∂SLASH : PROBABILISTIC CIRCUITS FOR FAST NEURAL ANSWER SET PROGRAMMING

Anonymous authors
Paper under double-blind review

ABSTRACT

Exact probabilistic inference is #P-hard and CPU-bound, and current neurosymbolic systems utilizing GPUs are approximate by design. Nonetheless, whether exact inference is cheap, can be *read off* the program. ∂SLASH compiles an answer-set program into a reasoning graph. It is a directed graph, implemented as a graph neural network and executed alongside deep neural networks on GPUs. Testing the graph for decomposability and determinism confirms it to be a probabilistic circuit. Hence, one message-passing sweep returns the exact query probability and the gradients of weighted model counting. Where the exactness fails, the offending disjunction goes to cutset sampling under a Hoeffding bound, which on visual question answering still beats every approximate baseline at every clause length, by up to 15 percentage points. Further, it surpasses the one second per epoch barrier on neurosymbolic benchmarks being order of magnitude faster than exact solvers. ∂SLASH sums 100 MNIST digits representing 10^{100} possible worlds, none of which are enumerated, at 99% digit accuracy on a single GPU.

1 INTRODUCTION

Deep Probabilistic Programming Languages (DPPLs) represent a powerful instantiation of neurosymbolic AI, combining high-level logical reasoning with low-level neural perception. By offering a declarative framework to integrate neural networks into logical theories, DPPLs such as DeepProbLog (Manhaeve et al., 2018), NeurASP (Yang et al., 2020), SLASH (Skryagin et al., 2022), and Scallop (Huang et al., 2021) allow models to learn from both data and background knowledge. A single obstacle has held this promise back: computing the probability of a query is #P-hard, and the machinery that computes it has historically run on the CPU. Consequently, perception and reasoning cannot execute as one pipeline, i.e., every gradient step stalls while a solver runs.

Two responses to that obstacle exist. Fuzzy approaches relax Boolean satisfiability into real-valued truth degrees combined by t-norms, which are differentiable, cheap, and map directly onto a GPU, but they discard probabilistic semantics, and the resulting gradients often fail to reflect the logic they were derived from (van Krieken et al., 2022). Probabilistic approaches keep the semantics and pay for it: they reduce learning to exact Weighted Model Counting (WMC), by knowledge compilation or by enumerating stable models, and inherit both its #P-hardness and its CPU-bound implementations. This leaves neurosymbolic AI with an apparent choice between rigour and speed.

Recent systems attack the speed side of that choice. Dolphin (Naik et al., 2025) keeps symbolic reasoning on the CPU and vectorises the probabilistic computation and gradient propagation around it; Lobster (Biberstein et al., 2026) compiles a Datalog-based language to an intermediate representation that executes end-to-end on the GPU through a library of provenance semirings. Both report substantial speed-ups, and both obtain them by approximating WMC. What is missing is not throughput but a statement of *when the fast answer equals the exact one*, that is, when the value returned is the query's true probability.

That question has a structural answer. A large majority of neurosymbolic programs satisfy Sato's independence and exclusiveness conditions, not by accident but by how their rules are written. Crucially, whether they do so is decidable by inspecting the program rather than by running it. We therefore translate an ASP program directly into a *reasoning graph*, a probabilistic reading of Answer Set Networks (Skryagin et al., 2024), and compute the query probability by message passing over it. This is not a way around knowledge compilation but a different route to what compilation

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

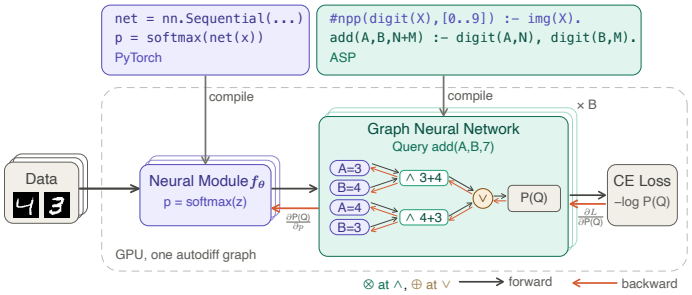


Figure 1: **∂SLASH is end-to-end differentiable.** The reasoning graph is compiled syntactically from the program into a GNN, and the neural module is instantiated from Python. Neural probabilities enter the graph's leaves via NPPs, and message passing yields $P(Q)$, encoding WMC differentiably. A single (batched) forward & backward path is all we need for end-to-end training.

produces: compilation *searches* for a decomposable, deterministic circuit, and that search is where its hardness lies. Moreover, our reasoning graph is read off the program syntactically, in time linear in the size of the ground program, making those same two properties something we *test* rather than construct. Because message passing is scatter–gather, the entire training loop, forward and backward, then runs natively on the GPU. After the reasoning graph construction ∂SLASH [1] tests these conditions.

Where these do hold, we prove the forward pass returns exactly the success probability, $P_{\Pi}(Q)$, and that backward propagation through it reproduces the gradients of exact WMC in ASP-based languages such as SLASH (Skryagin et al., 2022) and NeurASP (Yang et al., 2020). Where they fail, the same test routes the affected node to Monte Carlo sampling, whose sampling error is bounded on demand by Hoeffding's inequality rather than left unquantified. What this buys is measured in time rather than accuracy, and the theory says it must be: a method that computes the same quantity as exact WMC cannot be *more* accurate than the exact, solver-based DPPLs it reproduces. Matching them is the best result available to it. A large gain over them would cast doubt on the exactness claim rather than support it, so accuracy parity is the prediction confirmed, and the comparison that remains is cost. ∂SLASH reaches that accuracy in a fraction of the wall-clock time (Sec. 3); against approximate GPU systems, where no such ceiling applies, it is free to do better on both. The price is *memory* rather than time: what bounds the problems we can tackle is the amount of VRAM the grounded graph occupies, not the time spent reasoning over it. Sec. 3 measures where that ceiling lies.

In summary, we contribute:

- ∂SLASH , a DPPL whose forward and backward passes are fully differentiable and run natively on the GPU, with no solver in the training loop.
- a proof that under Sato's conditions the reasoning graph computes $P_{\Pi}(Q)$ *exactly*, and that its reverse-mode gradients coincide with those of exact WMC.
- a structural test deciding per node, before evaluation, whether these conditions hold, and, otherwise, a Monte Carlo fallback whose sampling error carries a Hoeffding certificate.
- evidence that exactness costs nothing in accuracy and little in time: ∂SLASH matches solver-based DPPLs across our benchmarks at a fraction of their runtime, and the limit that remains is one of *memory* rather than time. The grounded graph's footprint bounds the problems we reach, and we measure where that boundary lies.

2 GOING DIFFERENTIABLE WITH ∂SLASH

Neurosymbolic learning fits a network f_θ whose output $\mathbf{p} = f_\theta(x)$ is a distribution over atomic logical facts (e.g., an image's classes). A program Π encoding background knowledge then turns $\mathbf{p}$

[1] Implementation: `https://anonymous.4open.science/r/DifferentiableSLASH`

into the probability $P(Q)$ of a query it entails. The result is a pipeline $x \mapsto \mathbf{p} \mapsto P(Q)$ (Fig. 1) whose second mapping is the CPU-bound one.

We first introduce ∂SLASH and the reasoning graphs it builds to evaluate queries bottom-up, then show when that evaluation is exact, how the disjoint-sum problem is detected and paid for when it is not, and how the graph's gradients recover those of exact WMC in ASP-based DPPLs such as SLASH (Skryagin et al., 2022) and NeurASP (Yang et al., 2020). Proofs of all results in this section are in App. A. Here we state each result and the reason it holds. Where a result depends on a condition that can fail, we say so where it is used and point to where the failure is handled.

2.1 ∂SLASH PROGRAMS

Neural probabilistic predicates (NPPs) are the heart of ∂SLASH and act as a glue to bridge neural probabilities with the logic program. Manhaeve et al. (2018) introduced annotated disjunctions as conditional class probabilities. Skryagin et al. (2022) coined the term, generalized these to support other probabilistic queries, such as joint distributions. In ∂SLASH NPPs take the form:

$$\#npp\,(f(x), [c_1, \ldots, c_k]) \leftarrow \textit{Body}. \tag{1}$$

Here, $\#npp$ is a reserved keyword, f is the symbolic name of the NPP, x is the input term, and $c_1, \ldots, c_k$ denote the possible outcomes. The optional body allows for instantiating multiple NPPs during grounding.

A ∂SLASH program Π is thus defined as the union of standard ASP rules (Calimeri et al., 2020) and a finite set of NPPs. For the scope of this paper we exclude aggregates and minimization constraints, and restrict Π to *normal* rules, a single atom in the head, with constraints and classical negation. The substantive assumption is that NPPs are the *only* source of nondeterminism: fixing an outcome for every NPP, a *total choice* (Sec. 2.3), must leave exactly one stable model, which places total choices and stable models in bijection (Lem. A). This excludes choice rules $\{a_1; \ldots; a_k\} \leftarrow \textit{Body}$, though not their cardinality-one case $1\{a_1; \ldots; a_k\}1$, which is precisely an NPP with probabilities attached. For a different reason it also excludes disjunctive rules $a_1 \mid \cdots \mid a_k \leftarrow \textit{Body}$, which are not choice rules at all: their answer sets are subset-*minimal*. The graph's $\vee$-nodes are *definitional*, one per head atom, and must not be read as disjunctive heads (App. A.6).

2.2 REPRESENTING ∂SLASH PROGRAMS AS GRAPHS

To construct the reasoning graph we adopt the Answer Set Networks (ASN) compilation (Skryagin et al., 2024), but read node values as real-valued probabilities rather than Boolean truth values. The graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ gives each node v a probability $h_{v,t} \in [0, 1]$ at step $t \in \mathbb{N}$ and a type $\tau_v \in \{A_\wedge, A_\vee\}$, and each edge a weight $w_e \in \{+1, -1\}$; a negative edge propagates the complement $1 - h_{u,t}$, corresponding to default negation $(\text{not } a)$. We implement $\mathcal{G}$ as a graph neural network.

We first ground Π, then instantiate the truth node $\top$ over all facts, the false node $\bot$ receiving all constraints, and one probabilistic leaf per NPP outcome. Every rule with more than one body literal becomes a conjunction node ($\tau = A_\wedge$) fed by those literals, negated ones through negative edges; a single-literal body needs none. The head atom is itself the disjunction node ($\tau = A_\vee$) over the rules defining it, so several rules for one atom feed it directly. Classical negation needs no extra machinery: $-a$ compiles to a fresh atom with the constraint $\bot \leftarrow a, -a$. See Skryagin et al. (2024) for the full construction, and Fig. B in the appendix for worked examples.

Computing the query probability Message passing then propagates probabilities from the NPP leaves to the node v_Q representing the query, in the probabilistic semiring, which replaces the discrete logic operators $\wedge$ and $\vee$ used in ASN with $\otimes$ and $\oplus$: every node aggregates its incoming messages, $\otimes$ at conjunctions and $\oplus$ at disjunctions, and the iteration runs until the values stop changing, which App. A.2 states in full.

2.3 THE FORWARD PASS THROUGH THE REASONING-GRAPH IS $P_\Pi(Q)$

Scope. We assume throughout that Π is stratified, that distinct NPPs are independent, and that no two NPP outcomes overlap. Stratification governs *negation*; for *positive* recursion we require in addition that the ground ancestor cone of every query be acyclic. That is weaker than tightness,

since it constrains the grounding rather than the source text, and it is a property we check rather than assume (App. A.11, symbol table in App. A.1).

The forward pass is not a heuristic: under two structural conditions it computes the query probability *exactly*, recovering the distribution semantics of Sato & Kameya (2001) as a differentiable, parallel computation. Both conditions concern which neural facts a node ultimately rests on. An NPP *instance* is one grounded $\#npp$ rule, one random variable, collected in F. Its *outcome atoms* are the atoms $f{=}c$ (one per value e.g. the digits $0 \ldots 9$) represented by one leaf node each, collected in $\mathcal{A}_{\mathrm{npp}}$. We denote the instance an outcome atom a belongs to ($\mathrm{digit}(\mathrm{i1}, \ldots)$) as $\mathrm{inst}(a) \in F$. Writing $\mathrm{br}(v)$ for the branches feeding v, the *support* $\mathrm{supp}(v) \subseteq \mathcal{A}_{\mathrm{npp}}$ collects the outcome atoms v rests on, taken as $\bigcup_{b \in \mathrm{br}(v)} \mathrm{supp}(b)$ at conjunction and disjunction nodes (Def. 2). The first condition asks that a conjunction's branches not share a fact, $\mathrm{supp}(b_i) \cap \mathrm{supp}(b_j) = \varnothing$, *decomposability* (Def. 3). The second, the crux, asks that a disjunction's alternatives be mutually exclusive *as events*.

Definition 1 (Determinism)**.** An $A_\vee$-node with branches $b_1, \ldots, b_k$ is *deterministic* iff they are pairwise mutually exclusive as events, $P_\Pi(b_i \cap b_j) = 0$ for all $i \neq j$.

Determinism is exactly the *exclusiveness condition* of Sato & Kameya (2001), which licenses summing a disjunction's branches rather than taking a noisy-or. It holds by construction whenever a node's alternatives are keyed by distinct values of a single neural choice (e.g., MNIST-Addition's $\texttt{addition}(A, B, N)$), NPPs being in this respect precisely PRISM's multi-ary random switches, but it is an assumption about the program rather than a guarantee, and we return below to what happens when it fails.

