From Perception to Programs: Regularize, Overparameterize, and Amortize

Hao Tang
Cornell University
Ithaca, NY, USA
haotang@cs.cornell.edu

Kevin Ellis
Cornell University
Ithaca, NY, USA
kellis@cornell.edu

Abstract

Toward combining inductive reasoning with perception abilities, we develop techniques for neurosymbolic program synthesis where perceptual input is first parsed by neural nets into a low-dimensional interpretable representation, which is then processed by a synthesized program. We explore several techniques for relaxing the problem and jointly learning all modules end-to-end with gradient descent: multitask learning; amortized inference; overparameterization; and a differentiable strategy for penalizing lengthy programs. Collectedly this toolbox improves the stability of gradient-guided program search, and suggests ways of learning both how to perceive input as discrete abstractions, and how to symbolically process those abstractions as programs.

CCS Concepts: • **Computing Methodologies → Machine learning**.

Keywords: Program synthesis, neurosymbolic, learning, reasoning

ACM Reference Format:
Hao Tang and Kevin Ellis. 2022. From Perception to Programs: Regularize, Overparameterize, and Amortize. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS '22), June 13, 2022, San Diego, CA, USA*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3520312.3534865

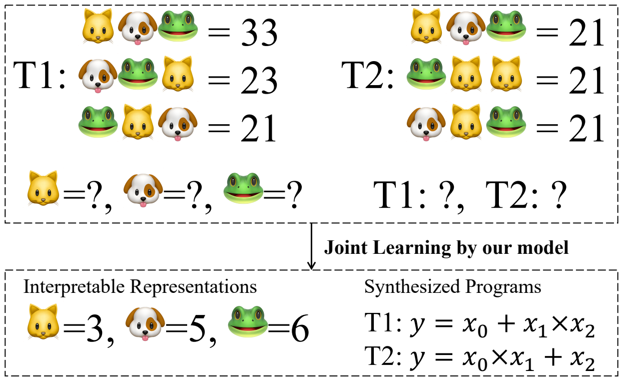
🐱🐶🐸 = 33
T1: 🐶🐸🐱 = 23
🐸🐱🐶 = 21
🐱🐶🐸 = 21
T2: 🐸🐱🐱 = 21
🐶🐱🐸 = 21
🐱=?, 🐶=?, 🐸=?
T1: ?, T2: ?
Joint Learning by our model
Interpretable Representations
🐱=3, 🐶=5, 🐸=6
Synthesized Programs
T1: $y = x_0 + x_1 \times x_2$
T2: $y = x_0 \times x_1 + x_2$

Figure 1. Solving inductive reasoning tasks grounded in perception. We consider tasks as inferring a simple formula with noisy perceptual inputs such as images. Solving this problem involves jointly learning to perceive and parse input into symbolic form (e.g., 3 numbers), and reason inductively to recover programs that explain each arithmetic task.

1 Introduction

We seek steps toward AI systems that learn to symbolically process perceptual input. Consider, for example, a system which learns to infer the 3D structure of objects: starting from pixels, it must infer low-level symbols (curves, parts), and then organize them according to symbolic relationships (symmetry, part repetitions, part hierarchy). Or, consider a system which learns to control a moving object that navigates around obstacles: starting from sensory data (lidar, RGBD), it must first parse the world (into objects, proximities, freespace), and then compute trajectories using high-level computations (PID controllers, etc.). Similar perceptual-symbolic problems arise when e.g. learning structured world models from pixels, or inferring instructions from natural language. We frame such tasks as **neurosymbolic program synthesis**: learning neural components that extract symbols from perception, and synthesizing programs to further process those symbols with more complex computations.

