Learning Neural Programs

ALAIA SOLKO-BRESLIN, University of Pennsylvania, United States

A neural program is a composition of a neural model M_θ followed by a program P that performs reasoning over the predictions of M_θ. Neural programs can be used to solve computational tasks that cannot be solved by neural perception alone but can be easily expressed by such a composite. P can take many forms, including a program written in a general-purpose language or an API call to a modern LLM. We study the problem of learning neural programs using only end-to-end input-output labels for the composite. When P is written in a differentiable logic programming language, neurosymbolic learning techniques are applicable. On the other hand, when P is non-differentiable, learning neural programs requires estimating the gradients of black-box components. In this survey, we compare different techniques that are suitable for learning neural programs in each setting. We evaluate the techniques on a diverse set of benchmarks, including those from the neurosymbolic learning literature as well as benchmarks in which P makes a call to GPT-4. We find that the techniques exhibit trade-offs between expressiveness, scalability, data efficiency, and sample efficiency.

Additional Key Words and Phrases: neurosymbolic learning

ACM Reference Format:
Alaia Solko-Breslin. 2024. Learning Neural Programs. 1, 1 (July 2024), 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Many computational tasks cannot be solved by neural perception alone but can be naturally expressed as a composition of a neural model M_θ followed by a program P written in a traditional programming language or an API call to an LLM. Such composites are called "neural programs" [17] and we study the problem of learning neural programs in an end-to-end manner. One problem that is naturally expressed as a neural program is leaf identification, where 3 neural networks predict the shape, margin, and texture features of a leaf image, and P classifies the leaf species given these features (Fig. 1). Note that P can be written as a decision tree in Python or as a call to GPT-4.

The neural program learning problem concerns how to optimize model parameters M_θ which are being supervised by a fixed program P. Specifically, we are given a training dataset $\mathcal{D}$ of length N containing input-output pairs, i.e., $\mathcal{D} = \{(x_1, y_1), \ldots (x_N, y_N)\}$. Each x_i represents unstructured data (e.g., image data) whose corresponding structured data (intermediate labels) are not given. Each y_i is the result of applying P to the structured data corresponding to x_i. Given a loss function $\mathcal{L}$, we want to minimize the loss of $\mathcal{L}(P(M_\theta(x_i)), y_i)$ for each (x_i, y_i) pair in order to optimize θ.

Several existing techniques aim to solve this problem, each providing a different mechanism for differentiating P. We divide these approaches into two categories: neurosymbolic learning frameworks and black-box gradient estimation techniques. We elaborate on these categories and describe prior work that is relevant to each one.

Neurosymbolic Learning Frameworks. Neurosymbolic learning [1] is a learning paradigm that is suitable for neural programs when P takes the form of a logic program. DeepProbLog (DPL) [10] and Scallop [9] are frameworks that extend ProbLog and Datalog, respectively, to ensure that the symbolic component P is differentiable. Since training uses only end-to-end labels, this differentiability requirement is what facilitates learning in many neurosymbolic learning

Author's address: Alaia Solko-Breslin, University of Pennsylvania, Philadelphia, Pennsylvania, 19104, United States, alaia@seas.upenn.edu.

2024. XXXX-XXXX/2024/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

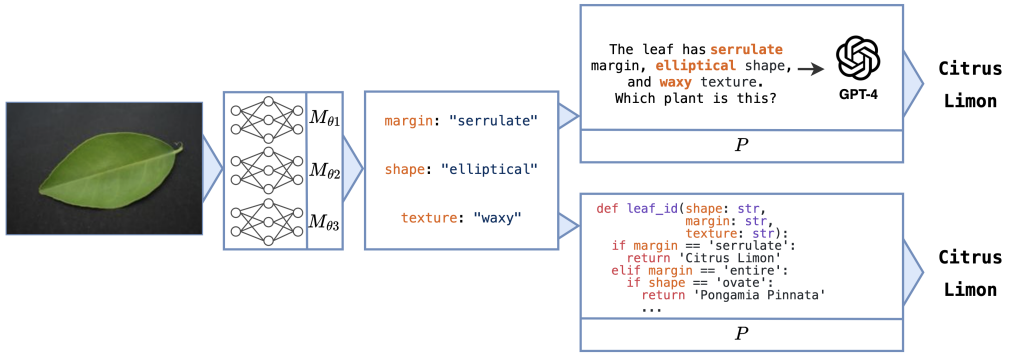

Fig. 1. Neural program decomposition for leaf identification. Three neural networks predict the margin, shape, and texture features of the leaf image. These features are then fed into a program P that returns the final classification. P can be written as a call to GPT-4 or as a simple decision tree implemented in Python.

frameworks. Despite not having intermediate labels for training, these frameworks are capable of solving complex tasks such as sorting [10], visual question answering [9], and path planning [18].

Black-Box Gradient Estimation Techniques. When P cannot be expressed as a logic program, techniques that estimate the gradient of P are applicable. For instance, REINFORCE [21] samples symbols from distributions predicted by M_θ and rewards sampled inputs that result in the correct output. Additionally, IndeCateR [16] is a REINFORCE-based gradient estimator with low variance that is well-suited for tasks with a high dimensional input space. NASR [4] is another REINFORCE-based gradient estimator that is tailored for the task of fine-tuning neural programs such as Sudoku solving. A-NeSI [19] is an algorithm that trains two additional neural networks that are used to estimate gradients of black-box programs. Lastly, ISED [17] is a REINFORCE-based algorithm that differentiates through a summary program of P that is produced through a sampling procedure.

We evaluate the techniques on a diverse set of benchmarks, including some from the neurosymbolic learning literature as well as benchmarks in which P makes a call to GPT-4. We include the latter tasks in the benchmark suite to demonstrate that certain tasks can be learned by black-box gradient estimation techniques but cannot be encoded in the logic programming languages that neurosymbolic learning frameworks support. With these benchmarks, we evaluate and compare the different approaches in terms of their overall accuracy as well as data and sample efficiency.

Our evaluation suggests that the presented techniques exhibit trade-offs between expressiveness, scalability, data efficiency, and sample efficiency. Specifically, neurosymbolic frameworks are limited in their expressiveness due to their use of logic programming languages. Additionally, A-NeSI is limited by data inefficiency and low accuracy on tasks involving complex programs due to its training of additional neural networks to aid in gradient approximation. Lastly, REINFORCE-based solutions often struggle with scalability or sample efficiency.

The structure for the rest of the paper is as follows. We describe the neurosymbolic learning frameworks DeepProbLog and Scallop in §2 and §3, respectively. We discuss the black-box gradient estimation techniques, REINFORCE, IndeCateR, NASR, A-NeSI, and ISED, in §4, §5, §6, §7, and §8, respectively. We summarize the results of our evaluation in §9. Lastly, we provide some final remarks on the strengths and weaknesses of the presented techniques in different settings and propose promising directions for future work in §10.

2 NEUROSYMBOLIC INFERENCE WITH DEEPPROBLOG

DeepProbLog (DPL) [10] is a neurosymbolic learning framework that requires program components to be expressed in ProbLog [13], a language that extends Prolog by adding the notion of probabilistic facts. This difference from Prolog is necessary because inference will ultimately be performed over probabilistic outputs of neural networks. One task whose solution can be expressed in DPL is the addition of two numbers (`sum2`). The predicate `sum2(X1, X2, Y)` indicates that `X1` and `X2` are images of digits `N1` and `N2` respectively, and `Y` is equal to the sum of `N1` and `N2`:

```
sum2(X1, X2, Y) :- digit(X1, N1), digit(X2, N2), Y is N1 + N2.
```

During training, the goal is to learn the neural predicate `digit` using only end-to-end labels. After training, DPL is able to assign a probability to the validity of predicates, e.g., `sum2(` , , 7).

2.1 Inference in DPL with sum2

Suppose that we want to train DPL on the `sum2` task with dataset $\mathcal{D}$. During training, DPL will perform inference on a sample $((x_1, x_2), y) \sim \mathcal{D}$. For each possible output, DPL will compute all possible proofs of the output. For example, there are seven possible input combinations that result in a final result of 7, and all such proofs will be considered during inference.