Theorem 1 (Exactness)**.** *Let Π satisfy the scope assumptions above, and let Q be a query whose ancestor sub-DAG is decomposable at every $A_\wedge$-node (Def. 3) and deterministic at every $A_\vee$-node (Def. 1). Then the forward pass computes $p_{v_Q} = P_\Pi(Q)$.*

The proof (App. A) takes two steps over the *total choices* Ω_F: one outcome fixed per NPP, weighted by the product measure that independence supplies, and precisely the sample space of Sato & Kameya (2001). Lem. B shows the two conditions make the pass arithmetically exact, so every node's value is a single sum over Ω_F; Lem. A turns that into a sum over the stable models satisfying both program and query, which is $P_\Pi(Q)$.

Under these conditions the reasoning graph *is* a probabilistic circuit. Defs. 1–3 are literally the decomposability and determinism of probabilistic circuits (Choi et al., 2020), with the NPP leaves as input distributions, which is the syntax, and Thm. 1 is the matching semantic statement, that each node's value is the probability of the event it denotes. Syntax and semantics together are what the term means, so this is a consequence of Thm. 1 rather than an analogy drawn alongside it. The tractability follows: such a circuit admits linear-time marginals, so the sum over total choices *distributes through the graph* instead of being enumerated, marginalizing the total-choice axis without ever building it (Prop. 5). PRISM performs the weighted model count *explicitly*, enumerating explanations and summing them; the forward pass performs the same count *implicitly*, in $O(|\mathcal{V}|)$ parallel message-passing rounds that never materialise a model.

When determinism fails, we sample. Inclusion–exclusion would repair this exactly, at $2^k - 1$ terms, the #P-hardness of exact WMC surfacing inside one disjunction (App. C). We sample instead, and only where we must.

This is the pattern of *cutset sampling* (Bidyuk & Dechter, 2007; Rahman et al., 2019): draw a subset of the random variables chosen so that the conditioned remainder stays tractable, and keep exact inference everywhere else. Our cutset is the set of NPP instances feeding the nodes a structural test flag. That test reads the program rather than decomposing a fixed network, and errs only towards sampling, never away from it (Alg. 1, Rem. 4, App. A.7). The circuit's exact marginalisation still applies at every other node. At a flagged node the sampled branches are combined by $\max$ rather than by a sum, and that one substitution removes the overcounting: two branches that hold together contribute $\max(1, 1) = 1$ rather than $1 + 1 = 2$, so overlapping explanations are counted once by construction and exclusivity is never needed.

Remark 1 (What the forward pass returns). Where Alg. 1 routes a node to sampling, the value returned differs from $P_\Pi(Q)$ by $\eta = \eta_{\mathrm{sampling}} + \eta_{\mathrm{bias}}$. The first is *certified*: the user names an accuracy ε and a failure probability δ, and since the estimator averages i.i.d. values in $[0, 1]$, Ho-

effding's inequality turns that request into a sample count $n_{\mathrm{mc}} \geq \ln(2/\delta)/(2\varepsilon^2)$. The second is the bias of the relaxation that keeps the pass differentiable: it is *controlled* by the temperature τ and vanishes as $\tau \to 0$, but we give it no closed-form bound at fixed τ (App. A.8). We therefore do not claim that Def. 1 always holds, because it does not, nor that the *total* deviation is certified, but that **the sampling error is certified on demand, and the relaxation bias is the single remaining uncontrolled term**.

What this buys over the distribution semantics. Sato & Kameya (2001) ask three things of a program: a sample space of total choices, a summation over a query's explanations, and the *exclusiveness condition* making that summation a plain sum. Here Ω_F is that sample space, Lem. A makes its correspondence with Π's stable models a theorem about splitting rather than a modeling convention, and the summation becomes the circuit's forward pass. Exclusiveness is where the two accounts diverge: PRISM's Modeling Principle places the obligation on the *author*, who receives no warning on failing it, whereas Alg. 1 moves it to the *system*, which detects the failure structurally and returns a certified value instead. We do not weaken the distribution semantics; we discharge its one unchecked precondition, and report η when we cannot.

2.4 Learning in ∂SLASH

Following the principle of learning from entailment (Manhaeve et al., 2018), we optimize the neural parameters θ to maximize the success probability $P(Q)$ for a set of queries Q. We do this by minimizing the cross-entropy loss L between $P(Q)$ and the ground-truth label y, which is 1 for an observed query:

$$L = -[\, y \cdot \log P(Q) + (1-y) \cdot \log(1 - P(Q))\,] = -\log(P(Q)). \tag{2}$$

Beyond maximizing $P(Q)$, we must satisfy the program's constraints, which all connect to a probabilistic false sink $\bot$: a value of 0 there means every constraint is fulfilled, 1 that at least one is fully violated. Because a query is itself encoded as a constraint reaching $\bot$ through a negative edge, we maximize $P(Q)$ indirectly by minimizing $P(\bot)$:

$$L = -1 \cdot \log(1 - P(\bot)). \tag{3}$$

If no other constraints exist besides the query, the sink probability is strictly $P(\bot) = 1 - P(Q)$, natively reducing Eq. 3 to standard cross-entropy as in Eq. 2.

2.5 Backward pass through the reasoning-graph recovers SLASH gradients

Minimising Eq. 2 needs $\partial L/\partial p_c$ for each NPP outcome probability $p_c = P_\Pi(f{=}c)$, and Thm. 1 has already settled what is being differentiated: the value at v_Q *is* $P_\Pi(Q)$. What remains is to ask what its derivative is, and whether that derivative agrees with the one exact WMC prescribes. We show that it does, and more cheaply than the WMC formulation suggests.

Exact WMC states that gradient as sums over stable models. Two such sums matter here: the *reward* α, collecting the models that support both the query and the outcome $f{=}c$, and the *normalisation* γ, collecting those that support the query at all,

$$\alpha = \sum_{\substack{I \models \Pi,\ I \models Q \\ I \models f=c}} \frac{P_\Pi(I)}{P_\Pi(f{=}c)}, \qquad \gamma = \sum_{I \models \Pi,\ I \models Q} P_\Pi(I). \tag{4}$$

Both range over precisely the enumeration Thm. 1 lets us avoid, and our forward pass differentiates to their ratio.

Proposition 2 (The graph gradient is reward over normalisation)**.** *Let Q satisfy the hypotheses of Thm. 1 and let $p_c = P_\Pi(f{=}c)$. Then reverse-mode differentiation of the forward pass computes $\partial \log p_{v_Q}/\partial p_c = \alpha/\gamma$, with α and γ as in Eq. 4.*

The proof (App. A.13) is one step of calculus on Thm. 1: that theorem makes p_{v_Q} a sum of products in which p_c occurs at most once, so differentiating leaves exactly α over the denominator γ, and no

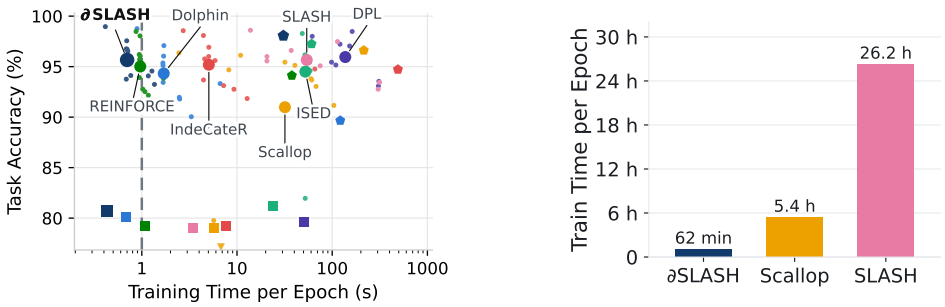


Figure 2: **∂SLASH breaks the 1 second train epoch barrier on many tasks.** (Left) Task accuracy against wall-clock time per epoch; small dots are individual tasks of MNIST-R, the big dot its mean. Leaf classification is a square and HWF a pentagon. DPL/SLASH timed out on 2 MNIST-R tasks and HWF. (Right) Time per epoch on VQAR (50k C_{all}), where training is reduced to an hour.

model is ever enumerated. Skryagin et al. (2022, Eq. 15) define their gradient from α and γ together with a third term penalising the competing outcomes of the same NPP; that term requires no separate computation here, and App. A.13 relates the two definitions in full.

This closes the argument: the reward α counts each model once only because the disjunctions are deterministic, so the very condition that makes the forward pass exact is what makes its gradient correct. Exactness, tractability and the gradient are three consequences of one structural property, and ∂SLASH obtains all three without ever calling a solver.

3 Experiments

Our contributions are a forward pass that computes P(Q) exactly whenever the program satisfies the conditions of Sec. 2.3, and a fallback to Monte-Carlo sampling for the nodes where it does not. The valuation in this section validates both and addresses two practical questions: whether our introduced GPU-based training regime resolves the runtime bottleneck of CPU-based solvers, and how different program encodings affect the resulting Reasoning. Concretely, we ask:

- **RQ1 (Exact regime)**: On tasks whose programs satisfy Sato's conditions, does ∂SLASH match the accuracy of existing exact DPPLs and approximate NeSy methods?
- **RQ2 (Sampling)** For problems where the exactness conditions do not hold, how does Monte-Carlo sampling as a fallback compare against other approximate methods?
- **RQ3 (Speed)** Does replacing the CPU-bound symbolic solving with GPU-based RG close the runtime gap between NeSy training and purely neural training?
- **RQ4 (Encoding)** How do variations of a task's program encoding (cartesian, chain, or tree) affect graph size and the scale of problems ∂SLASH can solve?

3.1 The exact regime: matching solver-based DPPLs in seconds

To answer RQ1 we follow the work of Manhaeve et al. (2018); Solko-Breslin et al. (2024), which covers a wide collection of varying tasks each fulfilling the nesessary conditions for exactness (see App. D for an analysis):

MNIST-R Li et al. (2023) introduces different flavors of the MNIST-addition task (totaling **11 tasks**) from Manhaeve et al. (2018). In the original MNIST addition task, a model must predict the sum of two MNIST images (LeCun et al., 1998) and, during training, receives only indirect feedback from the sum. MNIST-R extends this to different operators, such as *mult*, *mod*, and *compare*.

Hand-written formula. While operators combining the digits are explicitly provided in the MNIST-R tasks, the hand-written formula (HWF) task introduced by Li et al. (2020) requires the model to simultaneously learn to classify the digits (0-9) and identify the hidden operators ($+$, $-$, $\times$, $\div$), relying solely on the final expression result. Formulas are constructed of 1-7 elements.

Test ↓	Scallop	SAME	∂SLASH
1k C_2	84.34	79.26	84.60
1k C_3	81.97	72.95	85.00
1k C_4	80.90	69.50	85.80
1k C_5	83.76	69.33	87.50
1k C_6	82.61	63.18	86.80
5k C_{all}	82.72	70.90	85.94

	Average Precision		Time	
	Shapeworld4	CLEVR	Shapeworld4	CLEVR
Slot Att.	85.25	90.21	7.6s	1m:35s
SLASH	94.98	93.47	1m:37s	9m:31s
SAME	95.21	92.50	53.8s	7m:27s
∂SLASH	95.29	94.12	7.7s	1m:18s

Table 1: **Exact gradients beat approximate ones, at no cost in time. Left:** recall@5 on VQAR, trained on 50k C_{all} queries and tested on all clause lengths; the 10k C_2 regime is in App. E.2. **Right:** object-centric learning: average precision (higher is better) and wall-clock time (lower is better). Each column is shaded on its own scale, so colours compare down a column and not across.

Leaf classification. Further, in Leaf classification (Solko-Breslin et al., 2024), an image depicting a leaf is classified via a decision tree based on the features of margin, shape, and texture, obtained by a neural network. During training, only the class is provided, and gradients are obtained by backpropagating through possible features in the decision tree.

Models. For comparison, we choose the following representative DPPLs: DeepProbLog (DPL) (Manhaeve et al., 2018) and SLASH (Skryagin et al., 2022) for exact DPPLs. Dolphin (Naik et al., 2025), and Scallop (Huang et al., 2021) as approximate DPPLs living on the GPU and CPU, respectively. REINFORCE (Williams, 1992), IndeCateR (Smet et al., 2023), and ISED (Solko-Breslin et al., 2024) as execution sampling methods, which skip DPPL inference and estimate gradients by sampling NPP outcomes and executing the program on those samples. This makes them fast methods for comparison. Gradients estimated from samples of the entire outcome space have high variance, though, so REINFORCE and IndeCateR receive a higher epoch budget to converge (App. E).

Fig. 2 (left) shows the time taken for one epoch in relation to the achieved accuracy. We can see that since the introduction of DPL, the time taken has been reduced by many orders of magnitude, resulting in ∂SLASH breaking the 1-second boundary for a single training epoch. **Answering RQ1, we can see that ∂SLASH matches the performance of the best-performing exact DPPLs while also positively answering RQ3.**

3.2 Sampling beyond the exact regime for VQA

The VQAR benchmark asks a model to answer questions about objects in a scene, utilizing facts and rules encoded in a knowledge graph; the scences and their annotation come from GQA (Hudson & Manning, 2019) The task was introduced in Huang et al. (2021) and tested by Scallop, an approximate DPPL based on datalog that considers top-*k* proofs. We instead encode the entire knowledge graph as a shared base graph (1.35M nodes, Tab. 3) and attach the nodes of each query, a conjunction of 2–6 predicates, at training and inference time.

The base graph satisfies the exactness conditions, but 34.8% of attached queries introduce overlapping branches, mostly through `relation(A,B,R)` atoms. Further, we note that with increasing clause length, relations appear more often, increasing the need for sampling (See Tab. 3 in App. D). We compute these queries exactly wherever possible and fall back to sampling at the violating nodes.

