

A DOLPHIN EXPERIMENT DETAILS FOR BENCHMARKS

The following are more details on the experiment setup. For each experimental trial, we report the highest evaluation accuracy over all epochs. Unless otherwise noted, each trial was run on a machine with Intel Xeon Gold 6248 (2.50 GHz) CPUs and NVIDIA GeForce RTX 2080 Ti (11 GB) GPUs.

A.1 MNIST SUM-N

For this task, the base neural network model is a standard CNN classifying each image into 10 classes of digits (0, 1, ..., 9). The symbolic module sums the Distribution objects over the logits output by the neural model for each image.

Each of the MNIST Sum-N tasks had a batch train size of 64 samples, a learning rate of 0.001, and a top-k value of 1. Each of the tasks were trained on a dataset size of the original MNIST dataset divided by N of Sum-N. Sum5's dataset consisted of 12000 train samples and 2000 test samples. Sum10's dataset consisted of 6000 train samples and 1000 test samples. Sum15's dataset consisted of 4000 train samples and 666 test samples.

A.2 HAND-WRITTEN FORMULA

For Hand-Written Formula, the perception model is again a standard CNN that classifies images into 14 classes: 10 digits (0, 1, ..., 9), and 4 operations (+, -, \times , and /). 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.

We trained each task with a batch train size of 64 samples. The learning rate was 0.0001, sample-k was 7, and top-k value was 3. Length 7's dataset consisted of 9600 samples for training, 2400 samples for testing. Length 15 consisted of 24000 training samples and 6000 testing samples. Length 19 consisted of 32000 training samples and 8000 testing samples.

A.3 PATHFINDER

For this task, the perception model is also a CNN, but it predicts edges between pairs of nodes (denoted by dashes) as well as the end points depicted in the image of the maze. The DOLPHIN program for this task is recursive since it must search for paths between the two dots.

For each of the PathFinder tasks, we used a batch train size of 64 samples, a learning rate of 0.0001, and a top-k value of 3. Each task's dataset consisted of 539459 images for training and 59940 images for testing. Each task had its own dataset of images with dimensions of the task's pixel size.

A.4 CLUTRR

For each CLUTRR task, we used a single A100 GPU (40 GB), with a learning rate of 0.00001 and use a batch size of 16. The length of the training dataset for CLUTRR (Small) was 11,093 and that of the test set was 1146. The training set for CLUTRR (Medium) contained 15,083 samples and the test set contained 1048 samples.

The DOLPHIN program for CLUTRR receives as inputs pairs of entities from the input paragraph along with the logits for each pair over 21 possible relations produced by the classification head of the Roberta-base [Liu \(2019\)](#) model. The program then recursively derives relations over the graphs these pairs represent until no new relations can be derived. After that, it returns the Distribution over relations for the target pair of entities.

A.5 MUGEN

For each Mugen task, we use a batch size of 3 and a learning rate of 0.0001. From the full Mugen dataset, we sample a training set of 5000 examples for Mugen (Medium), and from that set, we sample a training set of 1000 for Mugen (Small). Both Small and Medium are evaluated on a fixed holdout set of 1000 samples. We train and evaluate for up to 100 epochs.

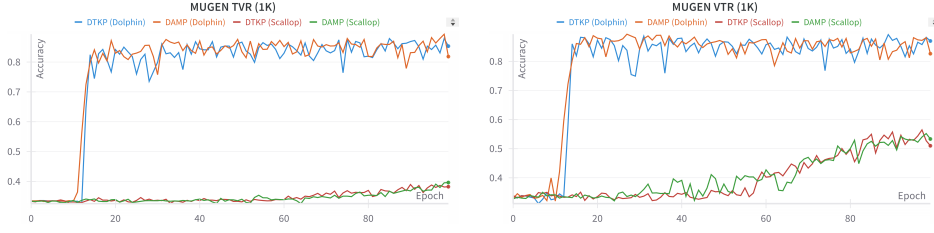


Figure 7: Test accuracies for each epoch of DOLPHIN and Scallop for Mugen (Small). The accuracies for TVR are on the left and VTR are on the right.