Computing all possible proofs is made possible by a Selective Linear Definite clause resolution (SLD resolution) procedure which decides whether a ground goal is entailed by by facts in a program by repeatedly applying rules to a goal. During inference, ProbLog computes the probability of a goal g as

$$P(g) = \sum_{M \models g} \sum_{f \in M} p_f \prod_{f \notin M} (1 - p_f) \tag{1}$$

where $M \models g$ means that M is a set of facts that make g true. Note that this inference procedure is equivalent to the Weighted Model Counting (WMC) procedure from the literature on probabilistic logic [6]. We can define the WMC procedure as follows. Suppose that a neural network M_θ predicts distributions for x_1 and x_2, i.e., $p_1 = M_\theta(x_1), p_2 = M_\theta(x_2)$. Then the probability of each possible output y would be defined as

$$p(y|(p_1, p_2)) = \mathbb{E}_{p((n_1,n_2)|(p_1,p_2))}[\mathbb{1}_{n_1+n_2=y}]. \tag{2}$$

While DPL is effective at learning tasks that have a small input space such as `sum2` and `sum3`, the WMC procedure becomes intractable with tasks with a larger input space. This is because computing the expectation in Equation 2 is known to be #P-hard [15]. We elaborate on the scalability challenges of DPL in §9. It is worth noting that DPL has been extended with approximate inference by using a procedure that considers only the best proof when performing inference [11]. However, this approach is limited because the best proof is often not correct, especially early in the training process when there is a large input space. To overcome this, DPL with exact inference often relies on curriculum learning to achieve high accuracy.

3 SCALABLE NEUROSYMBOLIC LEARNING WITH SCALLOP

Scallop [9] is a neurosymbolic learning framework that extends Datalog to use probabilistic facts. To understand how Scallop differs from DPL, we must first understand the differences between Prolog and Datalog in how they evaluate queries. During evaluation, Prolog starts from the query and works backwards to try to find facts and rules that can be used to satisfy the query. This strategy is known as a *top-down* evaluation strategy. Conversely, Datalog uses a *bottom-up* evaluation strategy in which the evaluation engine starts with base facts and derives new facts by applying rules until

no new facts can be derived. If the query is among the derived facts, then it is valid. Scallop uses this bottom-up evaluation strategy to maintain a set of proofs of derived facts.

3.1 Provenance for Deductive Databases

DPL with exact inference is unscalable because it considers all possible proofs for each fact in its evaluation. In contrast, Scallop considers only a subset of proofs for each fact during its bottom-up evaluation, making inference more efficient. For a given task, a user can select a *provenance semiring* [5] that gives the best heuristic for gradient calculation. Different semirings are well-suited for different tasks. One semiring, called `diff-top-k-proofs`, considers only the set of top-k most likely proofs for each fact during its evaluation. For example, when $k = 1$, the single highest-probability proof will be considered when deriving each fact, and the rest will be discarded.

3.2 Top-k Semiring Example

To demonstrate the evaluation strategy for `diff-top-k-proofs`, consider the `sum2` task from before. In Scallop, we can write the program as

```
sum2(a + b) :- digit_1(a), digit_2(b).
```

Suppose that we want to compute the probability of the fact `sum2(`  `,`  `, 7)`. First, we get outputs from the neural network, i.e., $p_a = M_\theta($$)$, $p_b = M_\theta($$)$. We state the probabilistic facts that will be used in this example in Table 1.

Table 1. Example predicted distributions from M_θ for each digit.

`0.01::digit(`  `, 0)`	`0.02::digit(``, 5)`	`0.01::digit(`  `, 0)`	`0.89::digit(``, 5)`
`0.01::digit(`  `, 1)`	`0.01::digit(``, 6)`	`0.01::digit(`  `, 1)`	`0.01::digit(``, 6)`
`0.81::digit(`  `, 2)`	`0.02::digit(``, 7)`	`0.01::digit(`  `, 2)`	`0.01::digit(``, 7)`
`0.09::digit(`  `, 3)`	`0.01::digit(``, 8)`	`0.01::digit(`  `, 3)`	`0.01::digit(``, 8)`
`0.01::digit(`  `, 4)`	`0.01::digit(``, 9)`	`0.03::digit(`  `, 4)`	`0.01::digit(``, 9)`

The top-1 most likely proof for `sum2(`  `,`  `, 7)` would be `digit (``, 2)`, `digit (``, 5)` with probability $0.81 \cdot 0.89 = 0.7209$. Scallop only keeps track of the top-k most likely proofs, so it would only consider this proof and not the other six that could lead to a final result of 7. Since the possible range of final outputs is 0 to 18, Scallop will keep the top-1 proof for each of the 19 outputs. The top-1 proofs for each possible output comprise the predicted value that gets passed into the loss function (such as binary cross-entropy loss), along with a one-hot vector corresponding to the ground truth output.

3.3 Limitations of Scallop

Scallop's bottom-up evaluation strategy, along with its consideration of a subset of possible proofs, makes it significantly more scalable than DPL. However, Scallop is based on Datalog, which is not Turing complete, so solutions to many tasks cannot be encoded in its language. For instance, writing a sorting algorithm for lists of arbitrary length would be possible in DPL but not Scallop. Additionally, both Scallop and DPL use logic programs, which do not support arbitrary calls to APIs, so they are not compatible with API calls to modern LLMs. Thus, they are not suitable for the general neural program learning setting. We now describe several black-box gradient approximation techniques that can address this limitation for learning general neural programs.

4 BLACK-BOX GRADIENT ESTIMATION WITH REINFORCE

While neurosymbolic learning frameworks restrict program components to logic programs, REINFORCE [21] allows the program component to take any form. This is possible because the algorithm does not treat the program component as a white-box from which it extracts a gradient. Instead, after neural networks predict distributions for the input data, REINFORCE samples symbols and evaluates the black-box program on those symbols. Correct outputs are then rewarded so that weight adjustments to the neural networks are made in the direction of expected reinforcement. REINFORCE works in the neural program learning setting because unlike general reinforcement learning, REINFORCE algorithms involve *immediate reinforcement*, which performs reinforcement only on the last input-output pair. We now introduce a theorem which is the theoretical basis of the gradient estimation in REINFORCE.

4.1 Gradient Estimation with The Log-Derivative Trick

The Log-Derivative trick allows us to estimate the gradient of the expected value of reward.

Theorem 4.1 (Log-Derivative Trick [21]). Let $p(X;\theta)$ be a probability distribution depending on a set of parameters θ, and assume $f(x)$ has a finite expectation. Then it holds that

$$\nabla_\theta \mathbb{E}_{p(X;\theta)}[f(X)] = \mathbb{E}_{p(X;\theta)}[\nabla_\theta f(X)] + \mathbb{E}_{p(X;\theta)}[f(X)\nabla_\theta \log(p(X;\theta))]. \tag{3}$$

Proof. To take the gradient of the logarithm of any probability density function $p(X;\theta)$, we can use the chain rule and do some rearranging. This is known as the Log-Derivative trick.

$$\nabla_\theta \log(p(X;\theta)) = \frac{\nabla_\theta p(X;\theta)}{p(X;\theta)} \tag{4}$$

$$\nabla_\theta p(X;\theta) = p(X;\theta)\nabla_\theta \log(p(X;\theta)) \tag{5}$$

We can write the original expectation as an integral and apply the product rule for differentiation.

$$\nabla_\theta \mathbb{E}_{p(X;\theta)}[f(X)] = \nabla_\theta \int p(X;\theta) f(x) dx \tag{6}$$

$$= \int \nabla_\theta (p(X;\theta) f(x)) dx \tag{7}$$

$$= \int (p(X;\theta)\nabla_\theta f(x) + f(x)\nabla_\theta p(X;\theta)) dx \tag{8}$$

$$= \int p(X;\theta)\nabla_\theta f(x) dx + \int f(x)\nabla_\theta p(X;\theta) dx \tag{9}$$

Using the Log-Derivative trick in the second term of Equation 9, we obtain the desired expression:

$$\nabla_\theta \mathbb{E}_{p(X;\theta)}[f(X)] = \int p(X;\theta)\nabla_\theta f(x) dx + \int f(x) p(X;\theta)\nabla_\theta \log(p(X;\theta)) dx \tag{10}$$

$$= \mathbb{E}_{p(X;\theta)}[\nabla_\theta f(X)] + \mathbb{E}_{p(X;\theta)}[f(X)\nabla_\theta \log(p(X;\theta))]. \tag{11}$$

□

Since both expectations in Equation 3 are usually intractable, they are usually estimated with the following Monte Carlo scheme:

$$\nabla_\theta \mathbb{E}_{p(X;\theta)}[f(X)] \approx \frac{1}{N}\sum_{n=1}^{N} (\nabla_\theta f(x^{(n)}) + f(x^{(n)})\nabla_\theta \log(p(x^{(n)}))). \tag{12}$$

4.2 Learning with REINFORCE

To demonstrate how this estimator works in practice, we continue with our `sum2` example and show how sampled symbols and outputs contribute to the gradient estimation. For the sake of simplicity, suppose that we restrict the inputs digits of `sum2` to be between 0 and 2. This means that the possible space of outputs is between 0 and 4. Suppose that the ground truth inputs are 1 and 2, i.e, $r_1 = 1, r_2 = 2$, and the ground truth output is $y = 3$. Suppose that the predicted distributions from M_θ for r_1 and r_2 are $[0.1, 0.6, 0.3]$ and $[0.2, 0.1, 0.7]$ respectively.