In this work, we study neurosymbolic program synthesis on a small-scale domain involving both perception and programmatic reasoning. Our ultimate goal is to develop general methods that could, we hope, apply to challenging neurosymbolic tasks like those previously mentioned. We take the stance that symbols should be grounded in perception, and that symbol processing should be implemented by learnable program-like representations. However, we also propose that rather than hand-code a preordained set of primitive symbols, AI systems should autonomously learn to carve the perceptual world into their own discretization. What constitutes a 'symbol' may vary across domain and

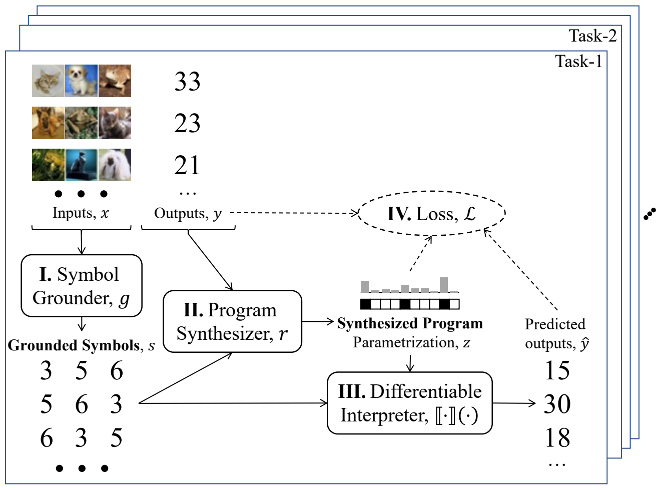
Task-2
Task-1
33
23
21
Inputs, x
Outputs, y
IV. Loss, $\mathcal{L}$
I. Symbol Grounder, g
II. Program Synthesizer, r
Synthesized Program Parametrization, z
Predicted outputs, $\hat{y}$
Grounded Symbols, s
3 5 6
5 6 3
6 3 5
III. Differentiable Interpreter, $[\![\cdot]\!](\cdot)$
15
30
18

Figure 2. The framework of our model (ROAP) for the neurosymbolic program synthesis problem. Given I/O pairs $\{(x, y)\}$ for each task, **I.** our model first uses the symbol grounder g to learn symbolic representations s for high-dimensional perceptual input x. **II.** The program synthesizer r then uses the symbol-output pairs as a program specification to infer the program for the task, $q(z|r(\{s, y\}))$. **III.** The differentiable interpreter finally executes the program on grounded symbols to predict the outputs $\hat{y} = [\![z]\!](s)$. **IV.** The whole model is trained end-to-end using a reconstruction loss of outputs and a program length regularizer.

across datasets, and can be hard for human engineers to anticipate. By jointly learning the symbols, as well as synthesizing the programs that operate on them, we hope to side-step the pitfalls associated with hand-engineered representations.

Concretely, we consider regression problems mapping perceptual input x to output y. The input x is first processed by a 'symbol grounder' neural net $g(\cdot)$, which produces intermediate symbolic output $s = g(x)$. The symbols in s are then transformed by a synthesized program z in order to predict the output $\hat{y} = [\![z]\!](s)$. We learn $g(\cdot)$ and z by encouraging the predicted $\hat{y}$ to be close to the ground-truth output y.

This setup serves as a basic model of the kinds of neurosymbolic program synthesis we seek to target. Although simple to write down, the setup is difficult to train: the grounder $g(\cdot)$ is a neural network (with unknown weights), while the program z is a discrete program (with unknown structure). To the best of our knowledge, there do not exist stable and scalable training procedures for such mixed discrete/continuous search problems, especially when $g(\cdot)$ has many parameters, and when z comes from a combinatorially large space of possible programs. At its core, we have a chicken-and-egg problem: if we had the program, then we could train $g(\cdot)$ with gradient descent to predict an element of $[\![z]\!]^{-1}(y)$; if we had the neural net, we could synthesize the program z using a variety of prior search methods [25, 29] on the input-output example $g(x) \mapsto y$.

Broadly speaking, prior works have taken three approaches for addressing this mixed discrete/continuous problem. First, the most popular strategy is to relax the discrete space of programs z and train the entire system end-to-end through backpropagation [9, 14]. Second, the complement of that approach has also been tried: discretizing the weight-space of the g network and solving for both the program and the network weights using SAT [8]. Last, alternating optimization frameworks which successively update continuous weights and discrete structure, in the style of Expectation-Maximization, have been tried [19].

