Data-Efficient Learning with Neural Programs

Alaia Solko-Breslin  Seewon Choi  Ziyang Li  Neelay Velingker  Rajeev Alur  Mayur Naik  Eric Wong
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
{alaia,seewon,liby99,neelay,alur,mhnaik,exwong}@seas.upenn.edu

Abstract

Many computational tasks can be naturally expressed as a composition of a DNN followed by a program written in a traditional programming language or an API call to an LLM. We call such composites "neural programs" and focus on the problem of learning the DNN parameters when the training data consist of end-to-end input-output labels for the composite. When the program is written in a differentiable logic programming language, techniques from neurosymbolic learning are applicable, but in general, the learning for neural programs requires estimating the gradients of black-box components. We present an algorithm for learning neural programs, called ISED, that only relies on input-output samples of black-box components. For evaluation, we introduce new benchmarks that involve calls to modern LLMs such as GPT-4 and also consider benchmarks from the neurosymbolic learning literature. Our evaluation shows that for the latter benchmarks, ISED has comparable performance to state-of-the-art neurosymbolic frameworks. For the former, we use adaptations of prior work on gradient approximations of black-box components as a baseline, and show that ISED achieves comparable accuracy but in a more data- and sample-efficient manner.\footnote{Code is available at https://github.com/alaiasolkobreslin/ISED/tree/v1.0.0}

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. We call such composites "neural programs" and study the problem of learning neural programs in an end-to-end manner with a focus on data and sample efficiency. One problem that is naturally expressed as a neural program is scene recognition [29], where $M_θ$ classifies objects in an image and $P$ prompts GPT-4 to identify the room type given these objects (Fig. 1).

Neurosymbolic learning [2] is one instance of neural program learning in which $P$ takes the form of a logic program. DeepProbLog (DPL) [14] and Scallop [13] are frameworks that extend ProbLog and Datalog, respectively, to ensure that the symbolic component $P$ is differentiable. This differentiability requirement is what facilitates learning in many neurosymbolic learning frameworks. There are also abductive learning frameworks that do not explicitly differentiate programs. Instead, they require that the symbolic component expose a method for abducing the function’s inputs for a given output, often using Prolog for the symbolic component as a result [6, 23]. While logic programming languages are expressive enough for these frameworks to solve tasks such as sorting [14], visual question answering [13], and path planning [23], they offer restricted features and a narrow range of libraries, making them incompatible with calls to arbitrary APIs or to modern LLMs.

Learning neural programs when $P$ is not expressed as a logic program is a difficult problem because gradients across black-box programs cannot be computed explicitly. One possible solution is to use...
REINFORCE \cite{Williams1992} to sample symbols from distributions predicted by $M_\theta$ and compute the expected reward using the output label. However, REINFORCE is not sample-efficient as it produces a weak learning signal, especially when applied to programs with a large number of inputs. There are other REINFORCE-based methods that can be applied to the neural program learning setting, namely IndeCateR \cite{Liu2021} and Neural Attention for Symbolic Reasoning (NASR) \cite{Fang2020}. However, IndeCateR struggles with sample efficiency despite providing lower variance than REINFORCE, and NASR performs poorly when intermediate labels are unavailable for pretraining. Another possible solution is Approximate Neurosymbolic Inference (A-NeSI) \cite{Yang2020}, which trains a neural model to estimate the gradient of $P$, but learning the surrogate neural network becomes more difficult as the complexity of $P$ increases. Moreover, the additional neural models in the learning framework in A-NeSI results in data inefficiency.

In this paper, we propose an algorithm for learning neural programs, based on reinforcement learning, which is compatible with arbitrary programs. Our approach, called ISED (Infer-Sample-Estimate-Descend), yields a framework that expands the applicability of neural program learning frameworks by providing a data- and sample-efficient method of training neural models with randomly initialized weights. ISED uses outputs of $M_\theta$ as a probability distribution over inputs of $P$ and samples representative symbols $u$ from this 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 $\text{if symbol} = u \text{ then output} = v$ summarizing $P$. The final step is to estimate the gradient across this symbolic summary, inspired by ideas from the neurosymbolic learning literature, to propagate loss across the composite model.

Our evaluation considers 16 neural program benchmark tasks. Our results show that ISED outperforms purely neural networks and CLIP \cite{Radford2021} on new neural program tasks involving GPT-4 calls. Additionally, ISED outperforms neurosymbolic methods on 9 of the 14 benchmarks tasks that can be encoded in logic programming languages. ISED is also the top performer on 8 out of the 16 benchmark tasks when compared to REINFORCE-based and black-box gradient estimation baselines. Furthermore, we show that ISED is more data- and sample-efficient than baseline methods.

In summary, the main contributions of this paper are as follows: 1) we introduce neural programs as a generalization of neurosymbolic programs, 2) we introduce new tasks involving neural programs that use Python and calls to GPT-4 called neuroPython and neuroGPT programs, respectively, 3) we present ISED, a general algorithm for data- and sample-efficient learning with neural programs, and 4) we conduct a thorough evaluation using existing techniques against a diverse set of benchmarks.

## 2 Neural Programs

**Problem Statement.** In the neural program learning setting, we attempt to optimize model parameters $M_\theta$ which are being supervised by a fixed program $P$. Specifically, we are given a training dataset $D$ of length $N$ containing input-output pairs, i.e., $D = \{(x_1, y_1), \ldots, (x_N, y_N)\}$. Each $x_j$ 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 $x_j$. Given a loss function $\mathcal{L}$, we want to minimize the loss of $\mathcal{L}(P(M_\theta(x)), y_i)$ for each $(x_i, y_i)$ pair in order to optimize $\theta$. Loss minimization is straightforward when there is some mechanism for automatically differentiating programs, but we focus on the setting of optimizing $\theta$ without assuming the differentiability of $P$. We now introduce three motivating applications that can be framed in this
Figure 2: Illustration of our inference pipeline for the leaf classification task. \texttt{leaf\_id} can be written with a decision tree (top program) or with a call to GPT-4 (bottom program).

setting, namely classifying images of leaves [9], scene recognition [29], and hand-written formula evaluation (HWF) [12].