Suppose that we initialize REINFORCE to use a use 3 samples, and the sampled symbol-output pairs are $((1, 2), 3)$, $((1, 0), 1)$, $((2, 1), 3)$. REINFORCE uses the Monte Carlo estimation from Equation 12 to compute an estimate of the gradient of the expected reward. It does this by performing element-wise multiplication on the log probability of each sample with its reward value and taking the mean:

$$\nabla \mathbb{E}_{p(X;\theta)}[f(X)] \approx \frac{1}{3} * \begin{bmatrix} \log(0.6) + \log(0.7) \\ \log(0.6) + \log(0.2) \\ \log(0.3) + \log(0.1) \end{bmatrix} * \begin{bmatrix} 1.0 \\ 0.0 \\ 1.0 \end{bmatrix}$$

REINFORCE uses this estimated gradient to update the weights of M_θ in the direction that maximizes expected reward. This solution offers a more general approach to the neural program learning problem compared to neurosymbolic solutions, but it is not scalable and struggles with high variance, which we discuss further in the following section.

5 LOW-VARIANCE GRADIENT ESTIMATION WITH INDECATER

While REINFORCE can achieve high accuracy on tasks involving a small number of inputs, it does not scale to high-dimensional input spaces. This is because REINFORCE produces a high-variance estimate: for tasks involving large input spaces, randomly sampled inputs are less likely to result in the correct output, and reward signals rely on getting the correct output. Increasing the number of samples helps with this problem, but can lead to exponential blowup as the size of the input space increases. IndeCateR [16] is an unbiased gradient estimator that aims to address this limitation of REINFORCE. IndeCateR uses the *CatLog-Derivative trick*, a variation of the Log-Derivative trick that is tailored towards categorical distributions, to achieve provably lower variance than REINFORCE.

5.1 CatLog-Derivative Trick

We start by introducing the CatLog-Derivative trick which aims to reduce the exponential number of states that would result from using REINFORCE on a multivariate categorical distribution.

Theorem 5.1 (CatLog-Derivative Trick [16]) Let $p(X;\theta)$ be a multivariate categorical probability distribution (depending on parameters θ), and assume $\mathbb{E}_{p(X;\theta)}$ is finite. Additionally, $\Omega(X)$ represents the finite sample space of random vector X. Then it holds that

$$\nabla_\theta \mathbb{E}_{p(X;\theta)}[f(X)] = \sum_{d=1}^{D} \sum_{x_\delta \in \Omega(X_d)} \mathbb{E}_{p(X_{<d};\theta)}[\nabla_\theta p(x_\delta | X_{<d}) \mathbb{E}_{p(X_{>d}|x_\delta, X_{<d};\theta)}[f(X_{\neq d}, x_\delta)]]. \tag{13}$$

Proof. We start by applying the Log-Derivative trick. Next, we replace the categorical distribution $P(X;\theta)$ with its product form. We then rewrite the log of products as the sum of log

terms.

$$\nabla_\theta \mathbb{E}_{p(X;\theta)}[f(X)] = \mathbb{E}_{p(X;\theta)}[f(X)\nabla_\theta \log(p(X;\theta))] \tag{14}$$

$$= \mathbb{E}_{p(X;\theta)}[f(X)\nabla_\theta \log(\prod_{d=1}^{D} p(X_d|X_{>d});\theta)] \tag{15}$$

$$= \sum_{d=1}^{D} \mathbb{E}_{p(X;\theta)}[f(X)\nabla_\theta \log(p(X_d|X_{<d});\theta)] \tag{16}$$

We write the expectation term as a sum, separating the expressions for X_d and $X_{\neq d}$.

$$\sum_{d=1}^{D} \sum_{x\in\Omega(X_{\neq d})} \sum_{x_\delta\in\Omega(X_d)} p(X_{\neq d}, x_\delta;\theta) f(X_{\neq d}, x_\delta)\nabla_\theta \log p(x_\delta|X_{<d};\theta) \tag{17}$$

We then factorize the joint probability $p(X_{\neq d}, x_\delta;\theta)$ as $p(X_{>d}|x_\delta, X_{<d};\theta)p(x_\delta|X_{<d};\theta)p(x_{<d};\theta)$.

$$\sum_{d=1}^{D} \sum_{x\in\Omega(X_{\neq d})} \sum_{x_\delta\in\Omega(X_d)} p(X_{>d}|x_\delta, X_{<d};\theta)p(x_\delta|X_{<d};\theta)p(X_{<d};\theta) f(X_{\neq d}, x_\delta)\nabla_\theta \log p(x_\delta|X_{<d};\theta) \tag{18}$$

Using the Log-Derivative trick again, multiplying $p(x_\delta|X_{<d};\theta)$ with $\nabla_\theta \log(p(x_\delta|X_{<d};\theta))$ gives us $\nabla_\theta p(x_\delta|x_{<d};\theta)$.

$$\sum_{d=1}^{D} \sum_{x\in\Omega(X_{\neq d})} \sum_{x_\delta\in\Omega(X_d)} \nabla_\theta p(x_\delta|X_{<d};\theta)p(X_{>d}|x_\delta, X_{<d};\theta)p(X_{<d}) f(X_{\neq d}, x_\delta) \tag{19}$$

Finally, writing the middle sum as an expectation yields the desired expression.

$$\sum_{d=1}^{D} \sum_{x_\delta\in\Omega(X_d)} \mathbb{E}_{p(X_{<d};\theta)}[\nabla_\theta p(x_\delta|X_{<d})\mathbb{E}_{p(X_{>d}|x_\delta,X_{<d};\theta)}[f(X_{\neq d}, x_\delta)]] \tag{20}$$

□

With this, we can now define the IndeCateR gradient estimator that has provably lower variance than REINFORCE.

5.2 IndeCateR Gradient Estimator

Proposition 5.1 (IndeCateR [16]). Let $p(X;\theta)$ be a multivariate categorical probability distribution (depending on parameters θ) which has factorization $p(X) = \prod_{d=1}^{D} p_d(X_d;\theta)$. Also assume that $\mathbb{E}_{p(X;\theta)}$ is finite. Then we can estimate $\mathbb{E}_{p(X;\theta)}[f(X)]$ with

$$\sum_{d=1}^{D} \sum_{x_\delta\in\Omega(X_d)} \nabla_\theta p_d(x_\delta;\theta)\frac{1}{N}\sum_{n_d=1}^{N} f(x_{\neq d}^{(n_d)}, x_\delta) \tag{21}$$

where $x_{\neq d}^{(n_d)}$ samples are drawn from $p(X_{\neq d};\theta)$.

Proof. Starting with Equation 13, we can use the fact that we have an independent set of random variables and simplify $p(x_\delta|X_{<d};\theta)$ to $p_d(x_\delta;\theta)$. Thus, we can rewrite the gradient of the expected

value of $f(X)$ as

$$\nabla_\theta \mathbb{E}_{p(X;\theta)}[f(X)] = \sum_{d=1}^{D} \sum_{x_\delta \in \Omega(X_d)} \mathbb{E}_{p(X_{<d};\theta)}[\nabla_\theta p_d(x_\delta;\theta)\mathbb{E}_{p(X_{>d};\theta)}[f(X_{\neq d}, x_\delta)]] \tag{22}$$

$$= \sum_{d=1}^{D} \sum_{x_\delta \in \Omega(X_d)} \nabla_\theta p_d(x_\delta;\theta)\mathbb{E}_{p(X_{\neq d};\theta)}[f(X_{\neq d}, x_\delta)]. \tag{23}$$

Assuming we draw N samples from $D-1$ independent random distributions $X_{\neq d}$, then we obtain the result

$$\sum_{d=1}^{D} \sum_{x_\delta \in \Omega(X_d)} \nabla_\theta p_d(x_\delta;\theta) \frac{1}{N} \sum_{n_d=1}^{N} f(x_{\neq d}^{(n_d)}, x_\delta) \tag{24}$$

as required. With this gradient estimator, we now look at an example of a multivariate distribution with independent categorical random variables that would benefit from the lower variance that the IndeCateR gradient estimator provides compared to REINFORCE.