We follow the training regime of Huang et al. (2021): training on 50k queries of all clause lengths (C_{all}), comparing ∂SLASH with Scallop and SLASH with SAME, an approximate variant which prunes NPP outcomes via top-k% (Skryagin et al., 2023), and report recall@5 in Tab. 1. It outperforms SAME by 15.0 points and Scallop by 3.2 points across all clause lengths. Furthermore, it handles relations robustly compared with SAME, which iteratively adds relations to the ASP solver until models are found. The same results hold for the VQAR generalization test from training on (C_2), for which results are shown in Sec. E.2 in the App. All in all, the increase in violations of the exactness conditions seen in longer clauses does not affect the recall, proving sampling as an effective strategy to minimize the loss of analytical exactness (Thm 1), answering RQ3 successfully.

N	Digit acc. (%)	Task acc. (%)
10	99.03	92.34
20	99.09	86.20
50	99.12	68.90
100	98.97	45.20

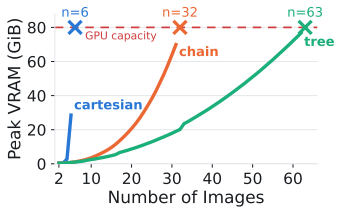


Figure 3: **Scaling to a googol of possible worlds.** (Left) MNIST-Addition over N images, tree encoding, mean over 5 seeds. Perception does not degrade, only the chance of getting all N digits right at once. Full results in App. E.4. (Right) Peak memory against N for the three $\texttt{sum}_N$ encodings on one 80 GiB A100. The encoding alone moves the ceiling from N=6 (cartesian) to N=32 (chain) to N=63 (tree).

3.3 Removing the CPU bottleneck on Object Centric Learning

In Object-Centric learning, the goal is to separate objects into slots that can be used for downstream tasks such as set prediction or object discovery (Locatello et al., 2020). Here, we focus on the set prediction task, in which n objects must be assigned to n slots. For this Locatello et al. (2020) relies on Hungarian matching to find the most likely slot-object assignment. Skryagin et al. (2022) used EiNets (Peharz et al., 2020) as NPPs within SLASH that model concepts for each slot and to constrain them with ground-truth objects, thereby improving convergence and recall. While beneficial for performance, their DPPL introduced substantial symbolic overhead relative to the fast neural component. Tab. 1 (right) mirrors that as the average time per epoch compounds due to slow multi-threaded solver call. **On the other hand, ∂SLASH 's tight integration of symbolic and neural layers, performed entirely in a single forward and backward pass, eliminates the whole overhead while preserving the performance benefits (RQ3).**

3.4 Googol MNIST Addition

Summing N MNIST images asks the program to sum over 10^N digit assignments. At N=100 that is a googol, and ∂SLASH trains on it, at 99% digit accuracy and three minutes an epoch, on a single GPU. Prop. 5 is why: the total-choice axis is marginalised, never built.

How the program is written decides whether that scale is reachable at all. The same addition admits three encodings (cartesian, chain and tree) that are semantically identical and structurally nothing alike, and they run out at N=6, N=32 and N=63 (Fig. 3). A tenfold difference in reachable problem size, from rewriting the program. All three encodings compute the same number and differ only in how they bracket the sum, which associativity leaves free. Take four images 2 3 5 7: the *cartesian* encoding brackets nothing and enumerates every assignment as its own conjunction, so the graph grows with 10^N. The *chain* encoding brackets left to right, ((2 + 3) + 5) + 7, reusing each partial sum to keep the graph linear in N, but N deep. The *tree* encoding brackets in pairs, (2 + 3) + (5 + 7), reusing the same partial sums at depth $\lceil \log_2 N \rceil$, which is what makes N=100 reachable at all.

Two more ingredients are not optional. In decimal space, the product over N probabilities underflows and the gradient vanishes with it, so every operator runs in log space. Further, a long sum is satisfied by many wrong digit assignments, so without a confidence penalty $-\lambda H(\mathbf{p})$ on the NPP outputs (Pereyra et al., 2017) the network settles into confident errors that cancel to the right total. Without them, at λ=0 and N=20, task accuracy falls from 86.2 to 28.7 and individual seeds collapse outright. App. E.4 derives both.

As Fig. 3 shows, the scalability stands and fails with the amount of available VRAM. To push beyond $N = 63$, we use gradient checkpointing (Chen et al., 2016). With all the buildings blocks, we googol MNIST Addition, compiling and training with the reasoning graph of 417k nodes and 1.2M edges. Perception does not degrade: what falls is the chance of getting all N digits right at once, not the digits themselves. This answers the RQ4 in full.

With all four research questions answered, the designed framework transforms the reasoning modules into an additional layer, compatible with deep neural networks, that can be executed on the same GPU in an end-to-end fashion without compromising accuracy.

4 Related works

Exact probabilistic DPPLs. DeepProbLog (Manhaeve et al., 2018) let neural networks parameterise the probabilistic facts of ProbLog. NeurASP (Yang et al., 2020) and SLASH (Skryagin et al., 2022) later carried the idea into Answer Set Programming, gaining non-monotonic reasoning, hard constraints, and the solver machinery built for them (Gebser et al., 2019). All three obtain gradients from exact inference and inherit its cost. Whether by compilation into a tractable circuit (Darwiche, 2003) or by enumerating stable models, inference is #P-hard in general and CPU-bound in practice.

Approximate and fuzzy alternatives. A second line of work approximates. Scallop (Huang et al., 2021; Li et al., 2023) retains only top-k proofs, while ISED (Solko-Breslin et al., 2024) and IndeCateR (Smet et al., 2023) sample executions, the latter through an unbiased categorical estimator with lower variance than REINFORCE. A third line abandons probability altogether, relaxing the connectives into t-norms over $[0, 1]$ so that logic becomes differentiable by construction and maps naturally onto GPUs. Logic Tensor Networks (Badreddine et al., 2022) are its fullest expression. Both lines reach programs the exact systems cannot, but neither returns the quantity the semantics asks for, and fuzzy gradients need not agree with the logic they relax (van Krieken et al., 2022).

Moving the hardware instead. Most recently, the CPU bottleneck has been attacked directly: Answer Set Networks (Skryagin et al., 2024) evaluate an ASP program by message passing on a graph; Dolphin (Naik et al., 2025) vectorizes the probabilistic computation around CPU-resident symbols; Lobster (Biberstein et al., 2026) compiles Scallop programs to a GPU intermediate language; REASON (Wan et al., 2026) offloads inference to a reconfigurable accelerator; and KLay (Maene et al., 2025) re-lays an already-compiled arithmetic circuit for GPU evaluation. Only KLay keeps exactness. Dolphin, for instance, replaces exact WMC with a vectorized surrogate because the exact procedure is difficult to vectorize (Naik et al., 2025)

Where ∂SLASH sits. We keep the exact semantics of the first group and the GPU execution of the last, and make the conditions under which they coincide explicit and checkable: the reasoning graph is a probabilistic circuit (Choi et al., 2020) whose decomposability and determinism are not searched for, as in knowledge compilation, but tested. The contrast with KLay is the sharpest. It parallelises the evaluation of a compiled circuit, amortising what it calls the "initial expensive knowledge compilation step" and timing that evaluation alone, whereas ∂SLASH has no such step, the graph being read off the ground program in time linear in its size. The two are complementary: KLay accelerates a circuit ∂SLASH never builds.

5 Conclusions and Future Work

∂SLASH makes perception and reasoning one GPU-resident computation: exact where a structural test says exactness is available, Hoeffding-certified where it is not. The test is what does the work: decided on the program, free at run time, and enough to let a message-passing sweep stand in for weighted model counting. The experiments confirm this. In the exact regime, ∂SLASH matches solver-based DPPLs in a fraction of their time. On VQAR, where sampling is required, it outperforms approximate systems. And it scales to problems whose possible worlds could never be enumerated, because the graph never enumerates them.

Magic Sets (Alviano et al., 2012) is the most direct purchase on that, being query-directed and answer-set preserving. It narrows the ground program while Thm. 1 still applies, and the forward pass already reads only a query's ancestor cone, which is what the rewriting would build. Past that, message passing is scatter–gather, the primitive distributed GNN systems partition across devices, so an oversized graph becomes a partitioning problem and not a wall. Orthogonally, the fragment of Sec. 2.1 leaves out choice rules, head disjunction, aggregates and optimisation; full ASP-Core-2, with a bound on the relaxation bias η_{bias}, is the natural completion.

AI USE STATEMENT

We used generative AI tools to assist with writing, coding, and literature exploration. All content of the paper was drafted by the authors. AI tools were then used for rephrasing, correcting typos, and condensing text, as well as for feedback on the draft alongside feedback from human colleagues. AI-suggested literature was used only as a starting point, and all cited works were read and verified by the authors. AI tools were also used for coding assistance, and all generated code was carefully reviewed and tested by the authors before use. The authors take full responsibility for the content of this work, including text and artifacts produced with the help of generative AI.

REPRODUCIBILITY STATEMENT

We took several steps to ensure that the results in this work are reproducible. Implementation details, hyperparameters and hardware are collected in Appendix E, together with the extended tables reporting means and standard deviations. An anonymized version of the code is available (`https://anonymous.4open.science/r/DifferentiableSLASH`). The metrics are the standard ones for each benchmark and are defined where they are first reported. All datasets used here are publicly available: MNIST (LeCun et al., 1998) and the MNIST-R task suite (Li et al., 2023), the hand-written-formula benchmark (Li et al., 2020), leaf classification (Solko-Breslin et al., 2024), Shapeworld4 and CLEVR for object-centric set prediction (Locatello et al., 2020), and VQAR (Huang et al., 2021). No perception module is pretrained: the convolutional encoders for the digit and symbol NPPs, and Slot Attention (Locatello et al., 2020) with EiNets (Peharz et al., 2020) for the object-centric tasks, are trained from scratch in every reported run. Every experiment runs on a single GPU, an A100 with 40 GB except for Googol MNIST Addition, which requires the 80 GB card; five seeds per run other than VQAR.

REFERENCES

Mario Alviano, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic sets for disjunctive Datalog programs. *Artificial Intelligence*, 2012.

Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In Jack Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, pp. 89–148. Morgan Kaufmann, 1988. doi: 10.1016/b978-0-934613-40-8.50006-3.

Samy Badreddine, Artur S. d'Avila Garcez, Luciano Serafini, and Michael Spranger. Logic Tensor Networks. *Artificial Intelligence*, 2022.

Paul Biberstein, Ziyang Li, Joseph Devietti, and Mayur Naik. Lobster: A gpu-accelerated framework for neurosymbolic programming. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2026.

Bozhena Bidyuk and Rina Dechter. Cutset sampling for Bayesian networks. *Journal of Artificial Intelligence Research*, 2007.

Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. ASP-Core-2 input language format. *Theory and Practice of Logic Programming*, 2020.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv:1604.06174*, 2016.

YooJung Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic Circuits: A unifying framework for tractable probabilistic models, 2020. URL `http://starai.cs.ucla.edu/papers/ProbCirc20.pdf`.

Keith L. Clark. Negation as failure. In *Logic and Databases*, 1978.

Adnan Darwiche. A differential approach to inference in Bayesian networks. *Journal of the ACM*, 2003.

Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 1995.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 2019.

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

Drew A. Hudson and Christopher D. Manning. GQA: A new dataset for real-world visual reasoning and compositional question answering. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6700–6709, 2019.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with Gumbel-Softmax. In *5th International Conference on Learning Representations*, 2017.

Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C. Lawrence Zitnick, and Ross B. Girshick. CLEVR: A diagnostic dataset for compositional language and elementary visual reasoning. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1988–1997, 2017.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

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 *Proceedings of the 37th International Conference on Machine Learning*, 2020.

Ziyang Li, Jiani Huang, and Mayur Naik. Scallop: A language for neurosymbolic programming. *Proceedings of the ACM on Programming Languages*, 2023.

Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *Proceedings of the Eleventh International Conference on Logic Programming*, 1994.

Francesco Locatello, Dirk Weissenborn, Thomas Unterthiner, Aravindh Mahendran, Georg Heigold, Jakob Uszkoreit, Alexey Dosovitskiy, and Thomas Kipf. Object-centric learning with slot attention. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems*, 2020.

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *5th International Conference on Learning Representations*, 2017.

Jaron Maene, Vincent Derkinderen, and Pedro Zuidberg Dos Martires. KLay: Accelerating arithmetic circuits for neurosymbolic AI. In *Thirteenth International Conference on Learning Representations*, 2025.

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. DeepProbLog: Neural probabilistic logic programming. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems*, 2018.

Aaditya Naik, Jason Liu, Claire Wang, Amish Sethi, Saikat Dutta, Mayur Naik, and Eric Wong. DOLPHIN: A programmable framework for scalable neurosymbolic learning. In *Forty-second International Conference on Machine Learning*, 2025.

Robert Peharz, Steven Lang, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Guy Van den Broeck, Kristian Kersting, and Zoubin Ghahramani. Einsum networks: Fast and scalable learning of tractable probabilistic circuits. In *Proceedings of the 37th International Conference on Machine Learning*, 2020.

Gabriel Pereyra, George Tucker, Jan Chorowski, Lukasz Kaiser, and Geoffrey E. Hinton. Regularizing neural networks by penalizing confident output distributions. In *5th International Conference on Learning Representations, Workshop Track*, 2017.

Tahrima Rahman, Shasha Jin, and Vibhav Gogate. Cutset Bayesian networks: A new representation for learning rao-blackwellised graphical models. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, 2019.

Taisuke Sato and Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 2001.

Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 2002.

Arseny Skryagin, Wolfgang Stammer, Daniel Ochs, Devendra Singh Dhami, and Kristian Kersting. Neural-probabilistic answer set programming. In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning*, 2022.