**Leaf Classification.** We consider a real-world example that deals with the problem of classifying leaf images. Traditional neural methods predict the species directly, without explicit notion of leaf features such as margin, shape, and texture, resulting in solutions that are data-inefficient, inaccurate, and harder to understand.

We instead present a neural programming solution, making use of leaf classification decision trees [22]. These decision trees allow identifying plant species based on the visible characteristics of their leaves. Here, the neural model takes a leaf image and predicts its shape, margin, and texture. The program can then be written in two ways: one implementation involves encoding the decision tree in Python; another involves constructing a prompt using the predicted leaf features and calling GPT-4 (see Fig. 2). The latter is possible because ISED allows the use of black-box programs, so programs can also use state-of-the-art foundation models such as GPT-4 for computation.

**Scene Recognition.** The goal of this task is to classify images according to their room types. The model receives an image from a scene dataset [16] and predicts among the 9 different room types: bedroom, bathroom, dining room, living room, kitchen, lab, office, house lobby, and basement.

The traditional neural solution trains a convolutional neural network that directly predicts the room type. On the other hand, the neural program solution decomposes the task into detecting objects in the scene and identifying the room type based on those objects. We use an off-the-shelf object detection model YOLOv8 [20] and finetune it with a custom convolutional neural network to output labels related to scene recognition. We then make a GPT-4 call to predict the most likely room type given the list of detected objects.

**Hand-written Formula.** In this task, a model is given a list of hand-written symbols containing digits (0-9) and operators (+, −, ×, and ÷) [12]. The dataset contains length 1-7 formulas free of syntax or divide-by-zero errors. The model is trained with supervision on the evaluated floating-point result without the label of each symbol. Since inputs are combinatorial and results are rational numbers, end-to-end neural methods struggle with accuracy. Meanwhile, neurosymbolic methods for this task either use specialized algorithms [12] or handcrafted differentiable programs [13].

With ISED, the program can be written in just a few lines of Python. It takes in a list of characters representing symbols, and simply invokes the Python \texttt{eval} function on the joined expression string. Under ISED, the \texttt{hwf} evaluation function can be used just like any other PyTorch [17] module, and will internally perform sampling and probability estimation to facilitate learning.

### 3 Learning Neural Programs

In this section, we present the intuition behind ISED and the values it approximates. Next, we introduce the programming interface for ISED, which lays the groundwork for presenting the algorithm. We then formally describe the steps of ISED.
3.1 ISED Overview

Assuming $P$ is a black-box, we can collect symbol-output samples $(u, v)$ from $P$. Such collection of samples can be viewed as a summary logic program consisting of rules of the form $\text{if } r = u \text{ then } y = v$. For instance, in the task of adding two digits $r_1$ and $r_2$, one rule of the logic program would be $r_1 = 1 \land r_2 = 2 \rightarrow y = 3$. Techniques from neurosymbolic literature via exact or approximate weighted model counting (WMC) [10] can then be used for computing the gradient across such a summary of $P$. However, having the complete summary of all combinations of symbols is not feasible for a black-box $P$. ISED samples symbols from the probability distribution predicted by the neural network $M_\theta$, evaluates $P$ on each sample, and takes the gradient across this partial summary of $P$. This is a good approximation of the complete summary since it is likely to contain symbols with high probability which contribute the most in exact computation.

This approach differs from REINFORCE in how it differentiates through this summary of $P$. REINFORCE rewards sampled symbols that resulted in the correct output through optimizing the log probability of the samples, weighted by reward values. This weighted-sum style estimation provides a weaker learning signal compared to WMC used by ISED, making learning harder for REINFORCE as the number of inputs to $P$ increases. See Appendix A for further details.

3.2 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 $\tau$ as either a discrete mapping (with $\Sigma$ 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 $\tau$ inductively as follows:

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

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

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 $\tau_o$ is the output type. For example, the structural input mapping for the hand-written formula task is $\text{List}_7(\text{Discrete}(\{0, \ldots, 9, +, -, \times, \div\}))$, and the structural output mapping is $\text{Float}$. 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 1.

<table>
<thead>
<tr>
<th>Mapping ($\tau$)</th>
<th>Set Interpretation ($\text{SET}(\tau)$)</th>
<th>Tensor Interpretation ($\text{DIST}(\tau)$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Discrete}(\Sigma)$</td>
<td>$\Sigma$</td>
<td>${v \mid v \in \mathbb{R}^{\Sigma \cdot</td>
</tr>
<tr>
<td>$\text{Float}$</td>
<td>$\mathbb{R}$</td>
<td>n/a</td>
</tr>
<tr>
<td>$\text{Permutation}_n$</td>
<td>${\rho \mid \rho$ is a permutation of $[1, \ldots, n]}$</td>
<td>${v_i, v_n \mid v_i \in \mathbb{R}^n, v_n \in [0, 1], i \in 1 \ldots n}$</td>
</tr>
<tr>
<td>$\text{Tuple}(\tau_1, \ldots, \tau_m)$</td>
<td>${(a_1, \ldots, a_m) \mid a_i \in \text{SET}(\tau_i)}$</td>
<td>${(a_1, \ldots, a_m) \mid a_i \in \text{DIST}(\tau_i)}$</td>
</tr>
<tr>
<td>$\text{List}_n(\tau')$</td>
<td>${a_1, \ldots, a_j \mid j \leq n, a_i \in \text{SET}(\tau')}$</td>
<td>${a_1, \ldots, a_j \mid j \leq n, a_i \in \text{DIST}(\tau')}$</td>
</tr>
</tbody>
</table>

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 $\text{SET}(\tau)$ mappings into $\text{DIST}(\tau)$ mappings. For this purpose, we define a vectorize function $\delta_v : (\text{SET}(\tau), 2^{\tau'}) ightarrow \text{DIST}(\tau)$ for the different output mappings $\tau'$ in Table 2. When considering a datapoint $(x, y)$ during training, ISED samples many symbols and
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 \( \tau_1, \ldots, \tau_m \) be the mappings for these inputs and \( \tau_o \) the mapping for the program’s output. We write \( P \) as a function from its input mappings to its output mapping: \( P : (\tau_1, \ldots, \tau_m) \rightarrow \tau_o \). For each unstructured input \( i \) to the program, there is a neural model \( M_{\theta_i} : x_i \rightarrow \text{DIST}(\tau_i) \) for each \( \tau_i \). The sampling strategy, \( k \) the sample count, \( \mathcal{L} \) the loss function, and \( \mathcal{D} \) the dataset.