We conjecture that relaxation and end-to-end backprop is likely to yield the best results, at least in the near future, because it allows one to build on the success and infrastructure of deep learning. To that end, we combine three techniques to ameliorate the optimization difficulties associated with mixed discrete/continuous problems, at least for this setting:

1. Multitasking with amortized inference: We consider a multitask setting with hundreds of interrelated programs sharing the same symbol grounding. Because they share the same symbol grounding, learning g is easier, as the many tasks 'triangulate' a good symbolic basis. Because the tasks share structure, we can make discrete program synthesis for z tractable by *amortizing* [13] program search by training an auxiliary recognition (inference) model. This 'recognition model' learns to predict programs conditioned on input-output specifications.

2. Overparameterization: We 'overparameterize' the space of programs: for example, even if we only expect a 4-line program, we search over the space of 20-line programs. Although this would be problematic for combinatorial search, gradient descent is well-known to benefit from overparameterized optimization landscapes [10], and this benefit carries over to continuous relaxations of program spaces, as first noted by terpret [12].

3. Regularization: We encourage shorter program solutions by designing program-length regularizer compatible with continuous program relaxations. We find this aids convergence. It also combats the code explosion that overparameterization naively leads to, giving shorter and more interpretable programs.

Our contribution is the combination of these techniques–**R**egularize, **O**verparameterize, **A**mortize, for **P**rograms–and so we call our approach ROAP (pronounced rope).

2 Related Work

Neurosymbolic programming is a growing area that seeks to engineer learning and inference methods for hybrid program/neural architectures [3], and our work is a special case of this broad framework. Specifically, we tackle inductive program synthesis [15]–synthesizing programs from input-output examples–but where the inputs are high-dimensional and must be preprocessed by a neural network into symbolic form. Prior works in this setting assume a hand-engineered inventory of basic symbols [6], while others backpropagate through differentiable programs to jointly train network weights and program structure [11]. Multitasking is a known strategy for this setting [31]. Training a neural network to help guide search for discrete programs (amortized inference) is standard [4, 7], and we extend that idea to continuous relaxations of program spaces.

The difficulties of gradient descent over relaxed program spaces is well known, to the extent that it has been dubbed the so-called 'terpret problem' [12]. Unfortunately, such an approach is the most straightforward way of training neural net→program architectures. From a technical perspective, our work hopes to make progress on the 'terpret problem', thereby unlocking scalable and reliable training of this class of neurosymbolic programs.

More fundamentally, our efforts connect to the body of work on the 'symbol grounding problem' [17]: How does a system learn to 'ground' abstract symbols (e.g., numbers) in terms of their high-dimensional perceptual counterparts (e.g., images of digits)? This problem is especially difficult absent strong supervision on the meaning of each abstract symbol [2], and here we consider a distantly supervised setting. Prior works consider a variety of techniques to address symbol grounding [28]. We hope that the complimentary methods we introduce can help chip away at a small part of the symbol grounding problem.

3 Methods

3.1 Relaxing Spaces of Programs

We consider straight-line code: sequences of variable assignments, each of which introduces a new variable and computes its value by applying an operator to variables that were previously computed on earlier lines. We write z to mean the syntactic structure of a straight-line program, which is represented by a triple (z^O, z^L, z^R) of binary 2D arrays encoding the syntactic structure of the program. Given this discrete *sketch* [26] of possible programs, we define a denotation operator $[\![z]\!]$ (Figure 3). Given input-output examples $x \mapsto y$, single-task neurosymbolic synthesis corresponds to

$$g(\cdot), z = \underset{\substack{g(\cdot), z \\ z\text{'s entries } \in \{0,1\}}}{\arg\min} \quad \left(y - [\![z]\!](g(x))\right)^2 \tag{1}$$

The constraint that the entries of z lie in $\{0, 1\}$ precludes end-to-end backpropagation. Because our denotation operator is actually a smooth function of z, we can relax the above optimization problem by saying the entries of z lie in $[0, 1]$:

$$g(\cdot), z = \underset{\substack{g(\cdot), z \\ z\text{'s entries } \in [0,1]}}{\arg\min} \quad \left(y - [\![z]\!](g(x))\right)^2 \tag{2}$$

Unfortunately the above objective is problematic: generically there is no reason for gradient descent to converge to a discrete z. While one can incorporate sparsity-favoring regularizers on z, doing so causes the probability of convergence to fall off rapidly as the target program length increases [9, 12], suggesting that z only converges to a discrete program if it is randomly initialized close to a correct solution.