Arseny Skryagin, Daniel Ochs, Devendra Singh Dhami, and Kristian Kersting. Scalable neural-probabilistic answer set programming. *Journal of Artificial Intelligence Research*, 2023.

Arseny Skryagin, Daniel Ochs, Phillip Deibert, Simon Kohaut, Devendra Singh Dhami, and Kristian Kersting. Answer set networks: Casting answer set programming into deep learning. *arXiv:2412.14814*, 2024.

Lennert De Smet, Emanuele Sansone, and Pedro Zuidberg Dos Martires. Differentiable sampling of categorical distributions using the CatLog-derivative trick. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems*, 2023.

Alaia Solko-Breslin, Seewon Choi, Ziyang Li, Neelay Velingker, Rajeev Alur, Mayur Naik, and Eric Wong. Data-efficient learning with neural programs. In *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems*, 2024.

Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 1955.

Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 1976.

Emile van Krieken, Erman Acar, and Frank van Harmelen. Analyzing differentiable fuzzy logic operators. *Artificial Intelligence*, 2022.

Zishen Wan, Che-Kai Liu, Jiayi Qian, Hanchen Yang, Arijit Raychowdhury, and Tushar Krishna. REASON: accelerating probabilistic logical reasoning for scalable neuro-symbolic intelligence. In *IEEE International Symposium on High Performance Computer Architecture*, 2026.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 1992.

Zhun Yang, Adam Ishay, and Joohyung Lee. NeurASP: Embracing neural networks into answer set programming. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, 2020.

READING GUIDE TO THE APPENDIX

The main text carries the claims and this appendix carries the record, so each section below answers one question Sec. 2.3– 3.4 raise and do not settle. Every section states at its head which claim it discharges.

App. A proves the results of the main paper's Sec. 2.3 and 2.5: the exactness theorem, the two lemmas composing it, the conditions it quietly requires, and the gradient identity. It is the section on which the paper's correctness rests. App. B shows three reasoning graphs the system itself rendered, so the construction of Sec. 2.2 and the test of Alg. 1 can be read off real output rather than an idealisation. App. C says why an overlapping disjunction is sampled rather than repaired exactly, a choice Sec. 2.3 makes and does not defend. App. D measures how often the conditions of Thm. 1 hold across the benchmarks of Sec. 3, which is what makes the exact regime a statement about practice rather than about a fragment. App. E.2 gives both VQAR training regimes where Sec. 3.2 reports one. App. E.4 supplies the three ingredients Sec. 3.4 names without deriving: the encodings, log space, and the confidence penalty. App. F lists the three $\texttt{sum}_N$ programs in full.

A PROOFS

Here we show the full proofs for the four main results: Thm. 1 and the two lemmas it composes (Sec. 2.3), and the gradient identity of Prop. 2 (Sec. 2.5). Sec. A.1–A.3 fix notation and restate the forward pass in full; the two propositions that follow establish that "propagate to a fixpoint" is a well-defined operation which terminates on the *least* fixed point; Sec. A.9 and A.10 are the structural and arithmetic halves of Thm. 1; and the closing three sections give the condition the theorem quietly requires, the identity it implies, and the gradient. A reader interested only in the comparison with Skryagin et al. (2022) can go directly to Sec. A.13. Everything Sec. 2.3 claims about exactness, and everything Sec. 2.5 claims about the gradient, is discharged here and nowhere else.

A.1 SYMBOLS, AND THEIR CORRESPONDENCE TO PRIOR WORK

∂SLASH inherits notation from two lines of work that do not agree with one another: the reasoning graph from Answer Set Networks (Skryagin et al., 2024), and the neural-probabilistic predicates and gradient from Skryagin et al. (2022). Where the two collide we keep the graph side unchanged, since it carries the bulk of the development, and adopt for the probabilistic side the symbols of the NPP *declaration* (Eq. 1) rather than those of the gradient.

here	role	(Skryagin et al., 2024)	(Skryagin et al., 2022)
v, u	node; source of an edge (u, v)	v, u	—
w_e, $h_{v,t}$	edge weight; node value at round t	w_e, $h_{v,t}$	—
$b_1, \ldots, b_k$	branches of an $A_\wedge/A_\vee$-node	—	—
f	a neural-probabilistic predicate	—	c
c	one outcome of f	—	v
p_c	probability of outcome c	—	$P_{\Pi(\theta)}(c{=}v)$
p_v	value at node v	—	—
ω, Ω_F	total choice; the space of them	—	—
α, β, γ	reward, penalty, normalisation	—	α, β, γ

Both f and c are taken from Eq. 1's own $\texttt{npp}(f(x), [c_1, \ldots, c_k])$, so the only departure from Skryagin et al. (2022) is that their $c{=}v$ becomes our $f{=}c$.

A.2 THE FORWARD-PASS UPDATE RULE

Every node aggregates its incoming messages in the probabilistic semiring, negative edges propagating the complement $1 - h_{u,t}$:

$$\begin{cases} \bigotimes_{\substack{e=(u,v)\in\mathcal{E},\\ w_e=1}} h_{u,t} \otimes \bigotimes_{\substack{e=(u,v)\in\mathcal{E},\\ w_e=-1}} 1 - h_{u,t} & \text{if } \tau_v = A_\wedge, \\ \bigoplus_{\substack{e=(u,v)\in\mathcal{E},\\ w_e=1}} h_{u,t} \oplus \bigoplus_{\substack{e=(u,v)\in\mathcal{E},\\ w_e=-1}} 1 - h_{u,t} & \text{if } \tau_v = A_\vee. \end{cases} \tag{5}$$

The number of steps t depends on the depth of the graph, so in practice we check for DAG convergence by comparing the current values with those at $t - 1$.

A.3 NOTATION

Fix a total choice $\omega \in \Omega_F$ and write Π_ω for the ground program with every NPP atom fixed by ω. Partition the graph's nodes as $V = V_{\text{npp}} \cup V_{\text{rest}}$. Each NPP node tests one outcome of one NPP, so write $v_{f=c}$ for the node asserting that NPP f takes value c. Held constant across all rounds are $h_t(\top) = 1$ and $h_t(v_{f=c}) = \mathbb{1}[\omega(f) = c]$: once ω is fixed the neural choices are plain facts, constants of the iteration rather than variables of it. Let $T^\omega_{\mathcal{G}} : \{0,1\}^{V_{\text{rest}}} \to \{0,1\}^{V_{\text{rest}}}$ be the synchronous update of Eq. 5 with those values held constant. For $h \in \{0,1\}^{V_{\text{rest}}}$ write $I_h := \{a : a \text{ an atom node}, h(a) = 1\}$, and let

$$T_{\Pi_\omega}(I) := \left\{A : \exists \left(A \leftarrow B^+, \text{ not } B^-\right) \in \Pi_\omega, \;\; B^+ \subseteq I \text{ and } B^- \cap I = \varnothing\right\} \tag{6}$$

be the immediate-consequence operator, written for a rule whose body splits into positive literals B^+ and negated ones B^-. On a definite program ($B^- = \varnothing$ throughout) this is van Emden–Kowalski's operator (van Emden & Kowalski, 1976) and is monotone. With negation it is not monotone in general, since the test $B^- \cap I = \varnothing$ can switch from true to false as I grows. That is precisely what stratification repairs: within one stratum every atom of B^- has already been fixed by strictly lower strata, so there the test is a constant and T_{Π_ω} is monotone stratum by stratum (Apt et al., 1988).

Why the next three sections exist. The reasoning graph is not introduced here: Skryagin et al. (2024) define it, give the synchronous update of Eq. 5, and specify the all-false initialisation. What that work does not establish is that the iteration is well behaved: that it terminates, that the state it settles in is a fixed point of the *classical* immediate-consequence operator rather than of some new one, and that it is the *least* such fixed point rather than an arbitrary one. Thm. 1 needs all three, since Lem. A identifies a program's stable models with least Herbrand models and nothing weaker will serve. Props. 3–4 and Lem. A supply them, and do so by reduction rather than by new machinery: the immediate-consequence operator of van Emden & Kowalski (1976), the fixed-point theorem of Tarski (1955), stratified semantics following Apt et al. (1988), and the splitting-set theorem of Lifschitz & Turner (1994) together with the choice-rule characterisation of Simons et al. (2002). The contribution is thus a bridge rather than a theory: it places the reasoning graph, and its differentiable reading, on the standing foundations of answer-set programming.

A.4 THE FORWARD PASS IS THE IMMEDIATE-CONSEQUENCE OPERATOR

Proposition 3 (Message passing implements the immediate-consequence operator)**.** *$h \in \{0,1\}^{V_{\text{rest}}}$ is a fixed point of $T^\omega_{\mathcal{G}}$ if and only if $I_h = T_{\Pi_\omega}(I_h)$. That is, $T^\omega_{\mathcal{G}}$'s fixed points are exactly T_{Π_ω}'s.*

Write $\underline{h}$ for the all-false state of the iteration, in which every node is false except the truth sink $\top$, and $\overline{h}$ for the all-true one. These are *states*, not nodes: they are the bottom and top of the lattice $\{0,1\}^{V_{\text{rest}}}$, and are unrelated to the graph's truth and constraint sinks $\top$ and $\bot$ of Sec. 2.2, which are ordinary nodes carrying values of their own.

Proof. Unfold the update's two logical cases against the definition of T_{Π_ω}. A conjunction node v represents one grounded rule's body, and $h(v) = 1$ iff every incoming positive-edge predecessor is 1 and every incoming NAF-edge predecessor is 0, which is exactly "this rule's body is satisfied in

I_h." An atom node a is the disjunction over its defining conjunctions, so $h(a) = 1$ iff at least one defining conjunction is 1, which is exactly "some rule for a fires given I_h," i.e. $a \in T_{\Pi_\omega}(I_h)$. Hence h is stationary precisely when $I_h = T_{\Pi_\omega}(I_h)$, atom by atom. □

Remark 2 (Granularity, not discrepancy). $T^\omega_\mathcal{G}$ updates conjunction and atom nodes in the same synchronous round, whereas T_{Π_ω} folds "check body" and "derive head" into one step, so two graph rounds simulate one application of T_{Π_ω} during the transient. This does not affect Prop. 3, which concerns fixed points rather than trajectories: a state is $T^\omega_\mathcal{G}$-stationary exactly when it is T_{Π_ω}-stationary, however many rounds either takes to arrive.

A.5 Existence and reachability of the fixed point

Proposition 4 (Existence and reachability). *$T^\omega_\mathcal{G}$ has a least fixed point h^ω, and the sequence $h_0 = \underline{h}$, $h_{t+1} = T^\omega_\mathcal{G}(h_t)$, converges to h^ω in at most $|V_{\mathrm{rest}}|$ rounds.*

Proof. (1) Monotonicity. Order $\{0,1\}^{V_{\mathrm{rest}}}$ pointwise. Conjunction and disjunction updates are monotone Boolean functions of their positive-edge predecessors. The only way monotonicity could fail is a NAF edge, whose contribution *decreases* as its source increases: if such an edge is read before its source has reached its final value, the reading node can fire on a stale "not yet derived". Since the operator never un-derives an atom, that stale fire is never retracted. The scope assumption splits this into two cases. If Π_ω is *definite*, meaning no rule body contains a negated literal, there is no NAF edge at all and every update is unconditionally a monotone Boolean function of positive-edge predecessors, however much positive recursion is present; this is VQAR's case. If Π_ω is stratified with negation present, every NAF edge points into a strictly lower stratum, which already assigns every node a finite stratum rank. Only *negative* cycles need excluding, and stratification excludes them by definition; positive recursion inside a stratum is permitted and is handled by the monotone iteration itself, so no acyclicity of the positive dependency graph is assumed here. By induction on stratum rank, each NAF source has reached its final value before any higher-ranked node reads it. In both cases $T^\omega_\mathcal{G}$ is monotone.

(2) Completeness. Π is ground with a finite Herbrand base, so $|V_{\mathrm{rest}}| < \infty$ and $\{0,1\}^{V_{\mathrm{rest}}}$ is a finite, hence complete, lattice.

(3) Existence. By Knaster–Tarski (Tarski, 1955), a monotone self-map of a complete lattice has a least fixed point h^ω.

(4) Reachability. By (1), $h_0 = \underline{h} \leq T^\omega_\mathcal{G}(h_0) = h_1$, and monotonicity propagates this to $h_0 \leq h_1 \leq h_2 \leq \cdots$. Continuity, which the Kleene-iteration argument of van Emden & Kowalski (1976) requires, follows automatically rather than being assumed: in a *finite* lattice every directed set contains its own supremum, so every monotone self-map preserves directed suprema. So $h_t \to h^\omega$; and since the chain is non-decreasing it must set at least one new node true at every non-stationary round, bounding convergence by $|V_{\mathrm{rest}}|$ rounds. □

Remark 3 (Why initialising at $\underline{h}$ is not a convenience). A monotone operator generally has many fixed points, and "all nodes true" is fixed for many programs. Knaster–Tarski's *least* fixed point is specifically what Kleene iteration *from the bottom* reaches; starting from $\overline{h}$ converges to the greatest one instead. The all-false initialisation is therefore what guarantees the iteration lands on M_ω, Π_ω's least Herbrand model, as Lem. A requires. Together, Props. 3–4 establish that "propagate to a fixpoint" is well defined, terminates, and computes M_ω specifically, not merely some self-consistent state.

A.6 The program fragment

Sec. 2.1 restricts ∂SLASH programs to normal rules with NPPs as the only source of nondeterminism. This appendix says why the two excluded constructs are excluded, and why they are not the same kind of thing, a distinction the reasoning graph's ∨-nodes make easy to blur.