Algorithm 1 ISED training pipeline

```
Require: \( \mathcal{D} \)

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

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 \( \hat{y} \). While \( \hat{y} \) is not used by \( \delta_\tau \) in most cases, we include it as an argument so that 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 \([0.0, 0.1, 0.0]\).

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 \( \tau \) in Table 2. 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 \( \hat{p}[0][1] = 0.1, \hat{p}[1][+] = 0.05, \text{ and } \hat{p}[2][1] = 0.1 \), it would return 0.05.

### 3.3 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 \( \tau_1, \ldots, \tau_m \) be the mappings for these inputs and \( \tau_o \) the mapping for the program’s output. We write \( P \) as a function from its input mappings to its output mapping: \( P : (\tau_1, \ldots, \tau_m) \rightarrow \tau_o \). For each unstructured input \( i \) to the program, there is a neural model \( M_{\theta_i} : x_i \rightarrow \text{DIST}(\tau_i) \). \( S \) 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 present the algorithm in pseudocode in Algorithm 1 and describe its steps with the hand-written formula task.
**Infer.** The training pipeline starts with an example from the dataset, \((x, y) = (\{+, \cdot, \geq\}, 3.0)\), 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 example \( j \), 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 \( \odot \) as the \( \min \) function, and \( \oplus \) as the \( \max \) function in this example. Note that ISED requires that \( \odot \) and \( \oplus \) represent either \( \min \) and \( \max \) or \( \text{mult} \) and add respectively. We refer to these two options as the \( \min\text{-}\max \) and \( \text{add\text{-}mult} \) semirings. We define an accumulate function \( \omega \) that takes as input an element of the output mapping \( y_k \), sampled outputs \( \hat{r} \), and predicted input distributions \( \hat{p} \). The accumulator performs the \( \odot \) operation on aggregated probabilities for elements of \( y \) that are equal to \( y_k \) and is defined as follows:

\[
\omega(y_k, \hat{y}, \hat{r}, \hat{p}) = \odot_{j=1}^{k} \mathbb{1}_{\hat{y}_j = y_k} \sigma_{\tau_o}(\hat{r}_j, \hat{p}_j)
\]

ISED then sets \( \hat{w} = [\omega(y_k, \hat{y}, \hat{r}, \hat{p}) \mid y_k \in \tau_o] \) in the case where \( \tau_o \) is \( \text{not} \) \( \text{FLOAT} \). When \( \tau_o \) is \( \text{FLOAT} \), as for hand-written formula, it only considers \( y_k \in \hat{y} \). Next, it performs \( L_2 \) normalization over each element in \( \hat{w} \) 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 \( \frac{\partial l}{\partial \theta} \) 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 \( \theta_i \) for each input \( i \) based on \( \frac{\partial l}{\partial \theta} \) using a stochastic optimizer (e.g., Adam optimizer). This completes the training pipeline for one example, and the algorithm returns all final \( \theta_i \) after iterating through the entire dataset.

### 4 Evaluation

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

**RQ1:** How does ISED compare to state-of-the-art neurosymbolic, REINFORCE-based, and gradient estimation baselines in terms of accuracy?

**RQ2:** What is the sample efficiency of ISED when compared to REINFORCE-based algorithms?

**RQ3:** How data-efficient is ISED compared to neural gradient estimation methods?

#### 4.1 Benchmark Tasks: NeuroGPT, NeuroPython, and Neurosymbolic

We first introduce two new neural program learning benchmarks which both contain a program component that can make a call to GPT-4. We call such models neuroGPT programs.

**Leaf Classification.** In this task, we use a dataset, which we call LEAF-ID, containing leaf images of 11 different plant species [4], containing 330 training samples and 110 testing samples. We define custom DISCRETE types MARGIN, SHAPE, TEXTURE. With this, we define LEAF-TRAITS = TUPLE(MARGIN, SHAPE, TEXTURE) and LEAF-OUTPUT to be the DISCRETE set of 11 plant species in the dataset. 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 [16], consisting of 830 training examples and 92 testing examples. We define custom types OBJECTS and SCENES to be DISCRETE set of 45 objects and 9 room types, respectively. We freeze the parameters of YOLOv8 and only optimize the custom neural network. The neural program solution prompts 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 [13, 14] contains 11 tasks operating on inputs of images of handwritten digits from the MNIST dataset [11]. This synthetic test suite includes tasks performing arithmetic (sum2, sum3, sum4, mult2, mod2, add-mod-3, add-sub), comparison (less-than, equal), counting (count-3-or-
Table 3: Performance on selected benchmarks. "TO" means time-out, and "N/A" means the task could not be programmed in the framework. Methods are divided (from top to bottom) by neurosymbolic, black-box gradient estimation, and REINFORCE-based.

<table>
<thead>
<tr>
<th>Method</th>
<th>sum₄</th>
<th>HWF</th>
<th>DT leaf</th>
<th>GPT leaf</th>
<th>scene</th>
<th>sudoku</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPL</td>
<td>TO</td>
<td>TO</td>
<td>39.70</td>
<td>N/A</td>
<td>N/A</td>
<td>TO</td>
</tr>
<tr>
<td>Scallop</td>
<td>80.10</td>
<td>96.65</td>
<td>81.13</td>
<td>N/A</td>
<td>N/A</td>
<td>TO</td>
</tr>
<tr>
<td>A-NeSI</td>
<td>78.10</td>
<td>3.13</td>
<td>78.82</td>
<td>72.40</td>
<td>61.46</td>