Example 5.1 As a simple example, suppose that we have a multivariate distribution involving 3-ary independent categorical random variables:

$$p(X_1, X_2, X_3) = p_1(X_1)p_2(X_2)p_3(X_3) \tag{25}$$

These categorical random variables might represent MNIST digits that we want to take the sum of. For simplicity, let's assume that we restrict the MNIST digits to the range 1-3. This means that X_1, X_2, and X_3 can take values from the set $\Omega(X) = \{1, 2, 3\}$ Using these values in the gradient estimator from Equation 21, we obtain

$$\sum_{d=1}^{3} \sum_{x_\delta \in \{0,1,2\}} \nabla_\theta p_d(x_\delta;\theta) \frac{1}{N} \sum_{n_d=1}^{N} f(x_{\neq d}^{(n_d)}, x_\delta). \tag{26}$$

With this, for $d = 1$ we consider the single sample REINFORCE estimate

$$\nabla_\theta p_1(x_1) f(x_1, x_2, x_3) \tag{27}$$

compared to the IndeCateR estimate

$$\nabla_\theta p_1(1) f(1, x_2, x_3) + \nabla_\theta p_2(2) f(2, x_2, x_3) + \nabla_\theta p_3(3) f(3, x_2, x_3) \tag{28}$$

where each x_1, x_2, and x_3 term represents a sampled value for the respective random variable. The difference between these techniques is that REINFORCE samples all variables whereas IndeCateR performs the sum over each of the random variables and only samples the remaining random variables. Performing the sum in this way leads to lower variance gradient estimates compared to REINFORCE. Consequently, IndeCateR is able to scale to higher dimensional input spaces compared to REINFORCE, which we discuss further in §9.

□

6 FINE-TUNING NEURAL PROGRAMS WITH NASR

Neural Attention for Symbolic Reasoning (NASR) [4] is a REINFORCE-based framework that is suitable for the neural program learning setting with black-box program components. NASR targets tasks that exploit domain knowledge constraints such as Sudoku solving. In the problem of Sudoku solving, the neural component is divided into two steps: a neuro-solver and a mask predictor. The neuro-solver is given a board with entries being blank or containing images of handwritten digits. The neuro-solver predicts the board's solution, but it is possible that is makes errors, i.e, predictions that violate the rules of Sudoku. To fix these violations, the output of the neuro-solver is fed into the

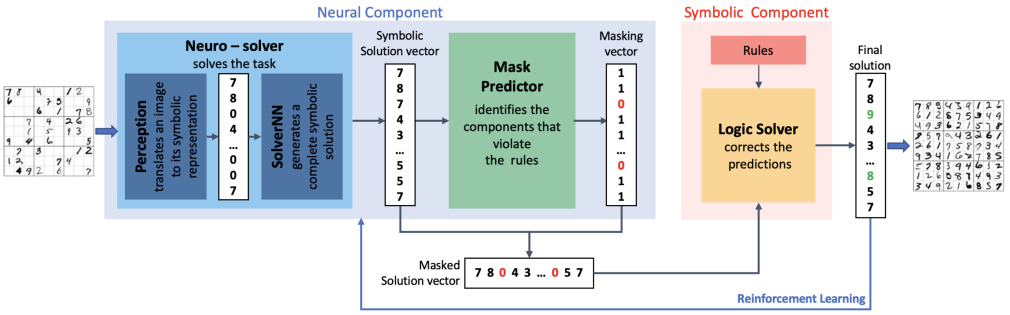

Fig. 2. A depiction of the NASR training pipeline for Sudoku solving [4]. The neural component involves 1) a neuro-solver which predicts a solution to an incomplete Sudoku board and 2) a mask predictor to identify components that violate the rules of Sudoku. The symbolic component is a Prolog solver that takes the output of the mask predictor and completes the board's solution.

mask predictor that identifies rule violations and masks them out. The output of the mask predictor is then fed into a Sudoku logic solver (written in Prolog), which completes the board (Fig. 2).

6.1 NASR Learning Algorithm

NASR works best for tasks involving pre-trained neural components where fine-tuning is the goal. For the Sudoku solving task, the neuro-solver is pre-trained using the image-label pairs from the MNIST dataset [7]. The mask predictor is trained on a synthetic dataset of Sudoku boards, where noise is added to each board to violate some constraints.

NASR can use these pre-trained components, along with a logic solver, to complete the training pipeline that uses REINFORCE for fine-tuning. REINFORCE is necessary for training because the logic solver is treated as black-box, so the entire pipeline is not end-to-end differentiable as it would be in neurosymbolic frameworks. It is important to note that the logic solver *could* be treated as a white-box with Scallop or DPL used training, since the logic solver is already written in Prolog and could be written in Datalog. However, training would be prohibitively expensive in either framework due to the complexity of the solver (see §9 for further details). Therefore, training is more efficient when the logic solver is treated as black-box and REINFORCE is employed.

NASR uses the REINFORCE algorithm with a standard policy loss: $\mathcal{L}(x;\theta) = -r \log(P_\theta(m|ns(x)))$, where r is the reward obtained after applying the logic solver to the prediction $ns(x)$ masked by m. The reward calculation (Equation 29) is comprised of two values: the entire reward and the cell reward. The entire reward r_e is equal to 10 if the output board b' is exactly equal to the ground truth board b. The cell reward is equal to the proportion of correct cells in the output board. There is no negative reward assigned for incorrect predictions.

$$r = r_e + r_c = 10 \cdot \delta_{b',b} + \frac{1}{81} \sum_{i=0}^{81} \delta_{b'_i,b_i} \tag{29}$$

NASR has demonstrated high accuracy on Sudoku solving (§9) and has also been applied to the task of predicate classification in which the goal is to predict the correct predicate between two objects in an image (e.g., "in front," "behind," "wearing," etc.). However, NASR is limited in its ability to apply to tasks where no intermediate labels are available for pre-training. This is because the algorithm was formulated to use REINFORCE with only a single sample during fine-tuning, which is rarely sufficient to train even simple tasks with randomly initialized network weights. We elaborate on this limitation in §9.

7 SCALABLE NEURAL PROGRAM LEARNING WITH A-NESI

Approximate Neurosymbolic Inference (A-NeSI) [19] is scalable learning framework that is well-suited for learning neural programs. The primary limitation of neurosymbolic learning frameworks operating over probabilistic logics is that the weighted-model counting procedure is computationally exponential [2]. A-NeSI was developed to address this limitation, introducing a solution that approximates the weighted model counting procedure by adding two additional neural networks to the training pipeline.

7.1 Preliminaries

The A-NeSI inference pipeline includes three neural networks: 1) a network that performs inference over the unstructured data, 2) a network that represents the *belief prior*, which we describe in greater detail in §7.3, and 3) a network that approximates the output of the WMC procedure. We introduce four sets representing the spaces of the variables of interest.

(1) X is the space of inputs. For example, X would represent pairs of MNIST digits for `sum2`.
(2) W is the space of structured inputs representing the k_W discrete variables w_i. We call the elements $w \in W = W_1 \times W_2 \times \ldots W_{k_W}$ *worlds* or *concepts* of some $x \in X$.
(3) Y is the space of structured outputs representing the k_Y discrete variables y_i. Each possible output of the pipeline is represented by an element $y \in Y = Y_1 \times \ldots Y_{k_Y}$. For `sum2`, outputs would be decomposed by digit, e.g., $y = (1, 2) \in \{0, 1\} \times \{0, \ldots, 8\}$ represents output 12.
(4) We call $P \in \delta^{|W_1|} \times \cdots \times \delta^{|W_{k_W}|}$ a *belief*, where each $\delta^{|W_i|}$ is the probability simplex over the options of the variable w_i. P assigns probabilities to different worlds, i.e., $p(w|P) = \prod_{i=1}^{k_W} P_{i,w_i}$. This means P is a parameter for an independent categorical distribution over the k_W discrete variables w_i.

With these definitions, we introduce the three quantities that A-NeSI is interested in efficiently computing. Note that we call the perception model M_θ.

(1) We are interested in computing the probability of outputs for the belief P that that M_θ computes on input x:

$$p(y|P = M_\theta(x))$$

(2) In order to train M_θ, we would like to compute the gradient of the WMC problem:

$$\nabla_P p(y|P = M_\theta(x))$$

(3) We are also interested in inferring likely worlds given a predicted output and a belief about the predicted digits:

$$p(w|y, P = M_\theta(x))$$

Since the WMC procedure is #P-hard [15], we need a way to approximate these quantities in order to learn efficiently. We now describe the A-NeSI architecture that is used in this approximation.