A *choice rule* $\{a_1; \ldots; a_k\} \leftarrow \mathit{Body}$ places a second, unweighted choice beside the NPPs. Lem. A needs $U = \mathcal{A}_{\mathrm{npp}}$ to be a splitting set whose bottom program is *exactly* the NPP rules, and it needs the top program Π_ω to have a unique stable model; a choice rule breaks the latter, so a total choice

would determine a set of models rather than one and the bijection would become one-to-many. Its cardinality-one form $1\{a_1; \ldots; a_k\}1$ costs nothing, being precisely how NPPs are compiled (Simons et al., 2002).

A *disjunctive rule* $a_1 \mid \cdots \mid a_k \leftarrow$ *Body* is not a choice rule in the classical sense. Its answer sets are subset-minimal, so $a \mid b$ admits $\{a\}$ and $\{b\}$ but never $\{a, b\}$, and deciding whether a disjunctive propositional program has an answer set is Σ_2^p-complete (Eiter & Gottlob, 1995) rather than NP-complete: head disjunction is not syntactic sugar over normal rules. The distinction matters here because the reasoning graph does contain $\vee$-nodes, and they resemble head disjunction without being it. An $A_\vee$-node is *definitional*: one per head atom, gathering the bodies of the rules that define it, exactly as in Clark's completion (Clark, 1978). Reading a disjunctive head as such a node would compute the non-minimal semantics, which is why the fragment excludes it.

Definition 2 (Support)**.** Write $\mathcal{A}_{\mathrm{npp}}$ for the program's NPP atoms and $\mathrm{br}(v)$ for the branches feeding a node v. The support $\mathrm{supp}(v) \subseteq \mathcal{A}_{\mathrm{npp}}$ of v is $\{v\}$ if v is an NPP atom, $\varnothing$ if v is a deterministic fact, and $\bigcup_{b \in \mathrm{br}(v)} \mathrm{supp}(b)$ otherwise.

Definition 3 (Decomposability)**.** An $A_\wedge$-node with branches $b_1, \ldots, b_k$ is *decomposable* iff $\mathrm{supp}(b_i) \cap \mathrm{supp}(b_j) = \varnothing$ for all $i \neq j$.

A.7 The exactness gate

The gate of Sec. 2.3 in full: it decides, per node and before any value is computed, whether that node's branches may be combined exactly or must be sampled. Thm. 1 rests on two conditions, and the gate tests both: decomposability (Def. 3) at every $A_\wedge$-node, determinism (Def. 1) at every $A_\vee$-node.

Both tests turn on *instances* rather than outcome atoms. Write $\mathrm{inst}(b)$ for the set of grounded NPP instances whose outcome atoms appear in $\mathrm{supp}(b)$. Two different values of one instance are two different atoms, so they are support-disjoint while remaining perfectly dependent; a test phrased on supp alone would call them independent and multiply.

At an $A_\vee$-node the test is a certificate, not a support check. Write $\mathrm{excl}(b_i, b_j)$ when some literal of b_i and some literal of b_j are *distinct outcome atoms of the same NPP instance*, so that $b_i \wedge b_j$ asks one instance to take two values at once and $P_\Pi(b_i \cap b_j) = 0$ exactly. Write $\mathrm{conflict}(b_i, b_j)$ when one branch asserts a literal the other negates, which is exclusive for the same reason and must be tested first: such branches share an outcome atom, so the support test below would otherwise flag them.

Support-disjointness is neither certificate nor implied by one: two branches resting on unrelated NPPs have disjoint supports yet are *independent*, so $P_\Pi(b_i \cap b_j) = p_{b_i} p_{b_j} > 0$ and a plain sum would double-count them.

Remark 4 (The gate is sound, not complete). A pair is certified only on structural evidence, so the gate never calls a node exact that is not: it returns a closed-form value only when every pair of branches is either structurally incompatible or rests on disjoint instances. It is not complete, and deliberately so. Branches made exclusive by something other than an NPP value conflict, a constraint elsewhere in the program for instance, are not recognised, and branches sharing an outcome atom are flagged even where they happen to be exclusive. Both errors send a node to sampling that could have been computed exactly. That costs time and a nonzero η where $\eta = 0$ was available; neither can return a wrong number as certified-exact.

A.8 The sampling branch

When Alg. 1 routes a node to Monte Carlo, each NPP outcome leaf c is drawn through a Binary Concrete relaxation (Maddison et al., 2017; Jang et al., 2017),

$$\tilde{c} \;=\; \mathrm{sigmoid}\!\left(\frac{\mathrm{logit}\, p_c + G}{\tau}\right), \qquad G \;=\; \log U - \log(1-U), \quad U \sim \mathcal{U}(0,1), \tag{7}$$

where the logistic noise G is a difference of two Gumbel variates and $\tau > 0$ is a temperature. As $\tau \to 0$ the draw becomes an exact Bernoulli sample and the $\max$ combining a node's branches becomes Boolean disjunction, so the estimator is consistent in the limit; keeping $\tau > 0$ is what leaves the whole pass differentiable, and is the sole origin of η_{bias} in Rem. 1.

Algorithm 1 Exactness gate, applied per node v

Require: node v of type τ_v with branches $b_1, \dots, b_k$; tolerance (ε, δ)

$O \leftarrow \varnothing$ ▷ pairs not certified
if $\tau_v = A_\wedge$ **then** ▷ Def. 3
  for $1 \le i < j \le k$ **do**
    if $\mathrm{inst}(b_i) \cap \mathrm{inst}(b_j) \neq \varnothing$ **then**
      $O \leftarrow O \cup \{(i, j)\}$ ▷ one instance feeds both: $p_{b_i} p_{b_j} \neq P_\Pi(b_i \cap b_j)$
    end if
  end for
else ▷ $\tau_v = A_\vee$; Def. 1
  for $1 \le i < j \le k$ **do**
    if $\mathrm{conflict}(b_i, b_j)$ **then** ▷ one asserts what the other negates: $P_\Pi(b_i \cap b_j) = 0$
    else if $\mathrm{supp}(b_i) \cap \mathrm{supp}(b_j) \neq \varnothing$ **then**
      $O \leftarrow O \cup \{(i, j)\}$ ▷ shared outcome atom: correlated, not certified
    else if $\mathrm{excl}(b_i, b_j)$ **then** ▷ distinct values of one instance: $P_\Pi(b_i \cap b_j) = 0$
    else
      $O \leftarrow O \cup \{(i, j)\}$ ▷ independent: $P_\Pi(b_i \cap b_j) = p_{b_i} p_{b_j} > 0$
    end if
  end for
end if
if $O = \varnothing$ **then**
  return $\prod_i p_{b_i}$ if $\tau_v = A_\wedge$, else $\sum_i p_{b_i}$ ▷ condition holds: exact, $\eta = 0$
else
  $n_{\mathrm{mc}} \leftarrow \lceil \ln(2/\delta) \,/\, (2\varepsilon^2) \rceil$
  sample $\omega^{(1)}, \dots, \omega^{(n_{\mathrm{mc}})} \sim P_F$
  return $\frac{1}{n_{\mathrm{mc}}} \sum_t \mathbb{1}\left[\omega^{(t)} \models v\right]$ ▷ $|\eta_{\mathrm{sampling}}| \le \varepsilon$ w.p. $1 - \delta$
end if

A.9 Structural correspondence

Write $\Pi = \Pi_{\mathrm{npp}} \cup \Pi_{\mathrm{det}}$, splitting the program into its NPP rules and its deterministic remainder, and write $\mathrm{lfp}(\cdot)$ for the *least fixed point* of forward-chaining a program to convergence, the h^ω that Prop. 4 shows exists and is reached from $\bot$ in finitely many rounds.

Lemma A (Structural correspondence)**.** *The map $\omega \mapsto \mathrm{lfp}(\Pi_{\mathrm{det}} \cup \omega)$ is well defined and total on Ω_F, and its restriction to $\{\omega : \mathrm{lfp}(\Pi_{\mathrm{det}} \cup \omega) \not\models \bot\}$ is a probability-preserving bijection onto the stable models of Π: $P_\Pi(\mathrm{lfp}(\Pi_{\mathrm{det}} \cup \omega)) = P_F(\omega)$.*

Proof. Fix a total choice ω. Replacing every NPP atom f by the fact $f{=}\omega(f)$ turns Π into Π_ω, a program with no remaining choice constructs; since no NPP atom's definition depends on another NPP atom (scope assumption), the substitution is well defined independently of order. Π_ω inherits Π's stratification directly, because fixing NPP atoms to facts only deletes nodes and edges from the dependency graph, so Π's stratification remains valid for Π_ω unmodified. By the classical result for stratified programs (Apt et al., 1988), which reduces to van Emden–Kowalski least-fixpoint semantics on each stratum once no negation remains, Π_ω has a *unique* minimal (= perfect = stable) model M_ω, computed by iterating the immediate-consequence operator to convergence, the very iteration Props. 3–4 show the message passing performs. Constraints compile to conjunctions feeding $\bot$, so M_ω is consistent with Π's constraints iff $M_\omega \not\models \bot$.

The bijection follows from splitting rather than assertion, but the splitting set is not $\mathcal{A}_{\mathrm{npp}}$ itself. An NPP rule may carry a body (Eq. 1), and that body mentions non-NPP atoms: `#npp(digit(X),[0..9]) :- img(X).` mentions $\mathtt{img}(X)$. Taking $U = \mathcal{A}_{\mathrm{npp}}$ would therefore leave $\mathrm{b}_U(\Pi)$ mentioning atoms outside U, and the Splitting Set Theorem would not apply. Let D be the set of atoms on which NPP rule bodies depend, transitively, and put $U := D \cup \mathcal{A}_{\mathrm{npp}}$. The scope assumption is exactly that D contains no NPP atom, so that no NPP's *instantiation* depends on another NPP's *outcome*. Π's stratification therefore places all of D strictly below $\mathcal{A}_{\mathrm{npp}}$, and U is a *splitting set* for Π (Lifschitz & Turner, 1994): the bottom program $\mathrm{b}_U(\Pi)$ consists of the rules defining D together with the NPP rules, mentions only atoms of U, and no atom of U

occurs in the head of any top rule. It is stratification, not the absence of NPP bodies, that licenses the split. Since D is deterministic and stratified, $\mathrm{b}_U(\Pi)$ restricted to D has a unique stable model M_D, which fixes which NPP rule bodies are satisfied and hence which NPPs are instantiated at all, so Ω_F is well defined, being the product over exactly those. Given M_D the surviving NPP rules are independent cardinality-one choice rules, whose stable models are by Simons et al. (2002) exactly the total choices, one per ω and no more; the stable models of $\mathrm{b}_U(\Pi)$ are therefore precisely $\{M_D \cup \omega : \omega \in \Omega_F\}$. The Splitting Set Theorem then gives $I \models \Pi$ iff $I \cap U$ is a stable model of $\mathrm{b}_U(\Pi)$ *and* I is a stable model of the top program with $I \cap U$ plugged in as facts. The top program given ω is precisely Π_ω, whose unique stable model is M_ω. Hence every stable model I of Π agrees on $\mathcal{A}_{\mathrm{npp}}$ with exactly one ω_I and satisfies $I = M_{\omega_I}$; conversely every consistent M_ω is a stable model of Π. Probability preservation is immediate, since $P_\Pi(M_\omega)$ is defined as $P_F(\omega)$. □

A.10 Arithmetic exactness

Lemma B (Arithmetic exactness)**.** *Let v be a node whose ancestor sub-DAG is decomposable (Def. 3) and deterministic (Def. 1). Then the forward pass computes*

$$p_v = \sum_{\omega \in \Omega_F \,:\, \mathrm{lfp}(\Pi_{\mathrm{det}} \cup \omega) \models v} P_F(\omega). \tag{8}$$

Proof. By induction on node rank, the length of the longest path to an NPP atom or fact leaf. Rank is well defined and finite provided Q's ancestor cone is acyclic, a purely structural requirement unrelated to the NAF-staleness concern of Prop. 4. Under the acyclic sub-case of the scope assumption this is automatic; under the wider stratified condition it is not free in general, since a stratum's positive recursion could ground out cyclically, and must be checked against the actual ground program. For VQAR we *check* rather than assume it: a pre-scan walks every query's full ancestor cone across all three example programs and reports zero cycles, the genuine source-level recursion of `is_a` included, since its transitive closure grounds out acyclic because the underlying ontology is itself a DAG.

Base case. For $v \in \mathcal{A}_{\mathrm{npp}}$, p_v is the NPP's reported probability for the relevant value, which equals $P_\Pi(v{=}\mathrm{true})$ under the total-choice distribution after marginalising the other NPP atoms, which do not affect v by independence. For a deterministic fact node, $p_v = 1 = P_\Pi(v{=}\mathrm{true})$, as it holds in every M_ω.

$A_\wedge$-step. Let v have branches $b_1, \ldots, b_k$ and be decomposable. By the splitting decomposition in Lem. A's proof, every M_ω contains the same deterministic part M_D, so b_i's truth value is a function of M_D together with $\omega|_{\mathrm{supp}(b_i)}$, and since M_D does not vary with ω, a function of $\omega|_{\mathrm{supp}(b_i)}$ alone. The supports are pairwise disjoint and distinct NPP atoms are independent, so the events $\{b_i{=}\mathrm{true}\}$ are mutually independent; v is true iff all b_i are, whence $P_\Pi(v{=}\mathrm{true}) = \prod_i P_\Pi(b_i{=}\mathrm{true}) = \prod_i p_{b_i} = p_v$ by the induction hypothesis.