<td>26.36</td>
</tr>
<tr>
<td>REINFORCE</td>
<td>13.84</td>
<td>18.50</td>
<td>23.60</td>
<td>34.02</td>
<td>47.07</td>
<td>79.08</td>
</tr>
<tr>
<td>IndeCateR</td>
<td>13.02</td>
<td>15.14</td>
<td>40.38</td>
<td>52.67</td>
<td>12.28</td>
<td>66.50</td>
</tr>
<tr>
<td>NASR</td>
<td>4.86</td>
<td>1.85</td>
<td>16.41</td>
<td>17.32</td>
<td>2.02</td>
<td>82.78</td>
</tr>
<tr>
<td>ISED (ours)</td>
<td>94.10</td>
<td>97.34</td>
<td>82.32</td>
<td>79.95</td>
<td>68.59</td>
<td>80.32</td>
</tr>
</tbody>
</table>

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 \([12]\). 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 solve an incomplete 9x9 Sudoku, where the problem board is given as MNIST digits. We follow the experimental setting used in NASR, including their pre-trained MNIST digit recognition models \([5]\). We use the SatNet dataset consisting of 9K training samples and 500 test samples \([25]\).

### 4.2 Evaluation Setup and Baselines

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. For additional details on experimental setup, see Appendix B. 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.

We pick as baselines neurosymbolic methods DeepProbLog (DPL) \([14]\) and Scallop \([13]\), A-NeSI \([24]\) which performs neural approximation of the gradients, and sampling-based gradient approximation methods REINFORCE \([26]\), IndeCateR \([21]\), and NASR \([5]\). IndeCateR achieves provably lower variance than REINFORCE by using a specialized sampling method (Appendix A), and NASR is a variant specialized for efficient finetuning by using a single sample and a custom reward function. We also use purely neural baselines and CLIP \([19]\) for GPT leaf and scene. CLIP is a multimodal model that supports zero-data image classification by simply providing names of the categories.

### 4.3 RQ1: Performance and Accuracy

To answer **RQ1**, we evaluate ISED’s accuracy against those of the baselines. ISED matches, and in many cases surpasses, the accuracy of neurosymbolic and gradient estimation baselines. We highlight the results for sum₄ from MNIST-R and other benchmarks in Table 3. Tables 7-10 in Appendix C contain results for the remaining MNIST-R tasks, including standard deviations for all tasks. ISED is the top performer on 8 out of the 16 total tasks. On the GPT leaf and scene tasks, ISED outperforms the purely neural baseline by 3.82% and 31.42% respectively, and zero-shot CLIP by 59.80% and 17.50%. For many tasks, A-NeSI is the non-neurosymbolic method that comes closest to ISED, sometimes outperforming our method. However, A-NeSI achieves significantly lower performance than ISED on tasks involving complex programs, namely HWF and sudoku. This is likely due to the difficulty of training a neural model to estimate the output of \(P\) and its gradient when \(P\) is complex. ISED also outperforms REINFORCE on all but 2 tasks due to the REINFORCE learning signal being weaker for tasks where \(P\) involves multiple inputs. NASR outperforms ISED only on sudoku by 2.46% due to NASR being well-suited for fine-tuning as it restricts its algorithm to use one sample.
Table 4: Performance comparisons for sum₈, sum₁₂, and sum₁₆ with different sample counts k.

<table>
<thead>
<tr>
<th>Method</th>
<th>sum₈ (k = 80)</th>
<th>sum₈ (k = 800)</th>
<th>sum₁₂ (k = 120)</th>
<th>sum₁₂ (k = 1200)</th>
<th>sum₁₆ (k = 160)</th>
<th>sum₁₆ (k = 1600)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REINFORCE</td>
<td>8.32</td>
<td>8.28</td>
<td>7.52</td>
<td>8.20</td>
<td>5.12</td>
<td>6.28</td>
</tr>
<tr>
<td>IndeCateR</td>
<td>5.36</td>
<td><strong>89.60</strong></td>
<td>4.60</td>
<td>77.88</td>
<td>1.24</td>
<td>5.16</td>
</tr>
<tr>
<td>IndeCateR+</td>
<td>10.20</td>
<td>88.60</td>
<td>6.84</td>
<td><strong>86.92</strong></td>
<td>4.24</td>
<td><strong>83.52</strong></td>
</tr>
<tr>
<td>ISED (Ours)</td>
<td><strong>87.28</strong></td>
<td>87.72</td>
<td><strong>85.72</strong></td>
<td>86.72</td>
<td><strong>6.48</strong></td>
<td>8.13</td>
</tr>
</tbody>
</table>

ISED also outperforms IndeCateR on all but 4 tasks due to IndeCateR being less sample efficient than ISED.

ISED outperforms the neurosymbolic methods on 9 out of 14 tasks that can be written in logic programming languages. Despite treating $P$ as a black-box, ISED even outperforms Scallop on HWF by 0.69% and comes within 1.16% of NGS, a specialized neurosymbolic learning framework that uses abductive reasoning [12]. Furthermore, DPL timed out on 4 tasks, and Scallop timed out on 1 (sudoku). These results demonstrate that even for tasks that can be written in a logic programming language, treating the program as a black-box can often yield optimal results.

### 4.4 RQ2: Sample Efficiency

To answer RQ2, we evaluate the sample efficiency of ISED against REINFORCE, IndeCateR, and IndeCateR+ on adding MNIST digits. IndeCateR+ [21] is a variant of IndeCateR with a sampling method and loss computation customized for higher dimensional setting such as the addition of 16 MNIST digits. We vary the size of the input and output space (sum₈, sum₁₂, sum₁₆) of $P$ as well as the sample count, and report the average accuracy and standard deviation obtained from 5 randomized runs (Tables 4, 11-13).

