used to support the recursive nature of this program. As with `apply`, DOLPHIN maps the symbolic operations denoted within these primitives to probability computations. Given that `Distribution` objects associate symbols with the batched logits themselves, these computations are vectorized and directly operate over PyTorch tensors. This deep integration of DOLPHIN into PyTorch allows programmers to write symbolic programs as *symbolic layers* that interact with stan-



(a) MNIST Sum-N. (b) PathFinder.

Figure 3: Computation graphs for neurosymbolic programs written using DOLPHIN.

Symbol	::	s	$\in$	S	(objects)	APPLY	: $\mathbb{D}^K \times (S^K \to S) \to \mathbb{D}$
Tag	::	t	$\in$	T	(tensors)	FILTER	: $\mathbb{D} \times (S \to \mathbb{B}) \to \mathbb{D}$
Distribution	::	D	$\in$	$\mathbb{D} = S \to T$		APPLYIF	: $\mathbb{D}^K \times (S^K \to S) \times (S^K \to \mathbb{B}) \to \mathbb{D}$
						UNION	: $\mathbb{D} \times \mathbb{D} \to \mathbb{D}$
						GETPROBS	: $\mathbb{D} \to [0,1]^N$

Figure 4: Formal definition of DOLPHIN's programming abstractions and primitives.

dard PyTorch neural layers within a neurosymbolic model. Furthermore, DOLPHIN can leverage the hardware acceleration supported by PyTorch to scale to large and complex programs. This is in contrast to systems like Scallop (Li et al., 2023), where tensors are converted into Scallop-friendly tags that are transferred to a process outside the Python environment, and where the probability computations are CPU-bound, restricting its scalability.

3 THE DOLPHIN FRAMEWORK

3.1 DOLPHIN CORE DESIGN PRINCIPLES

We based DOLPHIN's framework design on the following core principles:

- **Flexible programmability:** The framework should allow developers to write neurosymbolic applications in Python with minimal effort, providing intuitive primitives that seamlessly integrate with Python's rich and expressive language features.
- **End-to-end differentiability on GPUs:** The framework should allow any neurosymbolic program to be end-to-end differentiable on GPUs irrespective of the task characteristics.
- **Scalable:** The framework should easily scale with greater problem and data complexity.
- **Tunable:** Similar to hyperparameters in deep learning, the framework should provide a simple interface for developers to choose provenances (and their configurations) or define new ones.

Next, we show how we realize these principles by describing the key components of DOLPHIN.

3.2 THE DOLPHIN SYNTAX

DOLPHIN provides a programming interface that developers can use to express symbolic programs in a Pythonic manner. DOLPHIN maps each operation of the symbolic program to PyTorch which enables end-to-end GPU-accelerated differentiable reasoning. Figure 4 presents DOLPHIN's programming interface including the symbolic abstractions and operations over them.

3.2.1 ABSTRACTIONS

The three main abstractions provided by DOLPHIN for expressing differentiable symbolic programs are shown on the left of Figure 4. *Symbols* S represent symbolic entities relevant to the program. These entities can be any Pythonic object, such as hand-written digits in MNIST-SumN or coordinates of points in PathFinder. *Tags* T are tensors that represent relative likelihoods. Typically, tags for symbols are derived from the outputs of machine learning models, such as the probability distribution over digits produced by the CNN classifier in MNIST-SumN. Finally, *Distribution* D represents the likelihood of an input being classified as one of the pre-defined symbols.

Distributions serve as the fundamental datatype of a DOLPHIN program and act as its main interface with a machine learning model. For instance, when the developer instantiates a Distribution object, such as in the following code snippet from Figure 2a:

```
D_res = Distribution(self.CNN(imgs[0]), digits)
```

the output of the CNN model is directly passed to the Distribution object, effectively acting as an input to the symbolic program. The Distribution object itself, as shown in Figure 2b, contains batches of *tags* extracted from the model outputs, and maintains the set of corresponding symbols.