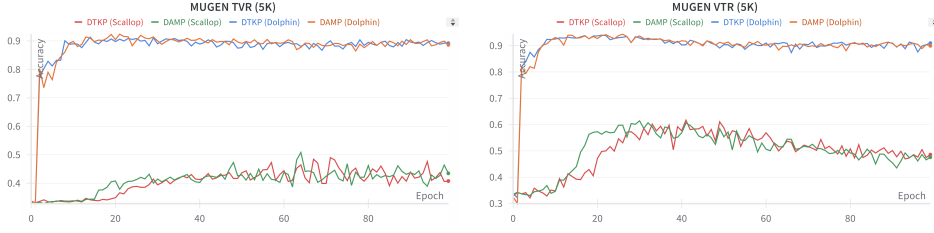


Figure 8: Test accuracies for each epoch of DOLPHIN and Scallop for Mugen (Medium). The accuracies for TVR are on the left and VTR are on the right.

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. The DOLPHIN program for both Mugen tasks computes the temporal alignment of a given text-video pair. The inputs (extracted from the text and video by neural components) are pairs of IDs and actions, where the ID order corresponds to the action sequence (e.g. the IDs for video actions are the frame numbers). The program finds the Distribution of all valid mappings between text event IDs and video frame IDs that preserve the order of actions.

A.6 ACCURACY CURVES FOR MUGEN

Figures 7 and 8 show the test accuracies for each epoch of DOLPHIN and Scallop for Mugen (Small) and Mugen (Medium) respectively. The accuracies for TVR are on the left and VTR are on the right. Note that DOLPHIN converges within 20 epochs in all cases, while Scallop does not converge within 100 epochs. Since DOLPHIN and Scallop implement identical symbolic programs and neural network components, we attribute DOLPHIN’s better convergence to differences in the neurosymbolic backend. Importantly, the two use different frameworks for differentiable symbolic reasoning: DOLPHIN uses PyTorch while Scallop uses its own implementation of auto-differentiation. There could be optimizations in PyTorch that are absent from Scallop, where the performance boost is magnified by the relative complexity of Mugen compared to the other tasks. However, this claim requires further investigation to confirm.

B GRAPH RESULTS OF RQ3

We show the results of the provenance comparison experiments (RQ3) in Figure 9. The graph on the top shows the accuracies achieved by each provenance over all tasks, while the bottom graph shows the average training time per epoch required for each provenance over all tasks.

C DTKP-AM PROVENANCE

We clarify and expand on some aspects of the DTKP-AM provenance.

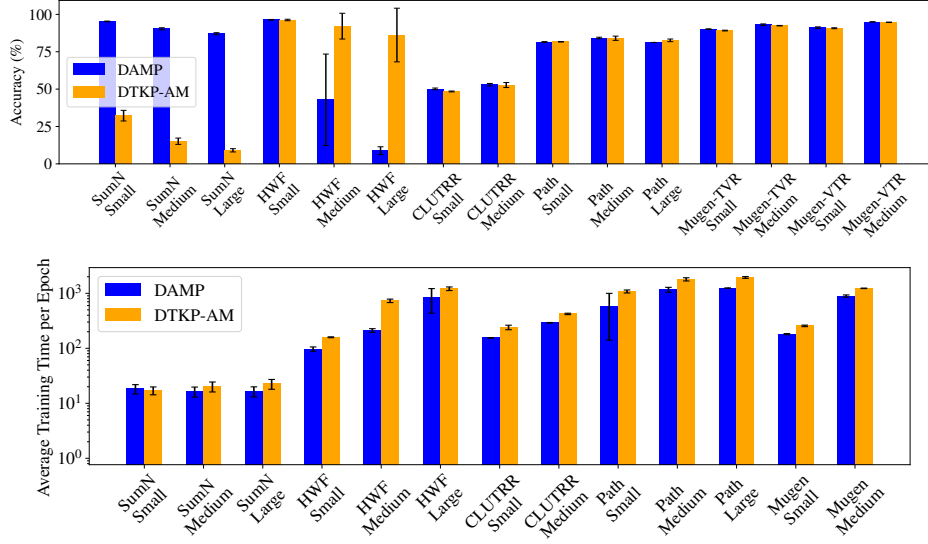


Figure 9: Accuracy and average training time per epoch in seconds for DAMP and DTKP-AM.