For a lower number of samples, ISED outperforms all other methods on the three tasks, outperforming IndeCateR by over 80% on sum₈ and sum₁₂. The experimental findings support the conceptual difference of REINFORCE-based methods providing a weak learning signal compared to ISED (Section 3.1). While ISED achieves accuracy similar to the top performer for sum₈ and sum₁₂ with a high sample count, it comes second on sum₁₆ with IndeCateR+ beating ISED by 75.39%. This suggests our approach is limited in scaling to high-dimensional inputs to $P$, and motivates exploring better sampling techniques, which is the core difference between IndeCateR and IndeCateR+.

### 4.5 RQ3: Data Efficiency

We now examine how ISED compares to state-of-the-art baselines in terms of data efficiency. We compare ISED and A-NeSI in terms of training time and accuracy on sum₃ and sum₄. We choose these tasks for evaluation because A-NeSI has been shown to scale well to multi-digit addition tasks [24]. Furthermore, these tasks come from the MNIST-R suite in which we use 5K training samples, which is less than what A-NeSI would have used in its evaluation (20K training samples for sum₃.
and 15K for \( \text{sum}_4 \). 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.

While ISED and A-NeSI learn at about the same rate for \( \text{sum}_3 \) 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 \( \text{sum}_4 \), 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 program output and gradient.

5 Limitations and Future Work

The main limitation of ISED is the difficulty of scaling with the dimensionality of the space of inputs to the program \( P \). There are interesting future directions in adapting and expanding ISED for high dimensionality. Specifically, improvements to the sampling strategy could help adapt ISED to a complex space of inputs. Techniques can be borrowed from the field of Bayesian optimization where such large spaces have traditionally been studied. Furthermore, there is merit to systematically combining white-box and black-box methods. ISED is especially useful when logic programs fail to encode reasoning components. Therefore, we believe that ISED can be used as an underlying engine for a new neurosymbolic language that blends the accessibility of black-box with the performance of white-box methods.

6 Related Work

**Neurosymbolic programming frameworks.** These frameworks provide a general mechanism to define white-box neurosymbolic programs. DeepProbLog [14] and Scallop [13] abstract away gradient calculations behind a rule-based language. Others specialize in targeted applications, such as NeurASP [28] for answer set programming, or NeuralLog [3] for phrase alignment in NLP. ISED is similar in that it seeks to make classes of neurosymbolic programs easier to write and access; however, it diverges by offering an interface not bound by any specific domain or language syntax.

**RL and sampling-based neurosymbolic frameworks.** ISED incorporates concepts found in the RL algorithm REINFORCE [26] such as the sampling of actions according to the current policy distribution, similar to NASR [5], and IndeCateR [21]. Other work has proposed a semantic loss function for neurosymbolic learning which measures how well neural network outputs match a given constraint [27]. While this technique resembles ISED in that it samples symbols from their predicted distributions to derive the loss, it relies on symbolic knowledge in the form of a constraint in Boolean logic, whereas ISED allows the program component to be any black-box program.

**Specialized neurosymbolic methods.** The majority of the neurosymbolic learning literature pertains to point solutions for specific use cases [7, 25]. In the HWF example, NGS [12] and several of its variants leverage a hand-defined syntax defining the inherent structure within mathematical expressions. Similarly, DiffSort [18] leverages the symbolic properties of sorting to produce differentiable sorting networks. Other point solutions address broader problem setups, such as NS-CL [15] which provides a framework for visual question answering by learning symbolic representations in text and images. For reading comprehension, the NeRd [3] framework converts NL questions into executable programs over symbolic information extracted from text. ISED aligns with all of these point solutions by aiming to solve problems that have thus far required technically specific solutions in order to access the advantages of neurosymbolic learning, but it takes an opposite and easier approach by forgoing significant specializations and instead leverages existing solutions as black-boxes.

**Differentiable programming and non-differentiable optimization.** Longstanding libraries in deep learning have grown to great popularity for their ability to abstract away automatic differentiation behind easy-to-use interfaces. PyTorch [17] is able to do so by keeping track of a dynamic computational graph. Similarly, JAX [1] leverages functional programming to abstract automatic differentiation. ISED follows the style of these frameworks by offering an interface to abstract away gradient calculations for algorithms used in deep learning, but ISED improves upon them by allowing systematic compatibility of non-differentiable functions.
7 Conclusion

We proposed ISED, a data- and sample-efficient algorithm for learning neural programs. Unlike existing general neurosymbolic frameworks which require differentiable logic programs, ISED is compatible with Python programs and API calls to GPT, and it employs a sampling-based technique to learn neural model parameters using forward evaluation. We showed that for neuroGPT, neuroPython, and neurosymbolic benchmarks, ISED achieves better accuracy than end-to-end neural models and similar accuracy compared to neurosymbolic frameworks. ISED also often achieves superior accuracy on complex programs compared to REINFORCE-based and gradient estimation baselines. Furthermore, ISED learns in a more data- and sample-efficient manner compared to these baselines.