To enable such a seamless integration between the PyTorch model and the symbolic program, Distributions are designed to operate directly over PyTorch tensors. This has two main advantages. First, it preserves the gradients of the model output throughout the symbolic program, enabling end-to-end differentiability. Second, it allows DOLPHIN to perform operations over an entire batch of tags, leveraging the vectorized operations provided by PyTorch. DOLPHIN can thus operate efficiently on specialized hardware like GPUs, allowing the symbolic program to scale effectively.

3.2.2 OPERATIONS

Figure 4 shows the five operations supported by DOLPHIN that developers can use to manipulate Distributions and express complex symbolic programs. We now expand on these operations.

APPLY. This is the primary operation that developers can use to manipulate Distributions. It takes as inputs K Distributions, where $K \geq 1$, along with a function f of the same arity. This function defines operations over the symbols of K distributions. APPLY then computes the results of f over all possible combinations of arguments sourced from the symbols of the Distributions as well as their associated tags, and returns a new Distribution with these results and tags.

This operation occurs in two stages akin to the popular map-reduce pattern. In the *map* stage, APPLY computes the results of f over the symbols of the input Distributions and their associated tags:

$$R \;=\; \{\, (f(s_1, s_2, \ldots, s_k), (t_1 \otimes t_2 \otimes \ldots \otimes t_k)) \mid D_i(s_i) = t_i, i = 1, \ldots, k \,\} \tag{1}$$

Here, the tag of each result symbol $f(s_1, s_2, \ldots, s_k)$ is the conjunction $\otimes$ of the tags $(t_1, t_2, \ldots, t_k)$ of the input symbols it was derived from.

Note that the function f can be any user-defined Python function, including complex functions like regex parsing, image processing, or Python's `eval()`. As a result, f may be a many-to-one function and the tags shared by a resulting symbol must be aggregated to form the final tags of the output Distribution. We, therefore, *shuffle* the results from the map stage to compute a function M from each symbol to tags from R associated with it:

$$M = \lambda\, s\, .\, \{\, t \mid (s, t) \in R \,\} \tag{2}$$

We then proceed to the *reduce* stage, where we aggregate the tags of each symbol in M using disjunction $\oplus$ to produce the final Distribution D_{res}:

$$D_{\text{res}} = \lambda\, s\, .\, \bigoplus \{\, t \mid t \in M(s) \,\} \tag{3}$$

Since the tags here are PyTorch tensors representing probabilities, the implementations of the conjunction and disjunction operations are dictated by the underlying provenance used by the program. A more detailed explanation of the provenances is provided in Section 3.3.

FILTER. The FILTER operation is used to filter out symbols from a Distribution based on some condition. It takes in a single Distribution, along with a user-defined function that returns a boolean value, which acts as the condition. This operation then returns a new Distribution that contains only the symbols that satisfy the condition, along with their tags.

APPLYIF. The APPLYIF operation is a conditional version of APPLY. It takes in K Distributions and two functions f_{apply} and f_{cond} of the same arity. For each combination of symbols from the K Distributions, APPLYIF computes the results of f_{apply} and its associated tags only if the condition f_{cond} is satisfied over that combination of symbols. The operation then returns a new Distribution with these results and tags.

UNION. The UNION operation is used to combine two Distributions. It takes in two Distributions and returns a new Distribution that contains the union of the symbols from the two input Distributions, along with their tags. Any symbols common to both input Distributions have their tags

Provenance	Domain	$\mathbf{0}$	$\mathbf{1}$	$t \oplus t'$	$t \otimes t'$
DAMP	$[0, 1]$	0	1	$\text{clamp}_0^1(t + t')$	$t \cdot t'$
DTKP-AM	$[0, 1] \cup \{\infty, -\infty\}$	$\hat{\mathbf{0}}_{ij} = \begin{cases} \infty & i = 1 \\ -\infty & i > 1 \end{cases}$	$\hat{\mathbf{1}}_{ij} = \infty$	$\text{top}_k(\text{cat}(t, t'))$	$\text{top}_k([\min(\text{abs}(t_i), \text{abs}(t'_j)) \mid t_i \in t, t'_j \in t'])$

Table 1: DOLPHIN provenances implemented in PyTorch.

merged via a disjunction operation. UNION is especially useful when writing recursive programs in DOLPHIN that require combining the results of multiple recursive calls.