C.1 WMC APPROXIMATION

In this section, we emphasize that DTKP-AM does not perform precise weighted model counting (WMC) and address possible shortcomings that could arise. A hardware-efficient vectorization of exact WMC is beyond the scope of this paper, and is itself an active area of research. Instead, we use the following add-mult approximation of WMC:

$$\Pr(t) = \sum_i \Pr(t_i) = \sum_i \prod_j \text{norm}(t_{ij})$$

We note that this approximation upper bounds the result from DTKP-WMC: the coarseness arises from the summation, which may double count models that satisfy more than one of the proofs. However, add-mult achieves significant computational speedup since it simplifies the exponential enumeration over all possible models into a linear pass over the tag’s elements.

We further claim that this approximation does not destroy *all* the semantics from DTKP-WMC due to DTKP-AM’s faithful implementation of the semiring operations \oplus and \otimes for tracking top-k proofs. DTKP-AM tags therefore remain similar to DTKP-WMC tags at every intermediate symbolic reasoning step. By contrast, the imprecise add-mult is a one-time transformation of the final tags into probabilities, performed only after the tags have been propagated through the entire symbolic program. Crucially, we show there exists information that is uniquely captured by top-k tag operations, and is not lost when fuzzily converting the tags to probabilities.

As a simple illustrative example, consider using APPLY with the following toy function:

$$f(a, b) = \begin{cases} \mathbb{T} & a = b \\ \mathbb{F} & \text{otherwise} \end{cases}$$

For any distribution D of mutually exclusive input symbols (e.g. the digit classification of a CNN), we intuitively would like the distribution $f(D, D)$ to assign a probability of 1 to symbol \mathbb{T} and a probability of 0 to symbol \mathbb{F} . According to our semantics, the tag for \mathbb{T} is actually given by:

$$f(D, D)(\mathbb{T}) = \bigoplus_i (D(i) \otimes D(i))$$

However, if we were to use DAMP to compute the tags for $f(D, D)$, the provenance treats the two input distributions as independent when they are the exact same distribution. Thus, the probability assigned to \mathbb{T} by $f(D, D)$ is incorrectly calculated as:

$$f(D, D)(\mathbb{T}) = \sum_i (D(i))^2$$

On the other hand, consider any top-k provenance that satisfies:

$$t \otimes t' = \text{top}_k(\{t_i \cup t'_j \mid (t_i, t'_j) \in t \times t'\})$$

where \times is the set Cartesian product. Note that DTKP-AM does satisfy this condition, where the set union is implemented with an element-wise minimum. Now assuming $D(i)$ is initialized in the natural way (i.e. a tag consisting of a single proof containing just the input symbol i), then $D(i) \otimes D(i) = D(i)$ and therefore:

$$f(D, D)(\mathbb{T}) = \bigoplus_i D(i)$$

Under both and the add-mult approximation, the probability of \mathbb{T} is:

$$\sum_i \prod_j \text{norm}(D(i)_{ij}) = \sum_i D(i)_{ii} = \sum_i \text{Pr}(i) = 1$$

for any normalized D with at most k symbols. Even if the number of symbols exceeds k , we note that the distributions we seek to learn are often skewed (an accurate model should assign a probability to the ground truth that significantly outweighs the other symbols). For such distributions, DTKP and DTKP-AM would still yield the same probability for \mathbb{T} , and it is much closer to 1 than the sum of squares result from DAMP.

While this example may seem contrived, it still suggests the smaller role a “correct” WMC can have on the final answer compared to \oplus and \otimes implemented with proper set-based semantics. We even hypothesize that in most cases, the add-mult approximation does not meaningfully affect the final result compared to DTKP-WMC. This is empirically demonstrated by our benchmark results, which shows DTKP-AM achieving similar accuracy to Scallop’s implementation of DTKP-WMC. In fact, DAMP can be considered as a sort of ablation, where both the WMC and semiring operations use fuzzy add-mult semantics instead of a set-based one, and indeed, its accuracy often performs worse than both DTKP-WMC and DTKP-AM.