References


A Explanation of Differences Between ISED and Baseline Methods

We explain the differences between ISED and prior techniques using the simple example of sum_2, where digits are restricted to be between 0-2. Suppose that we are training a neural model \( M_{\theta} \) for this task, and we are considering the symbol-output sample where the ground truth symbols 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 \( M_{\theta} \) for \( r_1 \) and \( r_2 \) are \([0.1, 0.6, 0.3]\) and \([0.2, 0.1, 0.7]\) respectively. We now explain how different methods perform their loss computations.

A.1 ISED

Suppose ISED is initialized with a sample count of 3, and the sampled symbol-output pairs are ((1, 2), 3), ((1, 0), 1), and ((2, 1), 3). We use the add-mult semiring in this example. ISED can be thought of as differentiating through the following summary logic program:

\[
\begin{align*}
    r_1 = 1 \land r_2 = 2 & \rightarrow y = 3 \\
    r_1 = 1 \land r_2 = 0 & \rightarrow y = 1 \\
    r_1 = 2 \land r_2 = 1 & \rightarrow y = 3
\end{align*}
\]

As a result, the final vector calculated for the loss function before normalization would be

\[
\begin{bmatrix}
    0.0 \\
    0.6 \times 0.2 \\
    0.0 \\
    0.6 \times 0.7 + 0.3 \times 0.1 \\
    0.0
\end{bmatrix}
\]

where each value corresponds to the probability of the given output (possible outputs are in the range 0-4). Note that if there are duplicate samples, ISED includes the duplicate probabilities in its aggregation. In our implementation, we would perform normalization on this vector and then pass it into the binary cross-entropy loss function, with the ground truth vector being:

\[
\begin{bmatrix}
    0.0 \\
    0.0 \\
    0.0 \\
    1.0 \\
    0.0
\end{bmatrix}
\]

We would then minimize this loss and update \( M_{\theta} \) accordingly.

If we use the min-max semiring instead, \( \ast \) is replaced by \( \min \) and \( + \) by \( \max \) in the final vector calculation, resulting in

\[
\begin{bmatrix}
    0.0 \\
    0.2 \\
    0.0 \\
    0.6 \\
    0.0
\end{bmatrix}
\]

A.2 REINFORCE

Suppose REINFORCE is also initialized with a sample count of 3, and it samples the same symbol-output pairs as before. The final reward is computed by element-wise multiplying the log probability of each sample with its reward value and taking the mean, as follows:

\[
\begin{align*}
    \log(0.6) + \log(0.7) & = 1.0 \\
    \log(0.6) + \log(0.2) & = 0.0 \\
    \log(0.3) + \log(0.1) & = 1.0
\end{align*}
\]

and the goal is to optimize \( M_{\theta} \) to maximize this reward. While this approach resembles ISED’s loss computation for the add-mult semiring, it does not involve the mult step. As it rewards possible values instead of possible combinations, the final reward would have been the same when (1,1) and (2,2) were the correct samples, instead of (1,2) and (2,1). Hence, the learning signal is weaker compared to ISED when there is more than one input to \( P\).
A.3 IndeCateR and NASR

IndeCateR is an extension of the REINFORCE estimator that is unbiased with a provably lower variance. It assumes and exploits the factoring of the underlying multivariate distribution into independent categorical variables by summing out one dimension while keeping a sample for other dimensions fixed. For each sample drawn, IndeCateR systematically creates additional samples that differ on a single entry by enumerating all possible values for each variable. NASR targets efficient finetuning by setting the sample count to one and customizing the reward function.

The loss computation for IndeCateR and NASR are identical to that of REINFORCE, also providing weak signals. Furthermore, both set the reward to 0 for samples leading to incorrect predictions, effectively ignoring them, unlike ISED which penalizes such symbols. Since only the correct symbol contribute to the final reward, the signal is sparser than ISED, making it sample-inefficient.

A.4 A-NeSI

A-NeSI trains two additional neural networks: a prediction model $Q_{\theta}$ as a surrogate for $P$, and a prior model $R_{\alpha}$ for learning the parameters $\alpha$ for the Dirichelet distribution $D_{\alpha}$.

Suppose A-NeSI is also initialized with a sample count of 3. At each training step, A-NeSI first updates $\alpha$ using $\hat{y} = M_{\theta}(x)$. Next, it samples a single symbol from each of the 3 distributions sampled from $D_{\alpha}$, and uses the sampled symbol-output pair and the standard cross entropy loss to update $Q_{\theta}$. Then, A-NeSI optimizes $M_{\theta}$ by minimizing the loss $L(Q_{\theta}(M_{\theta}(r_1, r_2)))$ using the prediction model instead of $P$.

A.5 DeepProbLog

DeepProbLog (DPL) enumerates all possible proofs for each output and aggregates probabilities accordingly. For example, the proofs for $y = 1$ include $r_1 = 0$, $r_2 = 1$ and $r_1 = 1$, $r_2 = 0$. Thus, the probability of this output is $0.6 * 0.2 + 0.1 * 0.1$. The final vector calculated would be

\[
\begin{bmatrix}
0.1 * 0.2 \\
0.6 * 0.2 + 0.1 * 0.1 \\
0.1 * 0.7 + 0.6 * 0.1 + 0.3 * 0.2 \\
0.6 * 0.7 + 0.3 * 0.1 \\
0.3 * 0.7
\end{bmatrix}
\]