$A_\vee$-step. Let v have branches $b_1, \ldots, b_k$ and be deterministic. The events $\{b_i{=}\mathrm{true}\}$ are then pairwise disjoint, so $P_\Pi(v{=}\mathrm{true}) = P_\Pi(b_1 \vee \cdots \vee b_k) = \sum_i P_\Pi(b_i{=}\mathrm{true}) = \sum_i p_{b_i} = p_v$.

Composition. Applying the step up the DAG to v_Q gives $p_{v_Q} = P_\Pi(v_Q{=}\mathrm{true})$. Lem. A's bijection then lets us regroup that sum by the stable model each consistent, Q-satisfying ω determines, giving $\sum_{I \models \Pi,\, I \models Q} P_\Pi(I) = P_\Pi(Q)$. □

Remark 5 (Support counts NPP atoms, not atoms). Def. 2 collects only NPP atoms, a deterministic fact contributing $\varnothing$, and the $A_\wedge$-step needs exactly that restriction. Take `#npp(a,[...]) :- B.` and `#npp(b,[...]) :- B.`, two NPPs sharing a body: no cycle, and neither NPP depends on the other's outcome. The branches do share an ancestor, so a support defined over *all* atoms would intersect at `B` and decomposability would fail, yet the product is still correct, because `B` is deterministic, takes the same value in every M_ω, and so contributes no probabilistic dependence. Restricting supp to $\mathcal{A}_{\mathrm{npp}}$ is what makes the criterion track the dependence that matters rather than mere shared ancestry, and the splitting decomposition above is what licenses the restriction.

Remark 6 (Why tree encodings are unconditionally exact). In a tree-encoded $\mathtt{sum}_N$ reasoning graph every leaf owns exactly one NPP atom, so every node's support is a singleton or a disjoint union of singletons all the way up: decomposability holds at every $A_\wedge$-node automatically, and no $A_\vee$-node has more than one live child. Thm. 1 is therefore unconditional on this class. One structural

fact accounts for both of the tree encoding's observed advantages: because no NPP atom is shared between branches, no branch's probability can be counted twice. That is at once why the arithmetic is exact and why node values never drift above 1, as they do under the cartesian and chain encodings.

A.11 POSITIVE CYCLES

Lem. B requires the query's ground ancestor cone to be acyclic, and the requirement in fact begins one step earlier: Def. 2's recursion does not terminate on a cycle, so supp is undefined there before any theorem is stated.

The condition is not vacuous. A positive cycle carrying NPP mass breaks the forward pass in one of two ways, depending on the disjunction operator. On the program

```
#npp(coin(X),[heads,tails]) :- item(X).  item(i1).  b :-
            coin(i1,tails).  b :- c.  c :- b.
```

with $P(\texttt{tails}) = 0.5$, a plain sum reaches $h(b) = 15.0$ after 60 rounds and is still climbing, while the probabilistic sum *converges*, to $h(b) = 1.0$ against a true 0.5, with $\max \Delta < 10^{-9}$, so the convergence test passes. The silent failure is the one that matters, since it is the regime our experiments run in.

Two mechanisms bear on this, and only the second covers the general case. First, grounding removes a cycle whose foundedness is settled by *deterministic* atoms: the textbook example $\{\, a.\quad b \leftarrow \mathrm{not}\ a.\quad b \leftarrow b. \,\}$ grounds to $\{a\}$ alone and the cycle is gone before message passing begins. This does *not* extend to NPP-guarded cycles, and it would be wrong to claim it does. Replacing the fact by a neural one, `#npp(a,[0,1]).` with $b \leftarrow \mathrm{not}\ a(0)$ and $b \leftarrow b$, leaves $a(0)$ undetermined at grounding time, so the grounder must keep both rules and the loop reaches the graph intact. That is the failure mode above, not an exception to it. Second, and this is what actually covers that case, supp is undefined on a cycle, so the pre-scan cannot certify such an atom and does not pretend to: it collects every atom sitting on a positive cycle, excludes it from the decomposability and determinism checks, and reports it (`CorrectionReport.cycle_skipped`) with a warning that its value is not covered by Thm. 1. The guarantee is therefore conditional and *checked* rather than argued away: where no query depends on such an atom, the case for every benchmark we report, Thm. 1 applies, and where one does, the system says so instead of returning a silently wrong number.

The encoding of the ontology also matters, and we choose it deliberately. VQAR's ontology is closed under the recursive rule $\texttt{is_a}(A, C) \leftarrow \texttt{is_a}(A, B), \texttt{is_a}(B, C)$, which is genuine positive recursion. Rather than let the grounder unfold it, the *expanded* encoding we use materialises that closure during preprocessing and drops the rule, so this recursion never reaches the grounder at all. A smaller mutual recursion between `name` and `is_a` survives, and the pre-scan does find it: scanning the whole program reports 108 cyclic atoms on C_2 and 84 on C_4, all of them inside the `name/is_a/oa_rel` closure. None is reachable from a query. Scanning instead only the ancestor cones of the 26 (C_2) and 20 (C_4) target queries reports no cycle at all. Lem. B's precondition therefore holds on these programs, verified against the actual ground program rather than assumed.

A.12 THE MARGINALIZATION IDENTITY

This is the formal content of the claim in Sec. 2.3 that the total-choice axis is never built, and it is also what joins the two semantics. Skryagin et al. (2024) evaluate a reasoning graph over Booleans: fixing a total choice ω assigns every node the truth value $h^\omega(v)$. The forward pass of Eq. 5 instead carries the probability p_v. The identity below says these are not two semantics but one, related by averaging. Note the asymmetry in cost: its right-hand side ranges over $|\Omega_F|$ terms, while the left is a single number the forward pass already holds, and Defs. 3–1 are exactly the conditions under which the one can be obtained without enumerating the other.

Proposition 5 (Marginalization identity)**.** *For every node v,* $p_v = \sum_{\omega \in \Omega_F} P_F(\omega)\, h^\omega(v) = \mathbb{E}_{\omega \sim P_F}[h^\omega(v)]$.

Proof. This is Lem. B's induction read node by node rather than restated: its base case, $A_\wedge$-step and $A_\vee$-step all establish $p_v = P_\Pi(v{=}\mathrm{true})$ purely from the total-choice distribution and v's dependence on ω, and none of them invokes consistency $h^\omega(\bot)$ at any point. By definition of a probability under

$\omega \sim P_F$, $P_\Pi(v{=}\text{true}) = \sum_\omega P_F(\omega)\,\mathbb{1}[h^\omega(v)=1] = \mathbb{E}_{\omega\sim P_F}[h^\omega(v)]$, the indicator being redundant since $h^\omega(v) \in \{0,1\}$. □

Note the contrast with Lem. A, which holds only on consistent choices: Prop. 5 holds unconditionally over *all* of Ω_F, inconsistent ω included, for any node v. This is the precise sense in which the forward pass marginalizes the definitization. It is not an analogy but the identity itself, and it is why the total-choice axis never has to be materialized: Defs. 3–1 are exactly the conditions under which $\sum_\omega(\cdot)$ distributes through the circuit by the sum–product identity, instead of requiring $|\Omega_F|$ explicit terms.

A.13 Gradients and the simplex constraint

Proof of Prop. 2. By Thm. 1, $p_{v_Q} = \sum_{\omega\in\Omega_F:\,\mathrm{lfp}(\omega)\models Q} P_F(\omega)$, which Lem. A rewrites as $\sum_{I\models\Pi,\,I\models Q} P_\Pi(I) = \gamma$. Each $P_F(\omega) = \prod_f P(f{=}\omega(f))$ is multilinear in the outcome probabilities and contains p_c at most once, so $\partial P_F(\omega)/\partial p_c = P_F(\omega)/p_c$ when $\omega(f) = c$ and 0 otherwise. Summing over ω gives $\partial p_{v_Q}/\partial p_c = \alpha$, and dividing by $p_{v_Q} = \gamma$ gives the claim. □

Relation to the SLASH gradient. Following NeurASP, Skryagin et al. (2022) do not derive their gradient but *define* it, as the reward α of Eq. 4 less a penalty β over the competing outcomes of the same NPP, normalised by the query probability γ:

$$\frac{\partial \log P_{\Pi(\theta)}(Q)}{\partial p_c} := \Bigg[\underbrace{\sum_{\substack{I\,:\,I\models Q\\ I\models f=c}} \frac{P_{\Pi(\theta)}(I)}{P_{\Pi(\theta)}(f{=}c)}}_{=:\,\alpha\ \text{(reward)}} - \underbrace{\sum_{\substack{I,c'\,:\,I\models Q\\ I\models f=c',\ c\neq c'}} \frac{P_{\Pi(\theta)}(I)}{P_{\Pi(\theta)}(f{=}c')}}_{=:\,\beta\ \text{(penalty)}} \Bigg] \cdot \underbrace{\frac{1}{P_{\Pi(\theta)}(Q)}}_{=:\,1/\gamma}. \tag{9}$$

We have relabelled the variables into our notation: where Skryagin et al. (2022) write $c{=}v$ for an NPP atom taking a value, we write $f{=}c$ (App. A.1); the equation is otherwise theirs unchanged. Writing α_c, β_c for the reward and penalty attached to outcome c, the penalty proves not to be an independent quantity: two identities do the work,

$$\beta_c = \sum_{c'\neq c} \alpha_{c'}, \qquad \text{and} \qquad \sum_c p_c\,\alpha_c = \gamma. \tag{10}$$

The first is the definition of β unfolded. The second holds because every total choice fixes exactly one value of f, so summing over c partitions $\{I \models Q\}$. It is the identity with which Skryagin et al. (2022) open their own Theorem 1. The two accounts therefore differ in one respect only: *where the simplex constraint $\sum_c p_c = 1$ is discharged.* Raising p_c necessarily lowers the competing $p_{c'}$, and that coupling has to be paid for somewhere. Skryagin et al. (2022) pay for it inside the logic, subtracting an explicit penalty β_c summed over the competing outcomes; we pay for it in the network, because the softmax that produces $\mathbf{p}$ already enforces the constraint and back-propagating through it generates the coupling unbidden. Everything else in the two gradients is the same α/γ.

Proposition 6 (Exact agreement for binary NPPs)**.** *If f has two outcomes, so that $p_{c'} = 1 - p_c$, then $\dfrac{\partial \log p_{v_Q}}{\partial p_c} = \dfrac{\alpha_c - \beta_c}{\gamma}$, i.e. exactly Skryagin et al. (2022, Eq. 15).*

Proof. The constraint gives $\mathrm{d}p_{c'}/\mathrm{d}p_c = -1$, so by Prop. 2 and the chain rule $\mathrm{d}\log p_{v_Q}/\mathrm{d}p_c = (\alpha_c - \alpha_{c'})/\gamma$, and $\beta_c = \alpha_{c'}$ by Eq. 10. □

For $n > 2$ outcomes the constraint admits no such canonical coupling. Eq. 9 charges the full penalty to every competing outcome, which Skryagin et al. (2022) describe as training "by the principle winner takes all". Back-propagation instead discharges the constraint through the layer that creates it. Write z for the NPP's pre-softmax output, so that $p_c = \mathrm{softmax}(z)_c = e^{z_c}/\sum_{c'} e^{z_{c'}}$: the parameters θ reach the outcome probabilities only through z, and it is the softmax, not the logic, that enforces $\sum_c p_c = 1$. Differentiating it gives the exact coupling $\partial p_c/\partial z_{c'} = p_c(\delta_{cc'} - p_{c'})$, which with Eq. 10 collapses to

$$\frac{\partial \log p_{v_Q}}{\partial z_c} = \frac{p_c}{\gamma}\Big(\alpha_c - \sum_{c'} p_{c'}\,\alpha_{c'}\Big) = p_c\Big(\frac{\alpha_c}{\gamma} - 1\Big). \tag{11}$$

The penalty is thus recovered, but *probability-weighted* rather than uniform: each competing outcome is charged $p_{c'}\alpha_{c'}$ rather than the full $\alpha_{c'}$, so an outcome the network has already ruled out contributes almost nothing. This difference is not cosmetic, and it is sharpest exactly where training ends up.

Proposition 7 (The graph gradient vanishes at a confident optimum)**.** *Let the outcome probabilities of an NPP f be produced by a softmax over logits z, and suppose $p_{c^\star} \to 1$ for some outcome $c^\star$ with $\alpha_{c^\star} > 0$. Then $\partial \log p_{v_Q}/\partial z_c \to 0$ for every outcome c.*

Proof. Every ω with $\omega(f) = c$ contributes $P_F(\omega)$, which carries p_c as a factor exactly once, so $P_F(\omega)/p_c$ is independent of p_c and hence so is α_c; in particular each α_c is bounded as $p_c \to 0$. By Eq. 10, $\gamma = \sum_c p_c\alpha_c \to \alpha_{c^\star}$. Eq. 11 gives $\partial \log p_{v_Q}/\partial z_c = p_c(\alpha_c/\gamma - 1)$, which for $c = c^\star$ tends to $1 \cdot (\alpha_{c^\star}/\alpha_{c^\star} - 1) = 0$, and for $c \neq c^\star$ is a vanishing p_c against a bounded factor. □

The contrast is then exact rather than rhetorical. Under the same limit Skryagin et al. (2022, Thm. 1) show Eq. 9 tends to $(-1, \ldots, 1, \ldots, -1)$: a fixed, non-vanishing vector. The two rules agree on which direction is uphill, and agree outright when f is binary (Prop. 6); they differ in *magnitude near the optimum*. Eq. 9 keeps applying a unit-scale update after the model has stopped being wrong, so the step actually taken at convergence is set by the learning-rate schedule rather than by the objective, the behaviour Skryagin et al. (2022) name training "by the principle winner takes all". Eq. 11 is a true gradient of the loss and stops on its own. We state this as a property of the two update rules rather than a claim about which trains better: the prediction it makes is that the penalty term should show up late in training, as residual drift in the NPP logits after accuracy has saturated, rather than as a difference in early convergence. That is the form an empirical test would have to take.