C.2 ROLE OF $+\infty$ AND $-\infty$

In this section, we motivate the use of $+\infty$ and $-\infty$ in DTKP-AM’s tensor representation of tags. Because tensors t are rectangular where every proof i and symbol j must have an entry t_{ij} , we require a way to denote the absence of an input symbol from a proof, and the absence of a proof from a tag. Importantly, an absent symbol should not influence the probability of a proof (i.e. its normalized value should contribute 1 to the probability’s product), and an absent proof should not influence the probability of a tag (i.e. it should contribute 0 to the sum during add-mult WMC). Indeed this is captured by our definition of norm, which clamps $+\infty$ to 1 (representing absent symbols) and $-\infty$ to 0 (representing absent proofs) during any probability calculation. While this introduces clamping operations, PyTorch’s implementation of clamp backpropagation ensures a gradient of 1 everywhere, even on the clamp boundaries (source: <https://github.com/pytorch/pytorch/pull/7049>).

Since $\hat{0}$ corresponds to the tag consisting of no proofs (i.e. a tag with probability 0), we initialize it to be a tensor where every proof is absent (all $-\infty$). Likewise, since $\hat{1}$ corresponds to the tag consisting of a single empty proof (i.e. a tag with probability 1), we initialize it to be a tensor where every symbol is absent from the first row / proof (all $+\infty$), while the remaining rows / proofs are absent (all $-\infty$).

C.3 FURTHER READING

For a more in-depth explanation of provenances in general, including the formalization of DTKP semantics with Boolean formulae, see Section 4 of Li et al. (2023). For worked examples of provenance computation with comparisons of top-k provenances to DAMP, we refer the reader to Scallop Language Group (2022).

D CONTROL FLOWS AND RECURSION IN DOLPHIN

Config	Time for UDF (s)	Time for Tag Computations (s)	Total Time (s)
No Parallelism	36.24 (C)	461.02 (C)	497.26
Parallelized Tag Computations	14.13 (C)	75.125 (G)	89.25

Table 3: Time taken by the symbolic program for the HWF task split by the time spent on the CPU and GPU. UDFs refer to user-defined functions where control flows reside for HWF. The times annotated with C and G indicate time spent on the CPU and GPU, respectively.

```

1 def compute_paths(paths, edges):
2     new_paths = ApplyIf(paths, edges, lambda p1, p2: (p1[0], p2[1]),
3     lambda p1, p2: p1[1] == p2[0])
4     merged = Union(paths, new_paths)
5     # checking for convergence via fix-point
6     if merged.symbols == paths.symbols:
7         return merged
8     else:
9         return compute_paths(merged, edges)
10 edges = Distribution(model(img), points)
11 paths = compute_paths(edges, edges)

```

Figure 10: Example of a transitive closure computation in DOLPHIN.

In this section, we provide a more detailed explanation of how DOLPHIN handles control flows and recursion. In DOLPHIN, control flows largely exist within the lambda functions supplied to the 'Apply', 'ApplyIf', and 'Filter' operations, which can be arbitrary Python functions over the symbols in the Distributions. As discussed in Section 3.2.2, these functions can include complex operations like if-then-else branches, loops, and even recursions. We do assume that divergent control flows are resolved within the lambda function itself. The nature of these functions means that they cannot be parallelized over the GPU. Instead, they are executed sequentially on the CPU, while the associated tags are computed parallelly on the GPU. We optimize the design of the Distribution class so that there is one set of CPU-based computations for the entire batch of samples rather than one set of computations for each sample, which is typical of other neurosymbolic frameworks. This allows DOLPHIN to maintain the benefits of parallelism even while the user-defined functions are executed sequentially.

D.1 CONTROL FLOW IN HWF

We demonstrate this by showing the time taken by the symbolic program for the HWF task split by the time spent on the CPU and GPU in Table 3. The first row shows the time taken when the neurosymbolic model is run sequentially on the CPU with no parallelism. The second row shows the time taken when tag computations are parallelized on the GPU over batches of 64 samples each. The times annotated with C and G indicate time spent on the CPU and GPU, respectively. We only show the time taken in the forward pass in the table.

Observe that the time, both for UDF computation and for Tag computation, decreases as we move from sequential CPU evaluation to the batched evaluation. Due to DOLPHIN's design, increases in batch size result in fewer CPU operations, since the set of CPU operations is shared for the entire batch, while parallelizing more tag computations over the entire batch.