7.2 A-NeSI Architecture

A-NeSI involves a *prediction model* $q_\phi(y|P)$, which serves to approximate the WMC procedure, and an *explanation model* $q_\psi(w|y, P)$, which predicts likely worlds given certain outputs and beliefs. Together, these models form the *inference model* $q_{\phi,\psi}$ defined as

$$q_{\phi,\psi}(w, y|P) = q_\phi(y|P)q_\psi(w|y, P) \tag{30}$$

We focus on the *prediction-only* variant of A-NeSI which trains the prediction model. We first model $q_\phi(y|P)$ by defining an autoregressive generative model over Y. We represent Y as a sequence: if Y

is the space of multi-digit outputs of a multi-digit MNIST addition, then we would decompose each output into its individual digits.

$$q_\phi(y|P) = \prod_{i=1}^{k_Y} q_\phi(y_i|y_{1:i-1}, P) \tag{31}$$

The prediction model can be used to train the perception network M_θ, where $x, y \sim \mathcal{D}$.

$$\mathcal{L}_{Perc}(\mathcal{D}, \theta) = -\log(q_\phi(y|P = M_\theta(x))) \tag{32}$$

To train the prediction model, we want to minimize the expected cross entropy between $p(y|P)$ and $q_\phi(y|P)$ over the prior $p(P)$, where $P, w \sim p(P, w)$.

$$\mathcal{L}_{Pred}(\phi) = \log(q_\phi(c(w)|P)) \tag{33}$$

7.3 Designing the Belief Prior

The design of A-NeSI's belief prior is one reason why the framework is able to scale to high-dimensional input spaces. A naive approach would sample $(x_1, y_1), \ldots (x_k, y_k) \sim \mathcal{D}$ from the training data, and train the inference model using $P_1 = M_\theta(x_1), \ldots, P_k = M_\theta(x_k)$. However, this approach would mean only training over the data that exist in $\mathcal{D}$, which isn't sufficient for tasks with a large input space as only a small subset of the possible input combinations exist in the dataset. For example, in the task of adding two N-digit MNIST numbers, there are $2 \cdot 10^N - 1$ possible outputs. When N is large, it is impractical for $\mathcal{D}$ to include all possibilities.

A-NeSI takes a different approach by fitting a Dirichlet prior $p(P)$ on $P_1, \ldots, P_k$ that covers all possible combinations of numbers. This prior design allows A-NeSI to simulate more beliefs during training, which in turn allows the inference model to generalize better as it has seen more combinations of inputs and outputs. This is one of the contributing factors in A-NeSI's ability to scale to tasks such as 15-digit MNIST addition.

7.4 A-NeSI Learning Algorithm

We express the A-NeSI training loop as pseudocode in Algorithm 1. We define two procedures: `inference_model_loss` and `training_loop`. The former procedure first fits the prior and then obtains the probability of each world given inputs sampled from the prior. It then computes outputs and returns the updated ϕ parameters from minimizing the loss. The latter procedure trains the inference and prediction models until convergence. In each training iteration, we take the output of the prediction model and update the beliefs with that prediction. These beliefs are fed into `inference_model_loss`, which returns optimized ϕ parameters (for the prediction model), which are in turn used to optimize θ parameters (for the perception network).

As previously mentioned, this learning algorithm scales well to tasks with high dimensional input spaces such as 15-digit addition. However, in §9 we discuss how training of additional neural networks can result in data inefficiency. Furthermore, training a neural network to approximate the weighted-model counting problem becomes difficult when the black-box program is complex, such as in the tasks of Sudoku solving and hand-written formula evaluation. The next section introduces a framework, called ISED, which aims to address some of the limitations of REINFORCE-based and neural gradient approximation techniques.

8 DATA-EFFICIENT NEURAL PROGRAM LEARNING WITH ISED

ISED (Infer-Sample-Estimate-Descend) [17] aims to improve upon some of the limitations of other black-box gradient estimation methods, namely data and sample inefficiency. ISED uses outputs of M_θ as a probability distribution over inputs of P and samples representative symbols u from this

Algorithm 1 A-NeSI Training loop

procedure INFERENCE_MODEL_LOSS($P_1, \ldots, P_k, \phi$)
  fit prior P on $P_1, \ldots P_k$
  $P \sim p(P)$
  $w \sim p(w|P)$
  $y \leftarrow c(w)$
  return $\phi + \alpha \nabla_\phi \log(q_\phi(y|P))$
end procedure
procedure TRAINING_LOOP($\mathcal{D}, \theta, \phi$)
  Input: dataset $\mathcal{D}$, params θ, params ϕ
  `beliefs` $\leftarrow$ []
  while not converged **do**
    $(x, y) \sim \mathcal{D}$
    $P \leftarrow M_\theta(x)$
    update `beliefs` with P
    $\phi \leftarrow$ `inference_model_loss(beliefs,` ϕ`)`
    $\theta \leftarrow \theta + \alpha \nabla_\theta \log(q_\phi(y|P))$
  end while
end procedure

distribution. ISED then computes outputs v of P corresponding to these symbols. The resulting symbol-output pairs can be viewed as a symbolic program consisting of clauses of the form `if symbol` $= u$ `then output` $= v$ summarizing P. For instance, in the task of adding two digits r_1 and r_2, one rule of the logic program would be $r_1 = 1 \wedge r_2 = 2 \rightarrow y = 3$. ISED is then able to differentiate through this summary to facilitate learning, which we describe in more detail in §8.2.

8.1 Preliminaries and Programming Interface

ISED allows programmers to write black-box programs that operate on diverse structured inputs and outputs. To allow such programs to interact with neural networks, we define an interface named *structural mapping*. This interface serves to 1) define the data-types of black-box programs' input and output, 2) marshall and un-marshall data between neural networks and logical black-box functions, and 3) define the loss. We define a *structural mapping* τ as either a discrete mapping (with Σ being the set of all possible elements), a floating point, a permutation mapping with n possible elements, a tuple of mappings, or a list of up to n elements. We define τ inductively as follows:

$$\tau ::= \textsc{Discrete}(\Sigma) \mid \textsc{Float} \mid \textsc{Permutation}_n \mid \textsc{Tuple}(\tau_1, \ldots, \tau_m) \mid \textsc{List}_n(\tau)$$

Using this, we may further define data-types such as $\textsc{Integer}_j^k = \textsc{Discrete}(\{j, \ldots, k\})$, $\textsc{Digit} = \textsc{Integer}_0^9$, and $\textsc{Bool} = \textsc{Discrete}(\{\text{true}, \text{false}\})$.

We also define a *black-box program* P as a function $(\tau_1, \ldots, \tau_m) \rightarrow \tau_o$, where $\tau_1, \ldots, \tau_m$ are the input types and τ_o is the output type. We use the hand-written formula (HWF) task as an example throughout this section, where the goal is to predict a list of hand-written symbols and evaluate the result of the formula. For the HWF task, the structural input mapping is $\textsc{List}_7(\textsc{Discrete}(\{0, \ldots, 9, +, -, \times, \div\}))$, and the structural output mapping is $\textsc{Float}$. The mapping suggests that the program takes a list of length up to 7 as input, where each element is a digit or an arithmetic operator, and returns a floating point number.

There are two interpretations of a structural mapping: the set interpretation $\text{SET}(\tau)$ represents a mapping with defined values, e.g., a digit with value 8; the tensor interpretation $\text{DIST}(\tau)$ represents

a mapping where each value is associated with a probability distribution, e.g., a digit that is 1 with probability 0.6 and 7 with probability 0.4. We use the set interpretation to represent structured program inputs that can be passed to a black-box program and the tensor interpretation to represent probability distributions for unstructured data and program outputs. These two interpretations are defined for the different structural mappings in Table 2.

Table 2. Set and tensor interpretations of different structural mappings.

Mapping (τ)	**Set Interpretation** (SET(τ))	**Tensor Interpretation** (DIST(τ))
$\textsc{Discrete}_\Sigma$	Σ	$\{\vec{v} \mid \vec{v} \in \mathbb{R}^{\lvert\Sigma\rvert}, v_i \in [0,1], i \in 1 \ldots \lvert\Sigma\rvert\}$
$\textsc{Float}$	$\mathbb{R}$	n/a
$\textsc{Permutation}_n$	$\{\rho \mid \rho \text{ is a permutation of } [1,...,n]\}$	$\{[\vec{v_1},...,\vec{v_n}] \mid \vec{v_i} \in \mathbb{R}^n, v_{i,j} \in [0,1], i \in 1 \ldots n\}$
$\textsc{Tuple}(\tau_1,\ldots,\tau_m)$	$\{(a_1,...,a_m) \mid a_i \in \text{SET}(\tau_i)\}$	$\{(a_1,..,a_m) \mid a_i \in \text{DIST}(\tau_i)\}$
$\textsc{List}_n(\tau')$	$\{[a_1,...,a_j] \mid j \leq n, a_i \in \text{SET}(\tau')\}$	$\{[a_1,..,a_j] \mid j \leq n, a_i \in \text{DIST}(\tau')\}$

In order to represent the ground truth output as a distribution to be used in the loss function, there needs to be a mechanism for transforming SET(τ) mappings into DIST(τ) mappings. For this purpose, we define a *vectorize* function $\delta_\tau : (\text{SET}(\tau), 2^\tau) \rightarrow \text{DIST}(\tau)$ for the different output mappings τ in Table 3. When considering a datapoint (x, y) during training, ISED samples many symbols and obtains a list of outputs $\hat{y}$. The vectorizer then takes the ground truth y and the outputs $\hat{y}$ as input and returns the equivalent distribution interpretation of y. While $\hat{y}$ is not used by δ_τ in most cases, we include it as an argument so that $\textsc{Float}$ output mappings can be discretized, which is necessary for vectorization. For example, if the inputs to the vectorizer for the hand-written formula task are $y = 2.0$ and $\hat{y} = [1.0, 3.5, 2.0, 8.0]$, then it would return $[0, 0, 1, 0]$.