GETPROBS. The GETPROBS operation is used to extract the probabilities from the tags of a Distribution. This is used mainly once the symbolic program has been executed to extract the final probabilities of the symbols in the output Distribution. These probabilities can then be used to compute the loss function for training the neural components of the neurosymbolic model. The actual extraction of the probabilities from the tags depends on the specific provenance used in the program, which we discuss next.

3.3 DOLPHIN PROVENANCES

As discussed earlier, each symbol in a distribution is associated with a batch of one or more tags. The DOLPHIN primitives define how to manipulate certain tags. For instance, Equations (1) and (3) specify the tags to be conjuncted or disjuncted. We now define the semantics of these operations.

The goal of such operations is to approximate the probabilities of the symbols in the output distribution as accurately as possible. This is achieved by using a mathematical framework called *provenance semirings* (Green et al., 2007). Provenance semirings provide generalized algebraic structure to propagate tags when computing over tagged data. In the case of DOLPHIN distributions, we can view the tags as representing the probabilities, and the data as the distribution's symbols.

Designing and implementing provenances can be challenging since they must be accurate enough to capture the semantics of the symbolic program, while at the same time being coarse enough to maintain computational feasibility. Furthermore, the provenances must be differentiable to enable end-to-end training for neurosymbolic tasks. While neurosymbolic frameworks like Scallop (Li et al., 2023) implement differentiable provenances, they are not designed to leverage hardware accelerations or batched optimizations due to the CPU-bound nature of their implementations. We thus design vectorized provenances in DOLPHIN that are differentiable and enable GPU computations.

We simplify the definition of provenances from Scallop as a 5-tuple: $(T, \mathbf{0}, \mathbf{1}, \otimes, \oplus)$. Here, T is the tag space, $\otimes : T \times T \to T$ is the conjunction operator with identity $\mathbf{0}$, and $\oplus : T \times T \to T$ is the disjunction operator with identity $\mathbf{1}$. We then implement two differentiable provenances in DOLPHIN: Differentiable Add-Mult Probabilities (DAMP) and Differentiable Top-K Proofs (DTKP). Table 1 summarizes the operations of these provenances.

Differentiable Add-Mult Probabilities. Differentiable Add-Mult Probabilities (DAMP) is a popular technique that uses the probability space as its tag space: $T = [0, 1]$. Its conjunction operation $\otimes$ is defined as the addition of probabilities, clamped at $\mathbf{1}$, and its disjunction operation $\oplus$ is defined as the multiplication of probabilities. The main assumption underlying the DAMP operations is that the input Distributions are mutually exclusive and independent. This assumption allows DAMP to compute probabilities extremely efficiently, as the operations are simple and can be easily vectorized.

Differentiable Top-k Proofs. Differentiable Top-k Proofs (DTKP) (Huang et al., 2021) was proposed to overcome the shortcomings of DAMP. This provenance tracks a set of up to k *proofs* for each symbol. Each proof, in turn, denotes the set of input symbols necessary to derive the output symbol. These proofs are then used to compute the probabilities of the output symbols.

In Scallop, DTKP tags are converted into probabilities via differentiable weighted model counting (WMC). This form of DTKP is computationally hard and is by nature difficult to vectorize due to the varying sizes of proof sets and the WMC procedure. We therefore design a vectorized version of DTKP, called DTKP-AM (DTKP with Add-Mult), that can be efficiently computed on GPUs.