D.2 RECURSION

In order to write recursive computations in DOLPHIN, one has two choices: either supply a recursive user-defined function to the DOLPHIN primitives, or write a more fine-grained program in Python that uses DOLPHIN primitives in the base case as well as the recursive case, set to terminate once

a condition is met. Here, the diverging control flows can be merged using the UNION primitive. We follow the latter approach for tasks involving recursion, such as Path and CLUTRR. The crux of those programs involves performing a transitive closure computation over a graph, represented by a set of edges for Path or relations for CLUTRR. We show an example of a transitive closure computation in Figure 10.

Here, let's say that `model` is a neural model that predicts the edges between each pair of points in a graph, represented by `points`. The `compute_paths` function computes the transitive closure of the graph by iteratively applying the edges to the paths. The `APPLYIF` function applies the edges to the paths if the end of the first path is the same as the start of the second path. The `UNION` function merges the new paths with the existing paths. The function `compute_paths` is called recursively until a fixpoint is reached, specifically until no new paths can be added. This is a simple example of a recursive computation in DOLPHIN, and also forms the core program needed for the PathFinder task. We perform a similar recursive computation for the CLUTRR task, where we find the transitive closure of a graph representing relations between people in a passage.

E COMPARISON WITH TENSOR-BASED NEUROSymbOLIC FRAMEWORKS

Systems like LYRICS (Marra et al., 2019), Logic Tensor Networks (LTNs) (Badreddine et al., 2022), and Tensorlog (Cohen et al., 2020) all have limited expressivity, which is one of the obstacles DOLPHIN aims to overcome. Specifically, they restrict the symbolic programs to first order logic and require users to specify low-level information such as how variables are grounded and what their domains are. They also restrict the symbols to be in the form of tensors and the user defined functions to consist of TensorFlow operations. These restrictions allow such systems to use TensorFlow to compile these programs into highly efficient computational graphs, but at the cost of expressivity. These frameworks also exclusively support simpler provenances and t-norms which are not sufficient for complex neurosymbolic programs.

On the other hand, DOLPHIN allows the user to track tags for specific symbols which can be arbitrary Pythonic objects. DOLPHIN programs further allow the user to manipulate Distributions over such symbols using arbitrarily complex code which may not necessarily translate to a computational graph. As such, there is a fine balance between the probabilistic computations, that happen over a GPU, and the symbolic computations, that take place on a CPU, all while maintaining a mapping between the two. This requirement sets a unique challenge addressed by DOLPHIN that we believe sets it apart from systems that use tensor operations for neurosymbolic learning. This fundamental design choice is also what allows DOLPHIN to be more expressive and flexible than existing systems. We also design DOLPHIN to be modular so that users can easily extend it to support new provenances and t-norms. As such, the t-norms used in LYRICS and LTN can be trivially added in a vectorized manner to DOLPHIN.

For instance, assume the case of MNIST Sum-2, where ‘model’ is the neural model. This is how it needs to be expressed in LTN:

```

956 1 ### Predicates
957 2 Digit = ltn.Predicate.FromLogits(model, activation_function="softmax")
958 3 ### Variables
959 4 d1 = ltn.Variable("digits1", range(10))
960 5 d2 = ltn.Variable("digits2", range(10))
961 6 ### Operators
962 7 Not = ltn Wrapper_Connective(ltn.fuzzy_ops.Not_Std())
963 8 And = ltn Wrapper_Connective(ltn.fuzzy_ops.And_Prod())
964 9 Or = ltn Wrapper_Connective(ltn.fuzzy_ops.Or_ProbSum())
965 10 Implies = ltn Wrapper_Connective(ltn.fuzzy_ops.Implies_Reichenbach())
966 11 Forall = ltn Wrapper_Quantifier(ltn.fuzzy_ops.Aggreg_pMeanError(),
967 12 semantics="forall")
968 13 Exists = ltn Wrapper_Quantifier(ltn.fuzzy_ops.Aggreg_pMean(), semantics="
969 14 exists")
970 15 # mask
971 16 add = ltn.Function.Lambda(lambda inputs: inputs[0]+inputs[1])
972 17 equals = ltn.Predicate.Lambda(lambda inputs: inputs[0] == inputs[1])
973 18