Table 3. Vectorize and aggregate functions of different structural mappings.

Mapping (τ)	**Vectorizer** ($\delta_\tau(y, \hat{y})$)	**Aggregator** ($\sigma_\tau(\hat{r}, \hat{p})$)
$\textsc{Discrete}_n$	$\boldsymbol{e}^{(y)}$ with dim n	$\hat{p}[\hat{r}]$
$\textsc{Float}$	$[\mathbf{1}_{y=\hat{y}_i} \text{ for } i \in [1,\ldots,\text{length}(\hat{y})]]$	n/a
$\textsc{Permutation}_n$	$[\delta_{\textsc{Discrete}_n}(y[i]) \text{ for } i \in [1,\ldots,n]]$	$\otimes_{i=1}^n \sigma_{\textsc{Discrete}_n}(\hat{r}[i], \hat{p}[i])$
$\textsc{Tuple}(\tau_1,\ldots,\tau_m)$	$[\delta_{\tau_i}(y[i]) \text{ for } i \in [1,\ldots,m]]$	$\otimes_{i=1}^m \sigma_{\tau_i}(\hat{r}[i], \hat{p}[i])$
$\textsc{List}_n(\tau')$	$[\delta_{\tau'}(a_i) \text{ for } a_i \in y]$	$\otimes_{i=1}^n \sigma_{\tau'}(\hat{r}[i], \hat{p}[i])$

We also require a mechanism to aggregate the probabilities of sampled symbols that resulted in a particular output. With this aim, we define an *aggregate* function $\sigma_\tau : (\text{SET}(\tau), \text{DIST}(\tau)) \rightarrow \mathbb{R}$ for different input mappings τ in Table 3. ISED aggregates probabilities either by taking their minimum or their product, and we denote both operations by $\otimes$. The aggregator takes as input sampled symbols $\hat{r}$ and neural predictions $\hat{p}$ from which $\hat{r}$ was sampled. It gathers values in $\hat{p}$ at each index in $\hat{r}$ and returns the result of $\otimes$ applied to these values. For example, suppose we use `min` as the aggregator $\otimes$ for the hand-written formula task. Then if $\otimes$ takes $\hat{r} = [1, +, 1]$ and $\hat{p}$ as inputs where $\hat{p}[0][1] = 0.1$, $\hat{p}[1][+] = 0.05$, and $\hat{p}[2][1] = 0.1$, it would return 0.05.

8.2 ISED Algorithm

We now formally present the ISED algorithm. For a given task, there is a black-box program P, taking m inputs, that operates on structured data. Let $\tau_1, ..., \tau_m$ be the mappings for these inputs and τ_o the mapping for the program's output. We write P as a function from its input mappings to its output mapping: $P : (\tau_1, ..., \tau_m) \rightarrow \tau_o$. For each unstructured input i to the program, there

Algorithm 2 ISED training pipeline

Require: P is the black-box program $(\tau_1, \ldots, \tau_m) \to \tau_o$, $M^i_{\theta_i}$ the neural model $x_i \to \text{DIST}(\tau_i)$ for each τ_i, S the sampling strategy, k the sample count, $\mathcal{L}$ the loss function, and $\mathcal{D}$ the dataset.

1: **procedure** TRAIN
2: **for** $((x_1, \ldots x_m), y) \in \mathcal{D}$ **do**
3: **for** $i \in 1 \ldots m$ **do**
4: $\hat{p}[i] \leftarrow M^i_{\theta_i}(x_i)$ ▷ **Infer**
5: **end for**
6: **for** $j \in 1 \ldots k$ **do**
7: **for** $i \in 1 \ldots m$ **do**
8: Sample $\hat{r}_j[i]$ from $\hat{p}[i]$ using S ▷ **Sample**
9: **end for**
10: $\hat{y}_j \leftarrow P(\hat{r}_j)$
11: **end for**
12: $\hat{w} \leftarrow \text{normalize}([\omega(y_k, \hat{y}, \hat{r}, \hat{p}) \text{ for } y_k \in \tau_o \text{ (or } y_k \in \hat{y})])$ ▷ **Estimate**
13: $w \leftarrow \delta(y, \hat{y})$
14: $l \leftarrow \mathcal{L}(\hat{w}, w)$
15: Compute $\frac{\partial l}{\partial \theta}$ by performing back-propagation on l
16: Optimize θ based on $\frac{\partial l}{\partial \theta}$ ▷ **Descend**
17: **end for**
18: **end procedure**

is a neural model $M^i_{\theta_i} : x_i \to \text{DIST}(\tau_i)$. S is a sampling strategy (e.g., categorical sampling) that samples symbols using the outputs of a neural model, and k is the number of samples to take for each training example. There is also a loss function $\mathcal{L}$ whose first and second arguments are the prediction and target values respectively. We describe the four steps of ISED using the hand-written formula task.

Infer. The training pipeline starts with an example from the dataset, $(x, y) = ([\image{1}, \image{+}, \image{2}], 3.0)$, and uses a CNN to predict these images, as shown on lines 3-4. ISED initializes $\hat{p} = M_\theta(x)$.

Sample. ISED samples $\hat{r}$ from $\hat{p}$ for k iterations using sampling strategy S. For each sample j, the algorithm initializes $\hat{r}_j$ to be the sampled symbols, as shown on lines 6-9. To continue our example, suppose ISED initializes $\hat{r}_j = [7, +, 2]$ for sample j. The next step is to execute the program on $\hat{r}_j$, as shown on line 10, which in this example means setting $\hat{y}_j = P(\hat{r}_j) = 9.0$.

Estimate. In order to compute the prediction value to use in the loss function, ISED must consider each output y_k in the output mapping and accumulate the aggregated probabilities for all sampled symbols that resulted in output y_k. We specify $\otimes$ as the `min` function, and $\oplus$ as the `max` function in this example. Note that ISED requires that $\otimes$ and $\oplus$ represent either `min` and `max` or `mult` and `add` respectively. We refer to these two options as the `min-max` and `add-mult` semirings. We define an *accumulate* function ω that takes as input an element of the output mapping y_k, sampled outputs $\hat{y}$, sampled symbols $\hat{r}$, and predicted input distributions $\hat{p}$. The accumulator performs the $\oplus$ operation on aggregated probabilities for elements of $\hat{y}$ that are equal to y_k and is defined as follows:

$$\omega(y_k, \hat{y}, \hat{r}, \hat{p}) = \oplus_{j=1}^{k} \mathbf{1}_{\hat{y}_j = y_k} \sigma_{\tau_o}(\hat{r}_j, \hat{p}_j)$$

ISED then sets $\tilde{w} = [\omega(y_k, \hat{y}, \hat{r}, \hat{p}) \text{ for } y_k \in \tau_o]$ in the case where τ_o is not Float. When τ_o is Float, as for hand-written formula, it only considers $y_k \in \hat{y}$. Next, it performs L_2 normalization over each element in $\tilde{w}$ and sets $\hat{w}$ to this result. To initialize the ground truth vector, it sets $w = \delta(y, \hat{y})$. ISED

then initializes $l = \mathcal{L}(\hat{w}, w)$ and computes $\frac{\partial l}{\partial \theta_i}$ for each input i. These steps are shown on lines 12-15.

Descend. The last step is shown on line 16, where the algorithm optimizes θ_i for each input i based on $\frac{\partial l}{\partial \theta_i}$ using a stochastic optimizer (e.g., Adam optimizer). This completes the training pipeline for one example, and the algorithm returns all final θ_i after iterating through the entire dataset.

Now that we have presented the ISED algorithm, we briefly summarize some of its strengths and weaknesses. ISED improves upon neurosymbolic learning frameworks in that it is applicable to many different input and output types, and it is compatible with arbitrary black-box programs, including those that make calls to LLMs. Furthermore, as we explain in §9, ISED addresses the limitations of other black-box gradient approximation techniques, namely data and sample efficiency. However, the main limitation of ISED is its inability to scale to high-dimensional input spaces.

9 EVALUATION

9.1 Benchmark Tasks: NeuroGPT, NeuroPython, and Neurosymbolic

In this section, we aim to answer the following research questions:

RQ1: How do the different approaches compare in terms of accuracy?

RQ2: How do the REINFORCE-based techniques compare in terms of sample efficiency?

RQ3: How do black-box gradient estimation techniques compare in terms of data efficiency?

We evaluate the presented techniques on a diverse benchmark suite to compare their performance. In our benchmark suite, we include two neural program learning benchmarks that contain a program component that can make calls to GPT-4. We call such models neuroGPT programs.