We first define the structure of tags in DTKP-AM in a manner that conforms to the constraints of PyTorch tensors. Each tag t for a symbol s is a 2-dimensional tensor of shape $(k, |I|)$, where k is the maximum number of proofs to be retained and I is an ordered list of all *input symbols* (symbols

that are present in the input Distributions). Each row t_i of t corresponds to one of the tag's k proofs. Each element t_{ij} thus represents the probability of the jth input symbol in the ith proof:

$$t_{ij} = \begin{cases} p_j & \text{if the } j\text{th symbol is present in the } i\text{th proof} \\ \hat{\mathbf{0}}_{ij} & \text{otherwise} \end{cases}$$

where p_j is the probability of the jth input symbol. The probability of each proof is then computed by taking the product of the normalized elements of that proof:

$$\Pr(t_i) = \prod_j \text{norm}(t_{ij}) \quad \text{where} \quad \text{norm}(t_{ij}) = \begin{cases} 1 & t_{ij} = +\infty \\ 0 & t_{ij} = -\infty \\ t_{ij} & \text{otherwise} \end{cases}$$

We next define the operations of DTKP-AM in Table 1. The $\oplus$ operation is defined as the union of two tag tensors t and t' while $\otimes$ is defined as the element-wise minimum of the absolute values of all possible combinations of proofs in t and t'. In each case, the top$_k$ operation retains only upto k proofs with the highest probabilities.

These definitions thus allow us to take advantage of the benefits of the DTKP provenance while enabling efficient computation on GPUs. To calculate the probability of the entire tag, DTKP-AM adds the probabilities of the individual proofs and clamps it at 1.

3.4 Building the Dolphin Program

The programmer specifies the neurosymbolic task using a Python program P that uses Dolphin's programming interface to connect the neural components (e.g., neural networks) with the symbolic components and operations. We call P the *symbolic program*. Because P is a Python program and Dolphin interfaces with PyTorch, Dolphin supports any PyTorch-based neural network(s), most Python language features, and custom user-defined functions. This feature enables greater flexibility and expressiveness in neurosymbolic programs than existing frameworks.

In addition to P, the programmer provides one or more neural networks $M_1, \ldots, M_k$, and a dataset $\mathcal{D}$. Given these inputs, Dolphin extracts the computation graph that encodes how the neural network outputs are transformed using symbolic operations to produce a final result D_{res}. All computations in Dolphin are expressed using distribution objects D_i. Each Dolphin primitive (e.g., Apply) takes one or more distribution objects as inputs and applies a transformation to produce another distribution object. Because each distribution object D_i only contains vectors of tags (or probabilities), the entire computation graph (including the neural network(s)) can be ported to a GPU for faster execution. During training, Dolphin optimizes over the standard objective function:

$$\phi(\theta) = \min_\theta \sum_{(x,y)\in\mathcal{D}} \mathcal{L}(P(M_\theta(x)), y) \tag{4}$$

Here $\mathcal{L}$ is the loss function, such as binary cross entropy. While Dolphin allows P to take multiple neural networks as inputs, we show only one neural network model M here for simplicity.

4 Experiments

We evaluate Dolphin on a set of 13 benchmarks of varying complexity and scale across 5 neurosymbolic tasks. Our evaluation addresses the following research questions:

- **RQ1: Scalability.** How does Dolphin scale to complex problems and large datasets compared to existing baselines?
- **RQ2: Accuracy.** Do models written in Dolphin converge to SOTA accuracies?
- **RQ3: Provenance Comparisons.** Which provenances are most effective for each benchmark?

4.1 Benchmarks

MNIST-SumN. The MNIST-SumN (or briefly, SumN) task from (De Smet et al., 2024) takes as inputs N handwritten digits from the MNIST dataset and returns their sum. We consider three versions of this task: small ($N = 5$), medium ($N = 10$), and large ($N = 15$). The perception model for this task is a standard CNN while the symbolic program sums the outputs of this model.