```

```

972 19 ### Axioms
973 20 @tf.function
974 21 def axioms(images_x, images_y, labels_z, p_schedule=tf.constant(2.)):
975 22     images_x = ltn.Variable("x", images_x)
976 23     images_y = ltn.Variable("y", images_y)
977 24     labels_z = ltn.Variable("z", labels_z)
978 25     axiom = Forall(
979 26         ltn.diag(images_x, images_y, labels_z),
980 27         Exists(
981 28             (d1, d2),
982 29             And(Digit([images_x, d1]), Digit([images_y, d2])),
983 30             mask>equals([add([d1, d2]), labels_z]),
984 31             p=p_schedule
985 32         ),
986 33         p=2
987 34     )
988 35     result_logits = axiom.tensor
989 36     return result_logits

```

Note that the FOL semantics of the Real Logic language used in LTN requires the user to specify the tracking of the probabilities with the symbols denoted by the ‘digits*’ variables.

On the other hand, DOLPHIN’s design allows the user to write the same program in a more intuitive way:

```

993 1 d1 = Distribution(model(img[0]), range(10))
994 2 d2 = Distribution(model(img[1]), range(10))
995 3
996 4 result_logits = GetProbs(Apply(d1, d2, lambda x, y: x + y))
997

```

If we consider the HWF task, where the neural model needs to predict both numbers and operators, DOLPHIN allows the user to write the following (naive) program:

```

1000
1001 1 symbols = [ str(i) for i in range(10) ] + [ '+', '-', '*', '/' ]
1002 2 res = Distribution(model(img[0]), symbols)
1003 3
1004 4 for i in range(1, expr_length):
1005 5     op = Distribution(model(img[i]), symbols)
1006 6     res = Apply(res, op, lambda x, y: x + y)
1007 7
1008 8 res = Apply(res, lambda expr: eval(expr))
1009 9 result_logits = GetProbs(res)

```

Writing the same program in LTN is not feasible due to the requirement of concatenating strings and evaluating the expressions they represent. The actual program for the HWF task in LTN would be much more complex, shown in the Appendix G

E.1 OPTIMIZING PROBABILISTIC COMPUTATIONS

Other works such as (Dang et al., 2021) and (Darwiche 2020), focus on solely on probabilistic computations rather than neurosymbolic frameworks. For instance, Juice (Dang et al., 2021) is a Julia package for logic and probabilistic circuits, which is not designed to be integrated with deep learning frameworks. On the other hand, Darwiche (2020) focuses on variable elimination with applications to optimize tensor-based computation. It will be interesting to see how DOLPHIN can be integrated with such systems to further improve the scalability and efficiency of neurosymbolic learning, and will include a discussion on this in the revised manuscript. However, we still believe that DOLPHIN’s novelty lies in its design that allows for the seamless integration of general purpose neurosymbolic programs within deep learning frameworks, which is not addressed by the existing systems.

N	B = 64		B = 256	
	Time per Epoch (s)	Accuracy	Time per Epoch (s)	Accuracy
4	11.42	0.96	8.92	0.97
8	12.55	0.95	9.15	0.95
16	27.45	0.94	15.71	0.89
20	36.59	0.92	18.73	0.85

Table 4: MNIST ProductN Training Epoch Times in Seconds.

F ON COMBINATORIAL EXPLOSIONS

The nature of the APPLY and APPLYIF primitives imply the possibility of combinatorial ballooning of computations in cases where either the number of symbols is large or where there are several distributions over which the function is applied. This is indeed a fundamental challenge in neurosymbolic frameworks as a whole. DOLPHIN mitigates this by leveraging the Distribution class, which condenses symbols into a single collection stored in CPU RAM while maintaining tags as a GPU tensor ($b \times N \times T$, where b is the batch size, N is the number of symbols and T is the shape of the tag). As shown in Figure 2b, this approach reduces symbolic overhead by avoiding redundant evaluations for each batch sample, unlike frameworks like Scallop, where each sample in a batch is independently evaluated. While tag evaluations still involve all combinations across all samples in a batch, they are computed in a vectorized manner on the GPU.