Leaf Classification. In this task, we use a dataset containing leaf images from 11 different plant species [3], containing 330 training samples and 110 testing samples. We define custom Discrete types representing the Margin, Shape, and Texture leaf features. The output type is a Discrete type. Neural program solutions either prompt GPT-4 (GPT leaf) or use a decision tree (DT leaf).

Scene Recognition. We use a dataset containing scene images from 9 different room types [12], containing 830 training samples and 92 testing samples. We define custom types Objects and Scenes to be a Discrete set of 45 objects and 9 room types, respectively. To predict objects, we use an off-the-shelf object detection model YOLOv8 [14], and fine-tune only the custom neural network. The neural program solution constructs a prompt using detected objects and calls GPT-4 to classify the scene.

We also consider several tasks from the neurosymbolic literature, including Sudoku solving and hand-written formula (HWF) evaluation. While the solutions to many of these tasks are usually presented as a logic program in neurosymbolic learning frameworks, neural program solutions can take the form of Python programs. We call such models neuroPython programs.

MNIST-R. MNIST-R [9, 10] contains 11 tasks operating on inputs of images of handwritten digits from the MNIST dataset [7]. This synthetic test suite includes tasks performing arithmetic (sum_2, sum_3, sum_4, mult_2, mod_2, add-mod-3, add-sub), comparison (less-than, equal), counting (count-3-or-4), and negation (not-3-or-4) over the digits depicted in the images. Each task dataset has a training set of 5K samples and a testing set of 500 samples.

HWF. The goal of the HWF task is to classify images of handwritten digits and arithmetic operators and evaluate the formula [8]. The dataset contains 10K formulas of length 1-7, with 1K length 1 formulas, 1K length 3 formulas, 2K length 5 formulas, and 6K length 7 formulas.

Visual Sudoku. The goal of this task is to predict the solution of a 9x9 Sudoku board where the incomplete board contains MNIST digits. We use the same experimental setting used in NASR,

Table 4. Performance comparison for DT leaf, GPT leaf, scene, and sudoku.

	Accuracy (%)			
Method	DT leaf	GPT leaf	scene	sudoku
DPL	39.70 ± 6.55	N/A	N/A	TO
Scallop	81.13 ± 3.50	N/A	N/A	TO
A-NeSI	78.82 ± 4.42	72.40 ± 12.24	61.46 ± 14.18	26.36 ± 12.68
REINFORCE	23.60 ± 8.27	34.02 ± 12.71	47.07 ± 14.78	79.08 ± 0.87
IndeCateR	40.38 ± 11.61	52.67 ± 10.85	12.28 ± 2.62	66.50 ± 1.37
NASR	16.41 ± 1.79	17.32 ± 1.92	2.02 ± 0.23	**82.78 ± 1.06**
ISED (ours)	**82.32 ± 4.15**	**79.95 ± 5.71**	**68.59 ± 1.95**	80.32 ± 1.79

including their pre-trained digit recognition networks [4]. We use the SatNet dataset consisting of 9K training samples and 500 test samples [20].

9.2 Evaluation Setup

All of our experiments were conducted on a machine with two 20-core Intel Xeon CPUs, one NVIDIA RTX 2080 Ti GPU, and 755 GB RAM. Unless otherwise noted, the sample count, i.e., the number of calls to the program P per training example, is fixed at 100 for all relevant methods. We configure ISED to use the `min-max` semiring for HWF and the `add-mult` semiring for all other tasks. We use categorical sampling and binary cross-entropy loss for ISED. We apply a timeout of 10 seconds per testing sample, and report the average accuracy and 1-sigma standard deviation obtained from 10 randomized runs.

9.3 Performance and Accuracy

To answer **RQ1**, we evaluate the accuracy of the presented techniques on the benchmark suite. We report the average test accuracy as well as the standard deviation from 10 randomized runs in Tables 4-7. The neurosymbolic learning frameworks generally achieve high accuracy across the benchmarks. However, they are not as expressive as the black-box gradient estimation techniques, as shown by their inability to encode the GPT leaf and scene tasks. Additionally, both DPL and Scallop time out on Sudoku, which shows that it can be beneficial to treat the Sudoku solver as a black-box rather than differentiating through its complex rules.

A-NeSI, a neural gradient approximation algorithm, also generally does well across all tasks; it is even the top performer on 3 of the 16 tasks. However, it achieves very low accuracy on the sudoku and HWF tasks, suggesting that the neural network that approximates the WMC result is unable to approximate well for complex programs.

REINFORCE and IndeCateR achieve high accuracy for simple tasks such as not-3-or-4. However, their low accuracy on tasks such as sum_4 suggest that they are limited by sample inefficiency. IndeCateR specifically has been shown to scale well to high dimensional input spaces, but fixing its sample count to the relatively small number that we have selected in our evaluation results in a weak learning signal, and consequently, low accuracy. NASR achieves high accuracy on the sudoku fine-tuning task but struggles to learn the other tasks which involve neural networks with randomly intialized weights. ISED performs well across all of the benchmark tasks and comes out as the top performer on 8 of the 16 total tasks.

Table 5. Performance comparison for HWF, sum_2, sum_3, and sum_4.

	Accuracy (%)			
Method	HWF	sum_2	sum_3	sum_4
DPL	TO	95.14 ± 0.80	93.80 ± 0.54	TO
Scallop	96.65 ± 0.13	91.18 ± 13.43	91.86 ± 1.60	80.10 ± 20.4
A-NeSI	3.13 ± 0.72	**96.66 ± 0.87**	94.39 ± 0.77	78.10 ± 19.0
REINFORCE	18.59 ± 7.88	74.46 ± 26.29	19.40 ± 4.52	13.84 ± 2.26
IndeCateR	15.14 ± 4.95	95.70 ± 0.29	66.24 ± 15.08	13.02 ± 0.94
NASR	1.85 ± 0.27	6.08 ± 0.77	5.48 ± 0.77	4.86 ± 0.93
ISED (ours)	**97.34 ± 0.26**	80.34 ± 16.14	**95.10 ± 0.95**	**94.1 ± 1.6**

Table 6. Performance comparison for $mult_2$, mod_2, less-than, and add-mod-3.

	Accuracy (%)			
Method	$mult_2$	mod_2	less-than	add-mod-3
DPL	95.43 ± 0.97	96.34 ± 1.06	**96.60 ± 1.02**	95.28 ± 0.93
Scallop	87.26 ± 24.70	77.98 ± 37.68	80.02 ± 3.37	75.12 ± 21.64
A-NeSI	**96.25 ± 0.76**	**96.89 ± 0.84**	94.75 ± 0.98	77.44 ± 24.60
REINFORCE	96.62 ± 0.23	94.40 ± 2.81	78.92 ± 2.31	**95.42 ± 0.37**
IndeCateR	96.32 ± 0.30	93.88 ± 3.76	78.20 ± 1.54	94.02 ± 0.88
NASR	5.34 ± 0.68	20.02 ± 2.67	49.30 ± 2.14	33.38 ± 2.81
ISED (ours)	96.02 ± 1.13	96.68 ± 0.93	96.22 ± 0.95	83.76 ± 12.89

Table 7. Performance comparison for add-sub, equal, not-3-or-4, and count-3-4.

	Accuracy (%)			
Method	add-sub	equal	not-3-or-4	count-3-4
DPL	93.86 ± 0.87	**98.53 ± 0.37**	98.19 ± 0.55	TO
Scallop	92.02 ± 1.58	71.60 ± 2.29	97.42 ± 0.73	93.47 ± 0.83
A-NeSI	93.95 ± 0.60	77.89 ± 36.01	98.63 ± 0.50	93.73 ± 2.93
REINFORCE	17.86 ± 3.27	78.26 ± 3.96	**99.28 ± 0.21**	87.78 ± 1.14
IndeCateR	70.12 ± 16.69	83.10 ± 2.76	**99.28 ± 0.17**	2.26 ± 1.42
NASR	5.26 ± 1.10	81.72 ± 1.94	68.36 ± 1.54	25.26 ± 1.66
ISED (ours)	**95.32 ± 0.81**	96.02 ± 1.74	98.08 ± 0.72	**95.26 ± 1.04**

9.4 RQ2: Sample Efficiency

To answer **RQ2**, we evaluate the techniques in terms of sample efficiency, i.e., how changing the number of samples affects accuracy. We evaluate several of the REINFORCE-baesd techniques, namely REINFORCE, ISED, IndeCateR, and IndeCateR+, where IndeCateR+ [16] is a variant of IndeCateR that is customized for higher dimensional settings. Our evaluation uses the MNIST addition task sum_n for $n \in \{8, 12, 16\}$ with varried sample count. We report the average accuracy and standard deviation obtained from 5 randomized runs (Tables 8-10).