Overparameterization. With this setup so far, we can compactly describe the first of our three techniques: Overparameterization. We simply expand the number of possible lines of code in the straight-line sketch, as illustrated in Figure 4, and expect that it will help convergence of the relaxed problem by providing a benign optimization landscape [5, 16].

3.2 Probabilistic Framing

To alleviate the problems of optimizing objective in Eq 2, we lift our problem statement to an amortized multitask setting and now give a conditional variational autoencoder [22] framing. We treat the program z as a latent variable which, given the input x, generates the output y through $p(y|x, z)$. As is standard, we introduce an auxiliary variational distribution $q\left(z|y, x\right)$ to bound the marginal likelihood $p(y|x)$. Because $q\left(z|y, x\right)$ predicts programs conditioned on input-outputs, we can think of training q as amortizing the cost of predicting programs given specifications. Mathematically we seek to maximize $p(y|x)$:

$$\begin{aligned} p(y|x) &= \sum_z p(y|x, z)p(z) \\ &\geq \mathbb{E}_q\left[\log \frac{p(y|x, z)p(z)}{q(z|y, x)}\right], \text{ELBo} \\ &= \mathbb{E}_q\left[\log p(y|x, z)\right] + \mathbb{E}_q\left[\log p(z)\right] + \mathbb{H}\left[q\right] \\ &\geq \underbrace{\mathbb{E}_q\left[\log p(y|x, z)\right]}_{\text{reconstruction}} + \underbrace{\mathbb{E}_q\left[\log p(z)\right]}_{\text{regularizer}} \end{aligned} \tag{3}$$

Above we have dropped the term which encourages max-entropy variational approximations ($\mathbb{H}\left[q\right]$), partly for convenience, and partly because we are concerned with maximum a posteriori (MAP) inference, for which a single point estimate of the posterior suffices.

The reconstruction term corresponds to the squared loss in Equation 1, given a suitable definition of $p(y|x, z)$.[1] Because our latent variable z is discrete, we use a neural network $r(y, s)$ to predict the parameters of multinomial distributions

[1]The negative log likelihood of a normal distribution corresponds to squared loss up to an additive constant

$$\begin{aligned}
[\![z]\!](s) &= \mathrm{Exec}(z, s, L+V) \quad \textit{execute program and extract output on line } L+V \\
\mathrm{Exec}(z, s, l) &= s_l, \text{ whenever } l \le V \quad \textit{load symbolic variables as first lines of code. We have } V \textit{ variables} \\
\mathrm{Exec}(z, s, l) &= \sum_o z^O_{lo} \times F_o\left(s, \sum_{1\le a<l} z^L_{la} \times \mathrm{Exec}(z,s,a), \sum_{1\le b<l} z^R_{lb} \times \mathrm{Exec}(z,s,b)\right), \text{ whenever } l > V \\
&\quad \text{where } z \text{ is a tuple of } (z^O, z^L, z^R) \\
F_1(s, A, B) &= A + B \quad \textit{add} \\
F_2(s, A, B) &= A - B \quad \textit{subtraction} \\
F_3(s, A, B) &= A \times B \quad \textit{multiplication} \\
F_4(s, A, B) &= A \quad \textit{no-op/skip connection}
\end{aligned}$$

Figure 3. Differentiable execution model for a program sketch containing L lines of code. z parametrizes the program via a triple of 2-dimensional arrays (z^O, z^L, z^R) containing values from 0-1. If $z^O_{lo} = 1$, then line l of the program computes its value by executing operator o. If $z^L_{la} = 1$, then line l of the program gets its left argument for the operator from line a. If $z^R_{lb} = 1$, then line l of the program gets its rights argument for the operator from line b. The first V lines of the program evaluate to variables stored in the symbolic representation, and we assume that there are V such variables and L lines of code that follow.