Remark 7 (Why ∂SLASH and SLASH need not agree in practice). Prop. 6 is an equality for *binary* NPPs, and the benchmarks of Sec. 3 use multi-valued ones throughout (ten digits, four operators, several admissible features), so the two updates are not the same quantity there. Where they part, Eq. 11 is the gradient of the loss while Eq. 9 is a rule Skryagin et al. (2022) *define* rather than derive, and Prop. 7 gives the behavioural consequence: ours stops at a confident optimum, theirs keeps applying a unit-scale update. On tasks where SLASH does not finish at all, the comparison is not one of gradient quality but of feasibility. Where both finish, we expect and report accuracy *close* to SLASH's rather than far above it. An exact method cannot beat the exact method it reproduces, so parity is the ceiling and the gain we claim is in time.

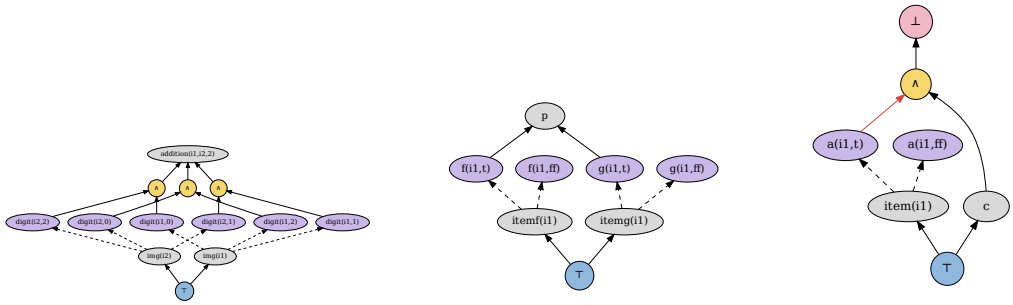
B EXAMPLE REASONING GRAPHS

Sec. 2.2 describes how a program becomes a reasoning graph in two paragraphs; this section shows the result on three small programs, so that the description can be checked against what the system builds. The three graphs in Fig. 4 were rendered by ∂SLASH from the programs listed below, so they show what the system actually builds rather than an idealisation of it. Read together they answer three questions: how a program becomes a graph, what the determinism test of Alg. 1 really turns on, and where negation and constraints appear.

(a) Determinism holds. MNIST-Addition over two images, ground and restricted to the query `addition(i1,i2,2)`:

```
#npp(digit(X), [0, 1, 2]) :- img(X).
img(i1).  img(i2).
addition(i1, i2, 2) :- digit(i1, 0), digit(i2, 2).
addition(i1, i2, 2) :- digit(i1, 1), digit(i2, 1).
addition(i1, i2, 2) :- digit(i1, 2), digit(i2, 0).
```

The NPP declaration yields one outcome atom per value, each instantiated from its guard by a dashed choice edge; these lavender leaves are where the network's probabilities enter. Each two-literal rule body becomes one $A_\wedge$ node, and all three feed `addition(i1,i2,2)` *directly*, because the head atom is itself the $A_\vee$ node of Sec. 2.2, not a separate one. Determinism holds here for a reason visible in the picture: $\wedge_0$ requires `digit(i1,1)` while $\wedge_1$ requires `digit(i1,0)`, and one NPP



(a) determinism holds (b) determinism fails (c) negation and $\bot$

Figure 4: Reasoning graphs for the three programs of App. B, drawn bottom-up in the direction the forward pass runs. Blue is the truth node $\top$, pink the constraint sink $\bot$, yellow the $A_\wedge$ nodes, lavender the NPP outcome atoms where neural probabilities enter, grey the derived atoms. Dashed edges instantiate an NPP's outcomes; red edges are negative ($w_e = -1$) and propagate $1 - h_{u,t}$.

instance cannot take two values, so no two branches can hold together and the disjunction is an exact plain sum.

(b) Determinism fails. Two independent NPPs supporting the same atom:

```
#npp(f(X), [t, ff]) :- itemf(X).
#npp(g(X), [t, ff]) :- itemg(X).
itemf(i1).  itemg(i1).
p :- f(i1, t).
p :- g(i1, t).
```

Structurally this is (a) in miniature: two branches converging on one head atom. The verdict is the opposite, and nothing about the shape says so. `f` and `g` are *different* NPP instances, so both branches can hold at once, $P_\Pi(b_1 \cap b_2) = p_{b_1} p_{b_2} > 0$, and a plain sum counts that world twice. This is why Alg. 1 tests excl, which instances the branches touch, rather than the graph's topology. Note also that a single-literal body needs no $A_\wedge$ node: `f(i1,t)` edges straight into `p`.

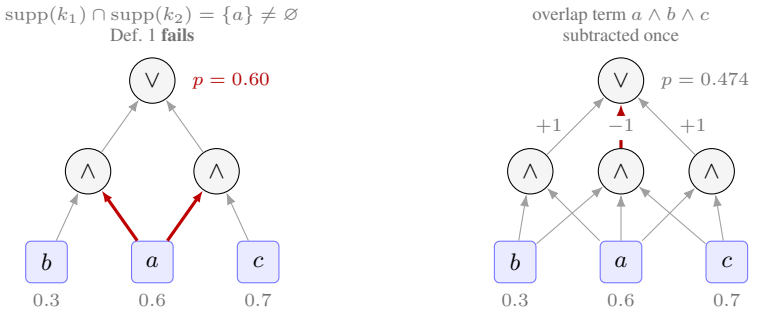
(c) Default negation and the constraint sink. A constraint whose body mixes a negated and a positive literal:

```
#npp(a(X), [t, ff]) :- item(X).
item(i1).  c.
:- not a(i1, t), c.
```

The constraint's two body literals meet at an $A_\wedge$ node that feeds $\bot$; a value of 1 there means the constraint is fully violated. The negated literal reaches that conjunction along a red negative edge, which propagates $1 - h_{u,t}$ rather than $h_{u,t}$. It survives grounding precisely because `a(i1,t)` is an NPP *outcome*: a choice, not a derivable fact, so its truth cannot be settled before the neural network is consulted. Finally, `a(i1,ff)` appears although no rule mentions it, because every declared outcome becomes a node, since the forward pass needs the full distribution over each instance, not only the values the program happens to use.

C WHY NOT INCLUSION–EXCLUSION?

Sec. 2.3 routes a flagged $\vee$-node to sampling without saying why the exact repair is not taken instead. This section gives the reason, and it matters for the paper's central claim: if inclusion–exclusion were affordable, the Hoeffding-certified fallback would be an unnecessary concession and η could always be driven to zero. The textbook repair for an overlapping disjunction is inclusion–exclusion. For an



(a) plain sum: $0.18 + 0.42$

(b) exact: $-P(a)P(b)P(c) = -0.126$

Figure 5: The overlap that breaks plain-sum disjunction, on the minimal program $p \leftarrow a, b$; $p \leftarrow a, c$. **(a)** a (red) feeds both branches, so supp of the two $A_\wedge$-nodes intersects and Def. 1 fails: the plain sum returns 0.60 against a true 0.474, double-counting exactly $P(a, b, c)$. Each $A_\wedge$-node is nonetheless decomposable (Def. 3), so the failure is localised at the disjunction. **(b)** Inclusion–exclusion adds the intersection term with weight -1, restoring exactness, but at VQAR scale ($k \approx 360$) that correction costs $2^k - 1$ terms, which is why Alg. 1 samples instead of repairing.

$A_\vee$-node whose branches $b_1, \ldots, b_k$ are not pairwise disjoint, the exact probability is

$$P_\Pi(v) \;=\; \sum_{\varnothing \neq S \subseteq \{1,\ldots,k\}} (-1)^{|S|+1}\, P_\Pi\Big(\bigcap_{i \in S} b_i\Big), \tag{12}$$

and on the two-branch program of Fig. 5 it is a single subtraction: $P(a)P(b) + P(a)P(c) - P(a)P(b)P(c) = 0.18 + 0.42 - 0.126 = 0.474$, against the plain sum's 0.60.

It does not scale, and not incidentally. Eq. 12 carries $2^k - 1$ terms, so restoring exactness costs time exponential in the number of mutually overlapping branches. This is the #P-hardness of exact weighted model counting, localised in one node. For example, one VQAR's `target(02)` has $k \approx 360$. Truncating the series is no escape either: it yields an *alternating* bound rather than a one-sided safety margin, with order 1 (the uncorrected plain sum) always an upper bound and order 2 a lower one, so only the full order $m = k$ is exact.

Monte Carlo pays a different price. It never forms the sum at all, so its cost is independent of k, and it returns a value whose error is bounded on demand (Remark 1) rather than being either exactly right or unaffordable. This is why ∂SLASH has one exact regime and one sampled one, with nothing in between: a middle branch would buy exactness only where the problem was easy anyway, at the cost of a case distinction the user would have to reason about.

D Analyzing overlap of current neurosymbolic benchmarks

Thm. 1 holds only where its two conditions hold, so the exact regime of Sec. 3 is worth nothing if real benchmarks violate them routinely. This section measures that directly: which tasks satisfy the conditions, which do not, and where the violations come from. It is the evidence behind Sec. 3.2's statement that the conditions hold for the base graph and fail for a measurable share of attached queries.

In this section, we will examine the number of samples that satisfy the exactness conditions per dataset and the number of nodes flagged by our checker for sampling.

Table 2 shows which of the task encodings give us exact query probabilities via the Sato conditions. We see that, except for one, all MNNIST-R tasks, LEAF, HWF, and object-centric learning tasks fulfill the conditions and don't require MC sampling. These tasks encode all queries in the same graph, so the graph remains the same for every sample, and we read probabilities from the corresponding query nodes during training and inference.

In VQA, we have a shared base graph that encodes all rules of the knowledge graph, as well as the NPPs. Each query is added to the base graph during training and inference, thus resulting in a

Table 2: **Neurosymbolic benchmark tasks fulfill the exactness conditions** Shown are the numbers of nodes flagged to violate the conditions (C), as well as the vertices (V) and edges (E) of each task encoding.

Task	C	V	E	Task	C	V	E	Task	C	V	E
sum_2	0	143	322	sum_3	0	410	941	sum_4	0	747	1801
mult_2	0	161	322	mod_2	0	134	322	equal	0	126	322
less_than	0	126	322	add_mod_3	0	127	322	add_sub	0	1063	4033
not_3_or_4	0	15	21	how_many_3_or_4	0	371	2,472	LEAF	0	42	63
HWF_3	0	750	1,823	HWF_5	0	8,354	24,427	HWF_7	0	76,027	231,704
OCL-SW4	0	355	544	OCL-CLEVR	0	11,367	20,582				

Table 3: **65% of all VQAR queries can exactly be computed.** Shown are the min, max and mean number of the 5k query nodes added to each fixed base graph (1,353,071 nodes / 1,512,677 edges)

	query nodes			query edges			
Clauses	min	max	mean	min	max	mean	exact
C_2	5	1,349	86	5	3,834	256	95.9%
C_3	5	2,485	227	5	9,656	687	75.8%
C_4	7	247,002	1,560	12	705,684	4,536	49.6%
C_5	5	349,044	978	11	1,005,312	2,991	59.9%
C_6	7	439,152	2,607	7	1,270,416	7,705	44.5%
all	5	439,152	1,091	5	1,270,416	3,234	65.2%

different graph for each batch. The basegraph fulfills the conditions, and Table 3 shows that 65.2% of all added queries do not introduce a violation. For the remaining 34,8% we fall back on MC sampling to approximate the query probability. Further, we observe that for simple queries, only a few nodes and edges are added, whereas for complex ones, we nearly double the number of nodes and edges relative to the base graph.

E Experimental Details

We have run all experiments on an A100 with 40GB and 80GB for Googol MNIST. Except for the VQAR experiment, 5 seeds were used. The means and standard deviations are shown in the extended tables below.

E.1 The exact regime

Tab. 4 shows the mean accuracies and std. for all eleven MNIST-R operations and the Leaf and HWF tasks reported in Fig. 2 in the main paper. We follow the setup of Solko-Breslin et al. (2024) training on 5k samples, testing on 500, using the same variant of LeNet (LeCun et al., 1998). The accuracy is reported after a fixed number of epochs. ∂SLASH as well as DPL, Scallop, SLASH, Dolphin, and ISED use an epoch budget of 10. Only REINFORCE and IndeCateR (50) are given more, as their own papers specify and convergence takes longer. The LR is fixed across all runs and between 1e-4 and 1e-3, with a fixed batch size. We select them based on the baseline code in Solko-Breslin et al. (2024) and note that we were able to reproduce their results. For all baselines except DPL, we found that on the add-mod-3 tasks, 1-2 seeds collapse (acc$<$10%) or get stuck in a local minimum (acc$<$70%), which we replaced with additional seeds.

In the LEAF task, we sample 30+10 images per class for training and testing, totaling 330+110=440 samples. We obtain higher DPL values in our reproduction, which may be due to our selection of 5 seeds rather than 10, as in Solko-Breslin et al. (2024). The batch size was 16, and the learning rate was 1e-4 for all baselines. Baselines run for a fixed epoch budget of 30-100.

For HWF, we use 10k training and 2k test formulas with varying lengths (1-7). As the current implementation of ∂SLASH does not yet support floating-point numbers, we use the least common

Table 4: Accuracy (%) on MNIST arithmetic operations, leaf classification, and HWF. Values are mean ± std over 5 seeds. TO = timeout; The mean row averages the 11 MNIST tasks a method completed. DPL and SLASH Timeout on tasks involving many digits.