To see the effect of such computations even on larger experiments, we consider MNIST ProductN, where we multiply digits classified by the MNIST CNN as opposed to adding them in SumN. We show the per epoch training times in Table 4 for batch sizes of 64 and 256. In both cases, the DOLPHIN program is able to achieve high accuracies even for $N=20$ while running in reasonable amounts of time. The scaling gets even better for larger batch sizes (e.g. 256) since it increases the number of parallelized operations executed at any given time.

G THE HWF MODEL

We show the neurosymbolic model written in DOLPHIN for the HWF task along with the base neural model. In the HWF task, the neural model simply classifies each input image into 14 symbols: 10 digits and 4 operations.

```

1060 1 class SymbolNet(nn.Module):
1061 2     def __init__(self):
1062 3         super(SymbolNet, self).__init__()
1063 4         self.conv1 = nn.Conv2d(1, 32, 3, stride = 1, padding = 1)
1064 5         self.conv2 = nn.Conv2d(32, 64, 3, stride = 1, padding = 1)
1065 6         self.fc1 = nn.Linear(30976, 128)
1066 7         self.fc1_bn = nn.BatchNorm1d(128)
1067 8         self.fc2 = nn.Linear(128, 14)
1068 9
1069 10     def forward(self, x):
1070 11         x = self.conv1(x)
1071 12         x = F.relu(x)
1072 13         x = self.conv2(x)
1073 14         x = F.max_pool2d(x, 2)
1074 15         x = F.dropout(x, p=0.25, training=self.training)
1075 16         x = torch.flatten(x, 1)
1076 17         x = self.fc1(x)
1077 18         x = self.fc1_bn(x)
1078 19         x = F.relu(x)
1079 20         x = F.dropout(x, p=0.5, training=self.training)
1080 21         x = self.fc2(x)
1081 22         return F.softmax(x, dim=1)

```

This neural model is then used in the DOLPHIN program as follows:

```

1080 1 class HWFNet(nn.Module):
1081 2     def __init__(self):
1082 3         super(HWFNet, self).__init__()
1083 4
1084 5         # Symbol embedding
1085 6         self.symbol_cnn = SymbolNet()
1086 7         self.operators = [("+", ), ("-", ), ("*", ), ("/", )]
1087 8         self.symbols = [ (str(i),) for i in range(10)] + self.operators
1088 9
1089 10        self.db = torchql.Database()
1090 11
1091 12        def forward(self, img_seq, img_seq_len):
1092 13            batch_size, formula_length, _, _, _ = img_seq.shape
1093 14            length = [l.item() for l in img_seq_len]
1094 15
1095 16            inp = img_seq.flatten(start_dim=0, end_dim=1)
1096 17            symbol = self.symbol_cnn(inp).view(batch_size, -1, 14)
1097 18
1098 19            def eval_formula(s):
1099 20                try:
1100 21                    return eval("".join(s))
1101 22                except:
1102 23                    return math.nan
1103 24
1104 25            def concat_symbol(formula, symbol):
1105 26                if formula[-1] == "":
1106 27                    return formula
1107 28                else:
1108 29                    if not isinstance(symbol, tuple):
1109 30                        symbol = (symbol,)
1110 31                    formula += symbol
1111 32                    if len(formula) % 2 == 1 and len(formula) > 1:
1112 33                        if formula[-2] in ["*", "/"]:
1113 34                            eval_result = str(eval_formula(formula[-3:]))
1114 35                            formula = formula[:-3] + (eval_result,)
1115 36                        return formula
1116 37
1117 38            def infer_expression(length, *symbols):
1118 39                res = symbols[0]
1119 40                for i in range(1, len(symbols)):
1120 41                    res = Apply(res, symbols[i], concat_symbol)
1121 42                x = (Apply(res, eval_formula), )
1122 43                return x
1123 44
1124 45            def reorg(symbols, lengths):
1125 46                distrs = []
1126 47                for i in range(symbol.shape[1]):
1127 48                    if i < lengths:
1128 49                        distrs.append(Distribution(symbols[i, :].view(-1, 14), self.
1129 50symbols))
1130 51                        if i % 2 == 0:
1131 52                            distrs[-1] = distrs[-1].filter(lambda s : s not in self.
1132 53operators)
1133 54                        else:
1134 55                            distrs[-1] = distrs[-1].filter(lambda s : s in self.
1135 56operators)
1136 57                        else:
1137 58                            distrs.append(Distribution(torch.ones(1, device=device), [("("
1138 59), ), ]))
1139 60
1140 61            res = (lengths, *distrs)
1141            return res
1142
1143            q = torchql.Query("hwf", base="symbols").join("lengths") \
1144                .project(lambda symbols, lengths: reorg(symbols, lengths)) \