Hand-Written Formula (HWF). The HWF task from Li et al. (2020) takes as input a set of images of handwritten digits and arithmetic operators representing a formula. The task is to evaluate the formula and return the result. We consider three versions of HWF: small (formulas of length up to 7), medium (formulas of length up to 15), and large (formulas of length up to 19). Similar to MNIST-SumN, the perception model for this task is a standard CNN. The DOLPHIN program for this task builds strings of formulae from the outputs of the neural model and evaluates them using Python's `eval` function, demonstrating the ability of DOLPHIN to support black-box functions.

PathFinder. PathFinder (or Path) from Tay et al. (2020) tests the ability of an agent to reason over long-range dependencies within an input image. The image consisting of two dots and a sequence of curved and dashed lines. The task is to identify whether the two dots are connected via the lines. We consider three versions of this task based on the image size in pixels: small (32 x 32), medium (128 x 128), and large (256 x 256). The perception model for this task is also a standard CNN. However, the DOLPHIN program for this task is recursive, since it must search for paths between the two dots.

CLUTRR. In this task from Sinha et al. (2019), the input is a passage of text containing some information about several individuals and some of their relationships. The task is then to infer the relationship between two given individuals, which is not explicitly provided in the input. Hence, the DOLPHIN program for this task needs to reason over long chains of relationships recursively. We consider two versions of this task, where the training data contains relation chains of lengths up to 3 (small) or 4 (medium). We use Roberta-base (Liu, 2019) as the language model for this task.

Mugen. In this task from Hayes et al. (2022), the input is a video of gameplay footage that is 3.2 seconds long and a natural language passage captioning the video. The goal is to measure how aligned the text is with the video. This task therefore has two variants: Mugen-TVR, where the model must retrieve the video that best aligns with the text, and Mugen-VTR, where the model must retrieve the text that best aligns with the video. This benchmark tests the ability of the model to reason over multimodal data. We consider two versions of this task: small, with 1000 training samples, and medium, with 5000 training samples. We use a combination of DistilBert (Sanh, 2019) and S3D (De Smet et al., 2024) as the perception model for the text and video inputs respectively.

4.2 EXPERIMENTAL SETUP AND BASELINES

Setup. All experiments, with the exception of the CLUTRR benchmark, were run on machines with two 20-core Intel Xeon Gold 6248 CPUs, four NVIDIA GeForce RTX 4352 GPUs, and 768 GB RAM. Since the CLUTRR benchmark requires a larger amount of GPU memory, it was run on a machine with 8 NVIDIA A100 40GB GPUs instead. We ran each tool thrice until convergence or until a timeout of 10 hours was reached and report the average best accuracy and training time. The code of DOLPHIN and the benchmarks is provided in the supplementary material.

Baselines. We select Scallop (Li et al., 2023), a contemporary state-of-the-art neurosymbolic framework supporting differentiable programming optimized to run on the CPU and use multiple cores to parallelize its computations. We also choose two sampling-based gradient approximation methods, ISED (Solko-Breslin et al., 2024) and IndeCateR+ (De Smet et al., 2024). We compare DOLPHIN against Scallop on all benchmarks, and against ISED and IndeCateR on MNIST-SumN and HWF.

4.3 RQ1: SCALABILITY

Table 2 presents the training times for DOLPHIN and baselines on all benchmarks, along with the total training time (T_{total}), time per epoch (T_{epoch}), and the scaling factor α. The scaling factor is the ratio of the per epoch times of the baselines to DOLPHIN. We observe that in almost all cases, DOLPHIN shows significant improvements in training times, training up to about 300x faster than Scallop, 44x faster than ISED, and 180x faster than IndeCateR+. On average, DOLPHIN reports a speedup of 36x times over all baselines. Further, DOLPHIN converges on all benchmarks, while the baselines time out on most of the Medium and Large versions of the benchmarks. These results thus demonstrate that, unlike existing tools, DOLPHIN can scale to complex problems and large datasets.