and we would pass this vector into some loss function (e.g., cross-entropy), with the same ground truth vector that ISED would use. DPL would then minimize this loss and update $M_{\theta}$ accordingly.

A.6 Scallop

Suppose Scallop is initialized to use the diff-top-1-proofs semiring. This means that for each possible output, Scallop will use the proof of that output with the highest probability. For instance, the most likely proof for $y = 1$ is $r_1 = 1$ and $r_2 = 0$, and the probability of the output $y = 1$ is $0.6 * 0.2$. The final vector calculated would be

\[
\begin{bmatrix}
0.1 * 0.2 \\
0.6 * 0.2 \\
0.1 * 0.7 \\
0.6 * 0.7 \\
0.3 * 0.7
\end{bmatrix}
\]

and we would pass this vector into some loss function (e.g., binary-cross-entropy), with the same ground truth vector that ISED would use. Scallop would then minimize this loss and update $M_{\theta}$ accordingly. This probability estimation would change depending on the choice of semiring (e.g., diff-top-k-proofs for a different value of $k$).

B Evaluation Setup

For tasks with the MNIST dataset as unstructured data, we employ LeNet [11], a 2-layer CNN-based model, except for sum8, sum12, and sum16 tasks where we choose a smaller 2-layer CNN used by
IndeCateR [21]. For HWF, we also use a 2-layer CNN-based model. For leaf classification tasks, images are scaled down and passed to a simple CNN-based network with 4 convolutional layers. For the scene recognition, we use YOLOv8 and a 3-layer convolutional network for neural program methods, 7-layer CNN for the purely neural solution, and CLIP with ViT-B/32.

We use the Adam optimizer with the best learning rate among \{1e\(-3\), 5e\(-4\), 1e\(-4\)\}. We train for maximum 100 epochs, but stop early if the training saturates. For MNIST-R tasks, we used learning rate 1e\(-4\) and trained ISED for 10 epochs, REINFORCE and IndeCateR for 50 epochs, and A-NeSI and NASR for 100 epochs. We trained ISED for 30 epochs, A-NeSI for 100 epochs, and the rest for 50 epochs for HWF and Leaf Classification with learning rate 1e\(-4\). For Scene Recognition, we trained A-NeSI and the purely neural baseline 50 epochs and the rest 100 epochs with learning rate 5e\(-4\). For tasks sum_8 to sum_16 we trained ISED for 50 epochs and the rest for 100 epochs with learning rate 1e\(-3\). For Visual Sudoku, we follow the setting in NASR [5] and train for 10 epochs with learning rate 1e\(-5\).

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.

### B.1 Neural-GPT Experiment Prompts

For leaf classification and scene recognition, the neural-GPT experiments, we used the up-to-date version of GPT-4, gpt-4-1106-preview and gpt-4-1106-preview respectively, with the parameter top-p set to 1e\(-8\). We present the prompts used for the experiments in Tables 5 and 6.

**Table 5: GPT-4 prompt for the leaf classification task.**

<table>
<thead>
<tr>
<th>System message</th>
<th>User message</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>You are an expert in classifying plant species based on the margin, shape, and texture of the leaves. You are designed to output a single JSON.</strong></td>
<td><strong>&lt;PLANT NAME&gt;. Classify into one of: &lt;MARGIN/SHAPE/TEXTURE&gt;.</strong></td>
</tr>
<tr>
<td><strong>Give your answer without explanation.</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Table 6: GPT-4 prompt for the scene recognition task.**

<table>
<thead>
<tr>
<th>System message</th>
<th>User message</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>You are an expert at identifying room types based on the object detected.</strong></td>
<td><strong>There are &lt;DETECTED OBJECTS&gt;.</strong></td>
</tr>
<tr>
<td><strong>Give short single responses.</strong></td>
<td><strong>What type of room is most likely? Choose among &lt;SCENES&gt;.</strong></td>
</tr>
</tbody>
</table>

We use MARGIN = \{entire, dentate, lobed, serrate, serrulate, undulate\}, SHAPE = \{elliptical, lanceolate, obovate, oblong, ovate\}, TEXTURE = \{glossy, leathery, smooth, medium\}, and PLANT NAME \in \{Alstonia Scholaris, Citrus limon, Jatropha curcas, Mangifera indica, Ocimum basilicum, Platanus orientalis, Pongamia Pinnata, Psidium guajava, Punica granatum, Syzygium cumini, Terminalia Arjuna\}.

Furthermore, SCENES = \{bathroom, bedroom, dining room, living room, kitchen, lab, office, home lobby, basement\} and DETECTED OBJECTS is a list of maximum length 10 with duplicates.

### C Full Performance Summary

We report the accuracy of all benchmarks with 1-sigma standard deviation in Tables 7, 8, 9, and 10. We further provide the performance comparison with varying sample counts with 1-sigma standard deviation in Tables 11, 12, and 13.

### D License Information

For implementing baselines, we adapt code from the official repositories of DeepProblog [14] (Apache 2.0), Scallop [8] (MIT), A-NeSI [24] (MIT), NASR [5] (MIT), and IndeCateR [21] (Apache 2.0). Additionally, our benchmarks use Multi-illumination dataset [16] (CC BY 4.0), HWF dataset (CC BY-NC-SA 3.0) from NGS [12], a subset of the leaf database [4] (CC BY 4.0), YOLOv8 (AGPL-3.0) and CLIP (MIT).
### Table 7: Performance comparison for DT leaf, GPT leaf, scene, and sudoku.

<table>
<thead>
<tr>
<th>Method</th>
<th>DT leaf</th>
<th>GPT leaf</th>
<th>scene</th>
<th>sudoku</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPL</td>
<td>39.70 ± 6.55</td>
<td>N/A</td>
<td>61.46 ± 14.18</td>
<td>26.36 ± 12.68</td>
</tr>
<tr>
<td>Scallop</td>
<td>81.13 ± 3.50</td>
<td>N/A</td>
<td>N/A</td>
<td>TO</td>
</tr>
<tr>
<td>A-NeSI</td>
<td>78.82 ± 4.42</td>
<td>72.40 ± 12.24</td>