```

```

1134 62         .project(infer_expression, batch_size=batch_size)
1135 63
1136 64         res = q(db, tensors={"symbols": symbol, "lengths": length}, disable
1137 =True).rows
1138 65
1139 66         stacked = Distribution.stack(res)
1140 67         return GetProbs(stacked)

```

The `HWFNet` class is the neurosymbolic model. It takes in a sequence of images, `img_seq`, and their lengths, `img_seq_len`. Note that within a single batch there may be image sequences of varying lengths. The neural model, `symbolicnn`, is used to classify each image in the sequence into one of the 14 symbols. Since we know that each number in the expression is a single digit, the `reorg` function is used to filter out relevant symbols based on their position in the sequence (operators in even places, digits in odd places). This function also pads sequences of smaller lengths with empty strings, written as Distributions with a single element and a probability of 1. Once reorganized, the `infer_expression` function is used to infer the expression from the symbols. It does so by first concatenating Distributions using the `concat_symbol` function, which also performs partial evaluations whenever possible. Once all the symbols are concatenated, the expression is evaluated using the `eval_formula` function. The final expression is then returned as a Distribution. As a sidenote, while optional, we use the TorchQL (Naik et al., 2024) library to help write certain parts of the program. This shows the ease with which Distributions can be used with existing machine learning frameworks.

For such a complex DOLPHIN program, using a simple provenance like DAMP proves insufficient for longer sequences since the tags of all possible combinations of symbols are collated into a single number. On the other hand, DTKP-AM is able to track the top-k proofs for each symbol, pruning out the less probable proofs. Furthermore, since each proof is a collection of *input* symbols leading to a specific output, once the loss is calculated, gradients can be backpropogated directly to the input symbols that had the most influence on the output. On the other hand, the gradients may be distributed across all symbols in DAMP as it backpropogates through each intermediate computation regardless of their role in the computation of the output, resulting in slower convergence.

H ON THE LANGUAGE AND SEMANTICS

H.1 LANGUAGE

We designed DOLPHIN primitives to allow the expression of complex neurosymbolic programs in conjunction with user-defined functions. To develop the primitives, we studied several neurosymbolic tasks to determine the most common operations needed for these tasks. We found that the main operation needed in most programs is to apply a function to symbols from different input models and relations. This is primarily achieved via the join operation in Datalog, but we introduce the `Apply` or `ApplyIf` primitives for a more Pythonic approach. `Filters` are used to remove symbols violating conditions, similar to Datalog selections, while `Union` mimics the disjunction operation in Datalog, typically needed for writing recursive programs as described in Appendix D.

H.2 SEMANTICS

We designed DOLPHIN to be a general-purpose neurosymbolic framework able to support various semantics, as long as they can be expressed as operations over tags tracked via the Distribution class. DOLPHIN assumes that the provenance supplied to it offers both the conjunction and disjunction operations that operate over combinations of tags from input symbols, as well as a way to translate tags to probabilities. As long as these assumptions are satisfied, the primitives of DOLPHIN preserve the semantics offered by the provenances.

As such, supplying the DAMP provenance to the DOLPHIN program introduces basic fuzzy semantics which are preserved by DOLPHIN. However, there are cases where the independence assumptions may not hold and fuzzy semantics may not be appropriate.

The DTKP-AM provenance, on the other hand, offers an alternative without the assumption of variable independence, except on the input variables. At each step of the program, each symbol is associated with the tags of the input symbols that produce it via the proofs. Again, since DTKP-AM satisfies the aforementioned assumptions, the top-k semantics of the provenance are preserved.

These tags are then translated into probabilities by performing an add-mult operation over the proofs. This approximation of the WMC operation is more complex and results in a more precise translation of tags to probabilities. However, as we see in the experiments where Scallop uses DTKP-WMC, the accuracies achieved by DTKP-AM and DTKP-WMC are comparable.