DOLPHIN trains slightly slower than Scallop on both versions of the Mugen benchmark. This is because the DOLPHIN program written for Mugen uses Python objects and operations that are not fully batchable across samples. In contrast, the Scallop program, which is written in a compiled and optimized language, runs around 1.3x faster than DOLPHIN on average per iteration. However,

Task	DOLPHIN		Scallop			ISED			IndeCateR+		
	T_{total}	T_{epoch}	T_{total}	T_{epoch}	α	T_{total}	T_{epoch}	α	T_{total}	T_{epoch}	α
SumN (S)	78.33	**15.67**	923.78	184.76	11.80	299.63	59.93	3.82	3.16e3	632.52	40.36
SumN (M)	144.92	**14.49**	3.41e3	341.57	23.57	2.16e3	216.54	14.94	2.63e4	2.63e3	181.88
SumN (L)	220.47	**14.70**	7.41e3	493.87	33.60	9.8e3	653.39	44.45	TO	2.27e3	154.37
HWF (S)	3.17e3	**158.79**	9.99e3	499.57	3.15	1.58e4	790.04	4.97	1.76e4	877.92	5.52
HWF (M)	1.46e4	**731.67**	TO	1.49e4	20.41	TO	6.83e3	9.34	TO	3.6e3	4.93
HWF (L)	2.42e4	**1.21e3**	TO	3.92e5	323.21	TO	1.05e4	8.61	TO	5.59e3	4.61
Path (S)	1.08e4	**1.08e3**	2.2e4	2.2e3	2.03	N.A.					
Path (M)	1.79e4	**1.79e3**	TO	4.17e3	2.32						
Path (L)	1.94e4	**1.94e3**	TO	1.12e4	5.81						
CLUTRR (S)	1.54e3	**154.85**	4.29e3	429.97	2.77	N.A.					
CLUTRR (M)	2.91e3	**291.36**	7.83e3	783.11	2.69						
Mugen (S)	3.62e3	180.80	1.34e4	**133.68**	0.74	N.A.					
Mugen (M)	1.78e3	890.34	TO	**634.86**	0.71						

Table 2: Comparison of training times taken by each baseline. The Timeout (TO) is set at 10 hours. α is the scaling factor, which is the ratio of the per epoch training times of the baselines and DOLPHIN.

DOLPHIN requires only 20 epochs to converge whereas Scallop requires almost 1000 epochs (Li et al., 2023). As a result, DOLPHIN's total training time is still significantly lower than Scallop's (3.7x for Mugen(S)). We show the accuracy curves for Mugen in Appendix B.

4.4 RQ2: ACCURACY

Figure 5 presents the accuracy of DOLPHIN and the baselines on the different benchmarks. DOLPHIN accuracies are marked in blue. In all cases, for DOLPHIN, we report the accuracies of the best-performing provenance. We use the DAMP provenance for MNIST, CLUTRR, and Mugen benchmarks, and the DTKP-AM provenance for the HWF and PathFinder benchmarks.

We observe that in all cases, DOLPHIN achieves state-of-the-art accuracy, except in CLUTRR, where DOLPHIN's accuracy is slightly lower than that of Scallop. For most Medium and Large versions of the benchmarks, DOLPHIN achieves better accuracy, whereas the baselines either report lower accuracy due to the complexity of the benchmark (e.g., HWF) or fail to converge within the time limit (e.g., Scallop on PathFinder-Large). Most importantly, these results show that DOLPHIN's scalability improvements do not come at the cost of accuracy.

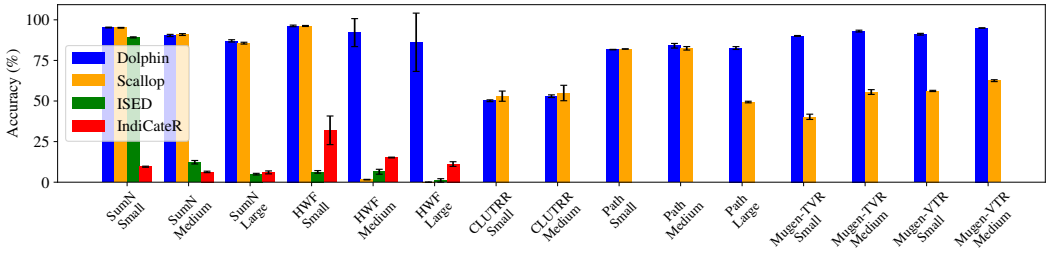


Figure 5: Accuracy of DOLPHIN and baselines across all benchmarks.