<td>61.46 ± 14.18</td>
<td>26.36 ± 12.68</td>
</tr>
<tr>
<td>REINFORCE</td>
<td>23.60 ± 8.27</td>
<td>34.02 ± 12.71</td>
<td>47.07 ± 14.78</td>
<td>79.08 ± 0.87</td>
</tr>
<tr>
<td>IndeCatR</td>
<td>40.38 ± 11.61</td>
<td>52.67 ± 10.85</td>
<td>12.28 ± 2.62</td>
<td>66.50 ± 1.37</td>
</tr>
<tr>
<td>NASR</td>
<td>16.41 ± 1.79</td>
<td>17.32 ± 1.92</td>
<td>2.02 ± 0.23</td>
<td>82.78 ± 1.06</td>
</tr>
<tr>
<td>ISED (ours)</td>
<td>82.32 ± 4.15</td>
<td>79.95 ± 5.71</td>
<td>68.59 ± 1.95</td>
<td>80.92 ± 1.79</td>
</tr>
</tbody>
</table>

### Table 8: Performance comparison for HWF, sum₂, sum₃, and sum₄.

<table>
<thead>
<tr>
<th>Method</th>
<th>HWF</th>
<th>sum₂</th>
<th>sum₃</th>
<th>sum₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPL</td>
<td>TO</td>
<td>95.14 ± 0.80</td>
<td>93.80 ± 0.54</td>
<td>TO</td>
</tr>
<tr>
<td>Scallop</td>
<td>96.65 ± 0.13</td>
<td>91.18 ± 13.43</td>
<td>91.86 ± 1.60</td>
<td>80.10 ± 20.4</td>
</tr>
<tr>
<td>A-NeSI</td>
<td>3.13 ± 0.72</td>
<td><strong>96.66 ± 0.87</strong></td>
<td>94.39 ± 0.77</td>
<td>78.10 ± 19.0</td>
</tr>
<tr>
<td>REINFORCE</td>
<td>18.59 ± 7.88</td>
<td>74.46 ± 26.29</td>
<td>19.40 ± 4.52</td>
<td>13.84 ± 2.26</td>
</tr>
<tr>
<td>IndeCatR</td>
<td>15.14 ± 4.95</td>
<td>95.70 ± 0.29</td>
<td>66.24 ± 15.08</td>
<td>13.02 ± 0.94</td>
</tr>
<tr>
<td>NASR</td>
<td>1.85 ± 0.27</td>
<td>6.08 ± 0.77</td>
<td>5.48 ± 0.77</td>
<td>4.86 ± 0.93</td>
</tr>
<tr>
<td>ISED (ours)</td>
<td><strong>97.34 ± 0.26</strong></td>
<td>80.34 ± 16.14</td>
<td><strong>95.10 ± 0.95</strong></td>
<td><strong>94.1 ± 1.6</strong></td>
</tr>
</tbody>
</table>

### Table 9: Performance comparison for mult₂, mod₂, less-than, and add-mod-3.

<table>
<thead>
<tr>
<th>Method</th>
<th>mult₂</th>
<th>mod₂</th>
<th>less-than</th>
<th>add-mod-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPL</td>
<td>95.43 ± 0.97</td>
<td>96.34 ± 1.06</td>
<td><strong>96.60 ± 1.02</strong></td>
<td>95.28 ± 0.93</td>
</tr>
<tr>
<td>Scallop</td>
<td>87.26 ± 24.70</td>
<td>77.98 ± 37.68</td>
<td>80.02 ± 3.37</td>
<td>75.12 ± 21.64</td>
</tr>
<tr>
<td>A-NeSI</td>
<td><strong>96.25 ± 0.76</strong></td>
<td><strong>96.89 ± 0.84</strong></td>
<td>94.75 ± 0.98</td>
<td>77.44 ± 24.60</td>
</tr>
<tr>
<td>REINFORCE</td>
<td>96.62 ± 0.23</td>
<td>94.40 ± 2.81</td>
<td>78.92 ± 2.31</td>
<td><strong>95.42 ± 0.37</strong></td>
</tr>
<tr>
<td>IndeCatR</td>
<td>96.32 ± 0.30</td>
<td>93.88 ± 3.76</td>
<td>78.20 ± 1.54</td>
<td>94.02 ± 0.88</td>
</tr>
<tr>
<td>NASR</td>
<td>5.34 ± 0.68</td>
<td>20.02 ± 2.67</td>
<td>49.30 ± 2.14</td>
<td>33.38 ± 2.81</td>
</tr>
<tr>
<td>ISED (ours)</td>
<td>96.02 ± 1.13</td>
<td>96.68 ± 0.93</td>
<td>96.22 ± 0.95</td>
<td>83.76 ± 12.89</td>
</tr>
</tbody>
</table>

### Table 10: Performance comparison for add-sub, equal, not-3-or-4, and count-3-4.

<table>
<thead>
<tr>
<th>Method</th>
<th>add-sub</th>
<th>equal</th>
<th>not-3-or-4</th>
<th>count-3-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPL</td>
<td>93.86 ± 0.87</td>
<td><strong>98.53 ± 0.37</strong></td>
<td>98.19 ± 0.55</td>
<td>TO</td>
</tr>
<tr>
<td>Scallop</td>
<td>92.02 ± 1.58</td>
<td>71.60 ± 2.29</td>
<td>97.42 ± 0.73</td>
<td>93.47 ± 0.83</td>
</tr>
<tr>
<td>A-NeSI</td>
<td>93.95 ± 0.60</td>
<td>77.89 ± 36.01</td>
<td>98.63 ± 0.50</td>
<td>93.73 ± 2.93</td>
</tr>
<tr>
<td>REINFORCE</td>
<td>17.86 ± 3.27</td>
<td>78.26 ± 3.96</td>
<td><strong>99.28 ± 0.21</strong></td>
<td>87.78 ± 1.14</td>
</tr>
<tr>
<td>IndeCatR</td>
<td>70.12 ± 16.69</td>
<td>83.10 ± 2.76</td>
<td><strong>99.28 ± 0.17</strong></td>
<td>2.26 ± 1.42</td>
</tr>
<tr>
<td>NASR</td>
<td>5.26 ± 1.10</td>
<td>81.72 ± 1.94</td>
<td>68.36 ± 1.54</td>
<td>25.26 ± 1.66</td>
</tr>
<tr>
<td>ISED (ours)</td>
<td><strong>95.32 ± 0.81</strong></td>
<td>96.02 ± 1.74</td>
<td>98.08 ± 0.72</td>
<td><strong>95.26 ± 1.04</strong></td>
</tr>
</tbody>
</table>
Table 11: Performance comparison for sum$_8$ with different sample counts $k$.

<table>
<thead>
<tr>
<th>Method</th>
<th>sum$_8$ $k = 80$</th>
<th>sum$_8$ $k = 800$</th>
</tr>
</thead>
<tbody>
<tr>
<td>REINFORCE</td>
<td>$8.32 \pm 2.52$</td>
<td>$8.28 \pm 0.39$</td>
</tr>
<tr>
<td>IndeCateR</td>
<td>$5.36 \pm 0.26$</td>
<td>$89.60 \pm 0.98$</td>
</tr>
<tr>
<td>IndeCateR+</td>
<td>$10.20 \pm 1.12$</td>
<td>$88.60 \pm 1.09$</td>
</tr>
<tr>
<td>ISED (Ours)</td>
<td>$87.28 \pm 0.76$</td>
<td>$87.72 \pm 0.86$</td>
</tr>
</tbody>
</table>

Table 12: Performance comparison for sum$_{12}$ with different sample counts $k$.

<table>
<thead>
<tr>
<th>Method</th>
<th>sum$_{12}$ $k = 120$</th>
<th>sum$_{12}$ $k = 1200$</th>
</tr>
</thead>
<tbody>
<tr>
<td>REINFORCE</td>
<td>$7.52 \pm 1.92$</td>
<td>$8.20 \pm 1.80$</td>
</tr>
<tr>
<td>IndeCateR</td>
<td>$4.60 \pm 0.24$</td>
<td>$77.88 \pm 6.68$</td>
</tr>
<tr>
<td>IndeCateR+</td>
<td>$6.84 \pm 2.06$</td>
<td>$86.92 \pm 1.36$</td>
</tr>
<tr>
<td>ISED (Ours)</td>
<td>$85.72 \pm 2.15$</td>
<td>$86.72 \pm 0.48$</td>
</tr>
</tbody>
</table>

Table 13: Performance comparison for sum$_{16}$ with different sample counts $k$.

<table>
<thead>
<tr>
<th>Method</th>
<th>sum$_{16}$ $k = 160$</th>
<th>sum$_{16}$ $k = 1600$</th>
</tr>
</thead>
<tbody>
<tr>
<td>REINFORCE</td>
<td>$5.12 \pm 1.91$</td>
<td>$6.28 \pm 1.04$</td>
</tr>
<tr>
<td>IndeCateR</td>
<td>$1.24 \pm 1.68$</td>
<td>$5.16 \pm 0.52$</td>
</tr>
<tr>
<td>IndeCateR+</td>
<td>$4.24 \pm 0.95$</td>
<td>$83.52 \pm 1.75$</td>
</tr>
<tr>
<td>ISED (Ours)</td>
<td>$6.48 \pm 0.50$</td>
<td>$8.13 \pm 1.10$</td>
</tr>
</tbody>
</table>