For lower sample counts, ISED outperforms the baselines on all three tasks, even outperforming IndeCateR by over 80% on sum_8 and sum_{12}. The results show that other REINFORCE-based method

Table 8. Performance comparison for sum_8 with different sample counts k.

	Accuracy (%)	
	sum_8	
Method	$k = 80$	$k = 800$
REINFORCE	8.32 ± 2.52	8.28 ± 0.39
IndeCateR	5.36 ± 0.26	**89.60 ± 0.98**
IndeCateR+	10.20 ± 1.12	88.60 ± 1.09
ISED (Ours)	**87.28 ± 0.76**	87.72 ± 0.86

Table 9. Performance comparison for sum_{12} with different sample counts k.

	Accuracy (%)	
	sum_{12}	
Method	$k = 120$	$k = 1200$
REINFORCE	7.52 ± 1.92	8.20 ± 1.80
IndeCateR	4.60 ± 0.24	77.88 ± 6.68
IndeCateR+	6.84 ± 2.06	**86.92 ± 1.36**
ISED (Ours)	**85.72 ± 2.15**	86.72 ± 0.48

Table 10. Performance comparison for sum_{16} with different sample counts k.

	Accuracy (%)	
	sum_{16}	
Method	$k = 160$	$k = 1600$
REINFORCE	5.12 ± 1.91	6.28 ± 1.04
IndeCateR	1.24 ± 1.68	5.16 ± 0.52
IndeCateR+	4.24 ± 0.95	**83.52 ± 1.75**
ISED (Ours)	**6.48 ± 0.50**	8.13 ± 1.10

often provide a weaker learning signal per sample compared to ISED. However, the results also demonstrate the main limitation of ISED, which is its inability to scale to high dimensional inputs as well as specialized techniques. While ISED achieves similar accuracy compared to the top performer (IndeCateR or IndeCateR+) on sum_8 and sum_{12} with a high sample count, it comes second to IndeCateR+ on sum_{16}, with an accuracy difference of 75.39%. This motivates exploring better sampling techniques for ISED, which is the core difference between IndeCateR and IndeCateR+, to improve its scalability.

9.5 RQ3: Data Efficiency

We now examine how ISED and A-NeSI compare in terms of data efficiency. We compare the two techniques in terms of training time and accuracy on sum_3 and sum_4. We choose these tasks for evaluation because A-NeSI has been shown to scale well to multi-digit addition tasks [19]. Furthermore, these tasks come from the MNIST-R suite in which we use 5K training samples, which is fewer than what A-NeSI would have used in its evaluation (20K training samples for sum_3 and 15K for sum_4). We plot the average test accuracy and standard deviation vs. training time (over 10 runs) in Figures 3 and 4, where each point represents the result of 1 epoch.

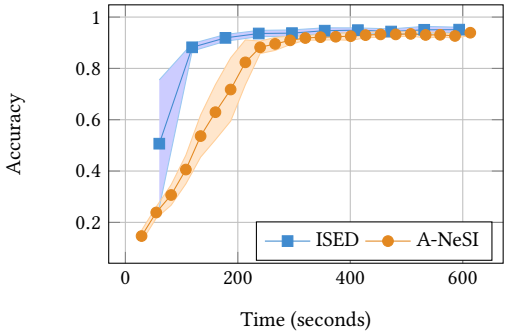

Fig. 3. Accuracy vs. Time for sum_3.

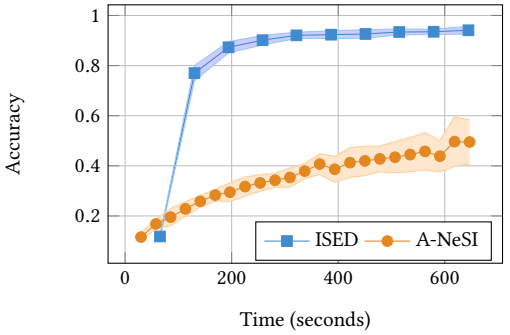

Fig. 4. Accuracy vs. Time for sum_4.

While ISED and A-NeSI learn at about the same rate for sum_3 after about 5 minutes of training, ISED learns at a much faster rate for the first 5 minutes, reaching an accuracy of 88.22% after just 2 epochs (Fig. 3). The difference between ISED and A-NeSI is more pronounced for sum_4, with ISED reaching an accuracy of 94.10% after just 10 epochs while A-NeSI reaches 49.51% accuracy at the end of its 23rd epoch (Fig. 4). These results demonstrate that with limited training data, ISED is able to learn more quickly than A-NeSI, even for simple tasks. This result is likely due to A-NeSI training 2 additional neural models in its learning pipeline compared to ISED, with A-NeSI training a prior as well as a model to estimate the WMC result and its associated gradient.

10 CONCLUSION AND FUTURE WORK

We compared neurosymbolic frameworks and black-box gradient-estimation techniques in their conceptual differences and ability to learn neural programs. We found that Scallop improves upon DPL, and IndeCateR improves upon REINFORCE, both in terms of scalability, without sacrificing accuracy. Our results demonstrate that NASR is able to fine-tune neural programs for complex tasks like Sudoku solving but does not learn even simple neural programs when the weights are randomly initialized. A-NeSI has shown promising potential in its ability to scale to tasks with a large space of inputs and outputs, but our experiments show that it is not suitable for learning complex tasks such as Sudoku solving and hand-written formula evaluation. Furthermore, our results demonstrate that ISED is limited in its inability to scale to high dimensional inputs, but it is able to learn in a more data- and sample-efficient manner compared to the other techniques.

For future work on ISED, it is worth exploring whether the scalability of the algorithm can be improved with better sampling techniques. Another promising direction would be to extend ISED so that it is compatible with other types of neural+black-box architectures. For example, one might want to train an architecture that consists of a neural network, followed by a program that calls an LLM, followed by another neural network. Such an architecture might be of interest in the case of training a neural model with a bias parameter that processes the output from the LLM. For this architecture, we would need a black-box gradient estimation algorithm that explicitly estimates a Jacobian, which could be accomplished with a finite-difference technique from numerical analysis.

REFERENCES

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

[2] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, pages 772–799, 2008.

[3] Siddharth Singh Chouhan, Uday Pratap Singh, Ajay Kaul, and Sanjeev Jain. A data repository of leaf images: Practice towards plant conservation with plant pathology. In *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*, pages 700–707, 2019. doi: 10.1109/ISCON47742.2019.9036158.

[4] Cristina Cornelio, Jan Stuehmer, Shell Xu Hu, and Timothy Hospedales. Learning where and when to reason in neuro-symbolic inference. In *International Conference on Learning Representations*, 2023.

[5] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, page 31–40, 2007.

[6] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. *CoRR*, 2012.

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

[8] Qing Li, Siyuan 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*, page 5884–5894, 2020.

[9] Ziyang Li, Jiani Huang, and Mayur Naik. Scallop: A language for neurosymbolic programming. In *ACM International Conference on Programming Language Design and Implementation*, page 1463–1487, 2023.

[10] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, page 3753–3763, 2018.

[11] Robin Manhaeve, Giuseppe Marra, and Luc De Raedt. Approximate Inference for Neural Probabilistic Logic Programming. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, pages 475–486, 2021.

[12] Lukas Murmann, Michael Gharbi, Miika Aittala, and Fredo Durand. A multi-illumination dataset of indoor object appearance. In *2019 IEEE International Conference on Computer Vision (ICCV)*, Oct 2019.

[13] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. *IJCAI*, page 2462–2467, 2007.

[14] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016. doi: 10.1109/CVPR.2016.91.

[15] Tian Sang, Paul Bearne, and Henry Kautz. Performing bayesian inference by weighted model counting. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1*, page 475–481, 2005.

[16] Lennert De Smet, Emanuele Sansone, and Pedro Zuidberg Dos Martires. Differentiable sampling of categorical distributions using the catlog-derivative trick. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, 2023.

[17] Alaia Solko-Breslin, Seewon Choi, Ziyang Li, Neelay Velingker, Rajeev Alur, Mayur Naik, and Eric Wong. Data-efficient learning with neural programs, 2024.

[18] Efthymia Tsamoura, Timothy Hospedales, and Loizos Michael. Neural-symbolic integration: A compositional perspective. In *AAAI Conference on Artificial Intelligence*, 2020.

[19] Emile van Krieken, Thiviyan Thanapalasingam, Jakub M. Tomczak, Frank van Harmelen, and Annette ten Teije. A-nesi: A scalable approximate method for probabilistic neurosymbolic inference. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, 2023.

[20] Po-Wei Wang, Priya Donti, Bryan Wilder, and Zico Kolter. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *Proceedings of the 36th International Conference on Machine Learning*, pages 6545–6554, 2019.

[21] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3–4):229–256, 1992. ISSN 0885-6125.