4.5 RQ3: PROVENANCE COMPARISONS

We perform ablation studies to compare the effectiveness of the DAMP and DTKP-AM provenances for each benchmark. Figure 6 presents the accuracy of DOLPHIN using DAMP and DTKP-AM in the top graph, and the average training time per epoch in the bottom. We observe that the relative effectiveness of the provenances differ across benchmarks. For instance, in the MNIST-SumN benchmark, the DAMP provenance is more effective than the DTKP-AM provenance, since the top-k proofs are not able to capture all the possible ways in which the sum of the digits can be computed. In contrast, for HWF, the DTKP-AM provenance is more effective than DAMP since the tags in DAMP provenance lack the structure needed to capture the semantics of the symbolic program. Otherwise, DTKP-AM mostly has a slight edge over DAMP. On the other hand, DAMP is faster than DTKP-AM for all benchmarks, since it does not track individual proofs. We thus see that selecting provenances is a trade-off between accuracy and training time.

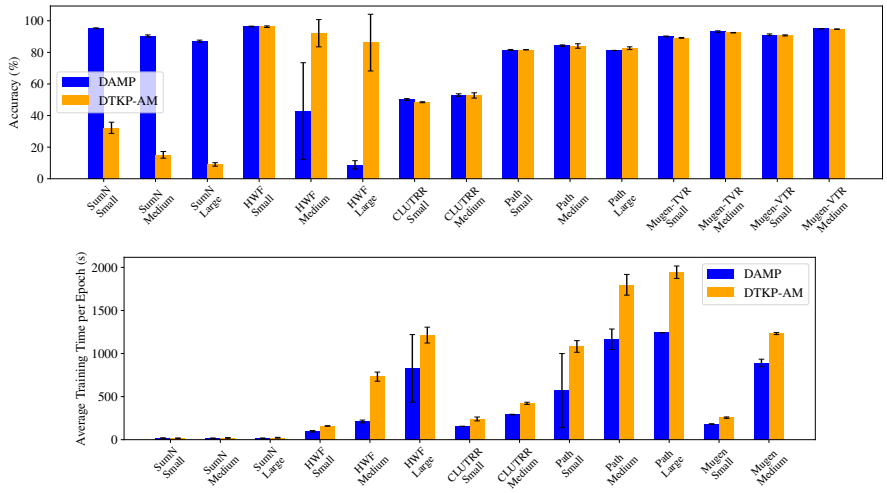


Figure 6: Accuracy and average training time per epoch for DAMP and DTKP-AM.

5 Related Work

Neurosymbolic programming frameworks. Frameworks like Scallop (Li et al., 2023), DeepProbLog (Manhaeve et al., 2018), and ISED (Solko-Breslin et al., 2024) provide a simple interface for neurosymbolic programming. There are also domain-specific tools like NeurASP Yang et al. (2023) for answer set programming and NeuralLog Chen et al. (2021) for phrase alignment in NLP. While these frameworks provide intuitive abstractions, they are bottle-necked due to expensive data transfers between symbolic computations done on CPU versus neural computations that execute on GPU, making neurosymbolic learning hard to scale. In contrast, Dolphin provides a deeper integration of the two worlds by building a Python-based API on top of PyTorch, which scales better.

Scalability improving techniques. Several optimization techniques have been proposed to improve the scalability of differentiable reasoning algorithms. For instance, Greedy NTP (Minervini et al., 2020a) reduces the computation cost of Neural Theorem Prover (NTP) (Rocktäschel & Riedel, 2017) by tracking only a subset of proof states using nearest neighbor search. Likewise, the conditional theorem prover (Minervini et al., 2020b) employs a machine learning-based proof selection technique. However, unlike Dolphin, these methods are point solutions that do not fundamentally address the scalability challenge for neurosymbolic learning.