Task	DPL	Scallop	REINFORCE	IndeCateR	ISED	SLASH	Dolphin	∂SLASH
Sum-2	95.2 ± 0.6	95.4 ± 0.9	95.5 ± 0.8	95.6 ± 0.5	95.6 ± 1.8	$\mathbf{96.0 \pm 0.5}$	95.1 ± 0.7	95.9 ± 1.1
Sum-3	93.5 ± 0.5	93.7 ± 0.7	92.5 ± 0.7	92.8 ± 1.3	$\mathbf{94.8 \pm 1.3}$	92.8 ± 1.1	92.0 ± 1.0	93.8 ± 0.9
Sum-4	TO	91.2 ± 2.0	92.2 ± 0.7	91.8 ± 1.3	$\mathbf{94.6 \pm 0.8}$	TO	90.0 ± 0.4	92.4 ± 1.0
Mult-2	95.4 ± 0.2	95.1 ± 1.2	95.7 ± 0.8	95.8 ± 0.9	95.7 ± 1.0	95.6 ± 0.5	95.4 ± 0.6	$\mathbf{96.2 \pm 0.5}$
Add/Sub	93.1 ± 0.7	93.8 ± 0.7	92.8 ± 0.4	93.1 ± 1.0	$\mathbf{95.4 \pm 1.0}$	93.4 ± 0.9	91.8 ± 1.0	93.2 ± 0.8
Equal	$\mathbf{98.5 \pm 0.3}$	71.6 ± 2.8	98.0 ± 0.4	98.1 ± 0.8	96.1 ± 1.8	97.5 ± 0.7	97.1 ± 0.9	97.8 ± 0.6
Less-Than	$\mathbf{97.0 \pm 1.3}$	79.8 ± 2.5	93.9 ± 1.7	93.7 ± 0.6	95.5 ± 1.4	95.1 ± 1.0	93.4 ± 1.4	94.0 ± 1.1
Add-Mod-3	95.6 ± 0.9	94.7 ± 0.8	96.0 ± 0.5	96.0 ± 0.5	82.0 ± 12.3	95.4 ± 0.5	94.6 ± 0.9	$\mathbf{96.1 \pm 0.5}$
Mod-2	$\mathbf{97.2 \pm 0.4}$	96.1 ± 0.6	96.2 ± 0.2	96.9 ± 0.7	96.2 ± 0.9	96.6 ± 0.9	96.0 ± 0.5	96.7 ± 0.6
Not-3-or-4	98.0 ± 0.5	96.4 ± 0.8	98.5 ± 0.3	98.6 ± 0.3	98.3 ± 0.8	98.6 ± 0.2	98.8 ± 0.4	$\mathbf{98.9 \pm 0.3}$
How-Many	TO	93.0 ± 1.1	93.8 ± 0.6	94.8 ± 1.4	$\mathbf{95.4 \pm 0.6}$	TO	93.3 ± 1.0	94.9 ± 0.5
Mean (11 ops)	$96.0^{(9/11)}$	91.0	95.0	95.2	94.5	$95.7^{(9/11)}$	94.3	**95.4**
Leaf-11	79.6 ± 3.5	79.0 ± 3.8	79.2 ± 5.5	79.2 ± 5.2	$\mathbf{81.2 \pm 4.4}$	79.0 ± 4.0	80.1 ± 3.8	80.7 ± 6.2
HWF	TO	96.6 ± 0.2	94.1 ± 0.5	94.7 ± 0.6	97.2 ± 0.6	TO	89.7 ± 3.0	$\mathbf{98.1 \pm 0.3}$

Table 5: Recall@5 on VQAR under both training regimes. Each column is shaded on its own scale.

Train →	10k C_2			50k C_{all}		
Test ↓	Scallop	SAME	∂SLASH	Scallop	SAME	∂SLASH
1k C_2	76.50	77.76	78.80	84.34	79.26	84.60
1k C_3	76.48	73.75	80.10	81.97	72.95	85.00
1k C_4	77.00	67.41	79.80	80.90	69.50	85.80
1k C_5	79.30	68.87	79.60	83.76	69.33	87.50
1k C_6	76.98	64.86	80.70	82.61	63.18	86.80
5k C_{all}	77.25	70.52	79.80	82.72	70.90	85.94

multiple to ensure that all divisions yield integer results. All baselines run for 50 epochs with LR 1e-4 and BS 16.

E.2 Visual Question Answering

In Sec . 3.2 in the main paper, the 50k C_{all} results are reported. Tab. 5 gives both training regimes, training on C_2 and C_{all}, side by side. For this task, we report the results from Skryagin et al. (2023) for Scallop and SLASH with Same, and use the same seed for comparability. Only one seed is used here, as rerunning Scallop and SLASH across all clauses takes multiple days. We have rerun the first 10 epochs on our hardware to enable a fair time comparison, which is shown in Fig. 6.

The ordering is identical in each: ∂SLASH leads at every clause length, and the gap to SAME widens as clauses grow longer, which is what one expects of a method that prunes outcomes rather than accounting for them. In the experimental details of (Skryagin et al., 2023) the authors noted that for some tasks they needed to enumerate millions of stable models. They reduced the number of models with Clingo's #show commands and iterative solving of relations. The beauty of ∂SLASH is that, via the marginalization identity, we don't have to enumerate all these models. Instead, we handle them implicitly by propagating their averages through the network (Sec. A.12).

E.3 Object-Centric Learning

For Object-Centric Learning, we used the same LR and batch size as in SLASH and SAME. The average precision measurements were taken from Skryagin et al. (2023), as rerunning the CLEVR (Johnson et al., 2017) experiments for SLASH/SAME would take multiple days, and we used five different seeds for our ∂SLASH runs. The time measurements are averages from reruns of the first 10 training epochs.

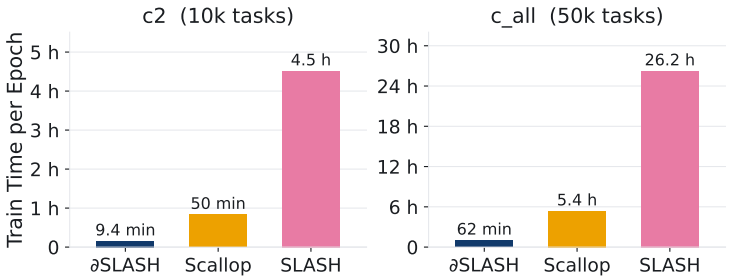


Figure 6: Time per epoch on VQAR 10k C_2 and 50k C_{all}.

E.4 Googol MNIST Addition in detail

Sec. 3.4 reports that ∂SLASH trains MNIST-Addition to $N{=}100$ and names three things that make it possible. This appendix supplies each: how the encodings differ, why the operators must run in log space, why the confidence penalty is a precondition rather than a refinement, and where the memory goes.

Three encodings of one problem. The reasoning graph is a syntactic image of the program, so rewriting the program rewrites the graph even when the semantics is untouched. The *cartesian* encoding introduces one conjunction per digit combination, each with N incoming edges; the graph is then essentially the total-choice space itself and grows as 10^N. The *chain* encoding threads a running partial sum through the images one at a time, reusing every intermediate result, which trades that product for a graph linear in N at a depth of N rounds. The *tree* encoding combines partial sums pairwise, keeping the same reuse at depth $\lceil \log_2 N \rceil$, which is both the smallest graph and the shallowest. Fig. 3 measures all three on one 80 GiB A100: they exhaust it at $N{=}6$, $N{=}32$ and $N{=}63$. App. F lists all three in full (Lst. 1–3). Every result reported in Sec. 3.4 uses the tree encoding.

Why the operators must run in log space. A conjunction node multiplies the probabilities of its branches, so a derivation of depth d forms a product of d factors in $[0, 1]$. Two things go wrong at once. The value underflows, and so, by the binding problem, does its gradient: $\partial \prod_i p_i / \partial p_j = \prod_{i \neq j} p_i$, which tends to 0 exactly when the product does, so the signal reaching an NPP leaf vanishes with depth. In log space $\otimes$ becomes a sum of logs, whose derivative with respect to each summand is 1; gradients then survive any depth, and $\oplus$ becomes the corresponding `log1mexp` co-norm. This is what makes the deep tree and chain sums trainable at all: with linear-space operators nothing beyond $N \approx 10$ converges.

Why the confidence penalty is a precondition. For large N the observed sum is a weak constraint: many assignments of N digits produce the same total, so a network that is confidently wrong in mutually-cancelling ways fits the data as well as one that is right. Gradient descent finds such solutions early and then cannot leave them, because the softmax outputs have already saturated. Following Pereyra et al. (2017) we subtract $\lambda H(\mathbf{p})$ from the loss, penalising low-entropy NPP outputs and keeping the distribution open long enough for the constraint to disambiguate it. Tab. 6 gives the ablation: the penalty costs nothing where the problem is small, and is the difference between training and collapse where it is not. The standard deviations are the symptom: at $\lambda{=}0$ they exceed 30 points because some seeds train and others fail outright, which an average alone would hide.

Table 6: Confidence penalty ablation, 5 seeds, tree encoding. At $\lambda{=}0$ the seed-to-seed standard deviation exceeds 30 points: individual runs collapse rather than degrade.

	Task accuracy (%)		Digit accuracy (%)	
N	$\lambda > 0$	$\lambda = 0$	$\lambda > 0$	$\lambda = 0$
20	86.2 ± 1.2	28.7 ± 31.6	99.09 ± 0.05	90.31 ± 4.92
50	68.9 ± 1.7	43.1 ± 31.6	99.12 ± 0.04	92.12 ± 9.84

Where the memory goes. Tab. 7 reports the full sweep. Graph size grows roughly quadratically in N under the tree encoding (4.7k nodes at $N{=}10$, 417k at $N{=}100$), and peak memory follows it. The consequence is visible in the batch column: $N \leq 50$ runs with batch size 500, while at $N{=}100$ a smaller batch size of 100 is required and still peaks at 49.3 GB. Per sample that is 0.49 GB against 0.11 GB at $N{=}50$, so $N{=}100$ at batch size 500 would have needed on the order of 250 GB. Time per epoch grows over the same range from 9.5 s to 181.1 s, which is a factor of 19 against a factor of 89 in graph size: the run is memory-bound, not compute-bound, and reducing the batch is the only lever these results use. Gradient checkpointing (Chen et al., 2016) recomputes activations in the backward pass instead of storing them, turning the $O(\text{steps})$ tape into $O(1)$ at the cost of one extra forward pass; it is implemented but not enabled for any run reported here, and it is the lever we used to carry the tree encoding past $N{=}100$.

Table 7: MNIST-Addition, tree encoding, mean over 5 seeds. Time is per epoch; memory is peak allocated.

N	nodes	edges	batch	task (%)	digit (%)	time (s)	VRAM (GB)
2	183	362	500	98.35	98.95	1.9	0.36
3	443	974	500	97.55	98.99	1.5	0.42
4	762	1807	500	96.84	99.03	1.2	0.59
5	1231	2970	500	96.11	98.98	1.1	0.86
6	1721	4298	500	94.89	98.95	1.9	1.10
7	2303	5889	500	93.94	98.97	2.0	1.38
8	2964	7721	500	93.47	98.99	2.1	1.68
9	3840	9975	500	92.67	98.99	2.3	2.22
10	4663	12284	500	92.34	99.03	9.5	2.62
20	17782	49407	500	86.20	99.09	17.8	9.26
50	106227	307370	500	68.90	99.12	68.7	54.71
100	416748	1224937	100	45.20	98.97	181.1	49.32

F Program Listings

Sec. 3.4 and App. E.4 argue that the encoding decides the reachable problem size, the three programs below being semantically identical and structurally unalike. They are given in full so that the claim can be checked rather than taken on the graph sizes alone.

```
#npp(digit(X), [0,1,2,3,4,5,6,7,8,9]) :- img(X).
img(i1). img(i2). img(i3). img(i4).

addition(i1,i2,i3,i4, N1+N2+N3+N4) :-
    digit(i1,N1), digit(i2,N2), digit(i3,N3), digit(i4,N4).
```

Listing 1: Cartesian encoding: one rule, 10^4 ground instances, depth 2.

```
#npp(digit(X), [0,1,2,3,4,5,6,7,8,9]) :- img(X).
img(i1). img(i2). img(i3). img(i4).

partial_sum(i1, V)   :- digit(i1, V).
partial_sum(i2, S+V) :- partial_sum(i1, S), digit(i2, V).
partial_sum(i3, S+V) :- partial_sum(i2, S), digit(i3, V).
partial_sum(i4, S+V) :- partial_sum(i3, S), digit(i4, V).

addition(i1,i2,i3,i4, S) :- partial_sum(i4, S).
```

Listing 2: Chain encoding: partial sums added one digit at a time, 570 ground rules, depth 5.

```
#npp(digit(X), [0,1,2,3,4,5,6,7,8,9]) :- img(X).
img(i1). img(i2). img(i3). img(i4).

% leaves
sum0(i1, V) :- digit(i1, V).   sum0(i2, V) :- digit(i2, V).
```

```
sum0(i3, V) :- digit(i3, V).   sum0(i4, V) :- digit(i4, V).

% level 1: pairs
sum1(g1_1, S+V) :- sum0(i1, S), sum0(i2, V).
sum1(g1_2, S+V) :- sum0(i3, S), sum0(i4, V).

% level 2: root
sum2(g2_1, S+V) :- sum1(g1_1, S), sum1(g1_2, V).

addition(i1,i2,i3,i4, S) :- sum2(g2_1, S).
```

Listing 3: Tree encoding: balanced pairwise sums, 561 ground rules, depth 4.