Specialized neurosymbolic solutions. There are many specialized solutions for various neurosymbolic tasks. For instance, NGS (Li et al., 2020) uses a hand-coded syntax to specify the structure of mathematical expressions for HWF. More general solutions, such as NS-CL (Mao et al., 2019) includes a framework for visual question answering that learns symbolic representations for text and images. NeRd (Chen et al., 2021) transforms questions in natural language into executable programs based on symbolic information extracted from text. In this work, we rather focus on building a general programming framework that improves the scalability of diverse neurosymbolic programs.

6 Conclusion and Limitations

We proposed Dolphin, a programmable framework for scaling neurosymbolic learning. Dolphin provides abstractions for writing symbolic programs along with pluggable vectorized provenances to compute symbolic gradients. This allows users to write differentiable symbolic programs in Python within PyTorch pipelines that can scale to complex programs and large datasets. We show that Dolphin scales significantly better than existing neurosymbolic frameworks while achieving state-of-the-art performance on a variety of tasks.

A limitation of Dolphin is that it requires the user to express symbolic programs in a batched manner. While this is a common pattern within deep learning, it may be restrictive to users new to batched programming. Another limitation is that while Dolphin works well with most models, the representation of Distributions needed by generative models (e.g., Causal LLMs like Llama) has not yet been investigated. We leave this to future work.

REFERENCES

Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, Yisong Yue, et al. Neurosymbolic programming. *Foundations and Trends® in Programming Languages*, 7(3):158–243, 2021.

Zeming Chen, Qiyue Gao, and Lawrence S Moss. Neurallog: Natural language inference with joint neural and logical reasoning. *arXiv preprint arXiv:2105.14167*, 2021.

Lennert De Smet, Emanuele Sansone, and Pedro Zuidberg Dos Martires. Differentiable sampling of categorical distributions using the catlog-derivative trick. *Advances in Neural Information Processing Systems*, 36, 2024.

Todd J Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 31–40, 2007.

Thomas Hayes, Songyang Zhang, Xi Yin, Guan Pang, Sasha Sheng, Harry Yang, Songwei Ge, Qiyuan Hu, and Devi Parikh. Mugen: A playground for video-audio-text multimodal understanding and generation. In *European Conference on Computer Vision*, pp. 431–449. Springer, 2022.

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

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Kaya Stechly, Mudit Verma, Siddhant Bhambri, Lucas Saldyt, and Anil Murthy. Llms can't plan, but can help planning in llm-modulo frameworks. *arXiv preprint arXiv:2402.01817*, 2024.

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

Ziyang Li, Jiani Huang, and Mayur Naik. Scallop: A language for neurosymbolic programming. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1463–1487, 2023.

Yinhan Liu. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. *Advances in neural information processing systems*, 31, 2018.

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

Pasquale Minervini, Matko Bošnjak, Tim Rocktäschel, Sebastian Riedel, and Edward Grefenstette. Differentiable reasoning on large knowledge bases and natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 5182–5190, 2020a.

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

Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. *Advances in neural information processing systems*, 30, 2017.

V Sanh. Distilbert, a distilled version of bert: Smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

Koustuv Sinha, Shagun Sodhani, Jin Dong, Joelle Pineau, and William L Hamilton. Clutrr: A diagnostic benchmark for inductive reasoning from text. *arXiv preprint arXiv:1908.06177*, 2019.

Alaia Solko-Breslin, Seewon Choi, Ziyang Li, Neelay Velingker, Rajeev Alur, Mayur Naik, and Eric Wong. Data-efficient learning with neural programs. *arXiv preprint arXiv:2406.06246*, 2024.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers. *arXiv preprint arXiv:2011.04006*, 2020.

Zhun Yang, Adam Ishay, and Joohyung Lee. Neurasp: Embracing neural networks into answer set programming. *arXiv preprint arXiv:2307.07700*, 2023.