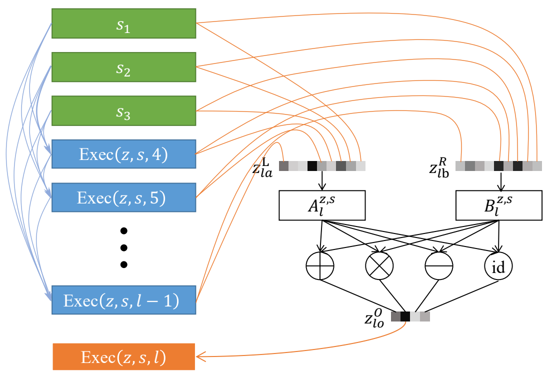

Figure 4. Illustration of the differentiable execution model and overparameterization. As depth l increases, there are exponentially more ways to implement the same program using the straight-line-code execution model.

over the entries of z:

$$p(y|x,z) = \mathcal{N}\left(y|\mu = [\![z]\!](s)\ ;\sigma = \sigma^t\right) \tag{4}$$

$$\begin{aligned}
q(z|y,x) = \prod_l &\mathrm{Multinomial}(z^O_l | r(y,s)^O_l) \\
&\times \mathrm{Multinomial}(z^R_l | r(y,s)^R_l) \\
&\times \mathrm{Multinomial}(z^L_l | r(y,s)^L_l)
\end{aligned} \tag{5}$$

where σ^t is the standard deviation of y for task t. Recall $s = g(x)$, and see Figure 3 for the structure of z.

We optimize Equation 3 w.r.t. the parameters of g and r, which are the only neural networks in our setup. We use the reparameterization trick to obtain stable gradients through the sampling process for optimization [22]. Typically, we use the gumbel softmax trick for each discrete multinomial distribution as defined in Equation 5.

This probabilistic framing ties together **Amortization** (by training q) and multitasking, by assuming we have a corpus of $\{(x_n, y_n)\}$ pairs, for which we seek to maximize $\sum_n \log p(y_n|x_n)$ using stochastic gradient descent. **Regularization** enters through a suitable definition of the prior over programs, $p(z)$:

$$p(z) \propto \exp\left(-\lambda \cdot \mathrm{length}(z)\right) \tag{6}$$

where λ is a regularization coefficient and $\mathrm{length}(z)$ is the length of the program encoded by z. Program length can be recursively computed from z in a manner analogous to how z is recursively executed.

Model Structure of the Program Synthesizer. The program synthesizer predicts the distribution of programs given output and grounded symbols $z \sim q(\cdot|r(y,s))$. One single symbol-output pair leaves much ambiguity for program inference. Therefore, we use multiple symbol-output pairs (together as a set) from the same task to predict the program distribution as $z \sim q(\cdot|r(\{y^i, s^i\})$. Noticing that this is a mapping from a set of vectors $\{(y^i, s^i)\}$ to one vector z, we adopt PointNet [24] to build the model. (The descriptions and justification about PointNet are stated in Appendix A).

Our program synthesizer model first extract features for the properties of the outputs y^i, the relationships between the j-th symbols s^i_j and outputs, and the properties of the whole symbol-output pairs, using PointNets:

$$\begin{aligned}
h_{\text{out}} &= \max_i \mathrm{MLP}_{\text{out}}\left(y^i\right), \\
h_{\text{rel}}[j] &= \max_i \mathrm{MLP}_{\text{rel}}\left(\left[y^i; s^i_j\right]\right), \ \forall j \in \{1, 2, \cdots, V\}, \\
h_{\text{io}} &= \max_i \mathrm{MLP}_{\text{io}}\left(\left[y^i; s^i\right]\right),
\end{aligned}$$

where MLP_{out}, MLP_{rel}, and MLP_{io} are three multi-layer perceptron networks, $[\,\cdot\,;\,\cdot\,]$ means concatenation of the components, V is the dimensionality of the symbolic representations s, and s^i_j is the j-th dimensional element of s^i.

The features are then concatenated together to predict the distribution of the programs z as:

$$z \sim q\left(\cdot|\text{MLP}_{\text{prog}}\left([h_{\text{out}}; h_{\text{io}}; h_{\text{rel}}[0]; h_{\text{rel}}[1]; \cdots; h_{\text{rel}}[V]]\right)\right),$$

where MLP_{prog} is another MLP network for predicting the program distributions according to the features.

Intuitively, our models can learn informative hints for synthesizing programs, such as the outputs y^i are never prime, the outputs y^i are always bigger than the second symbols s^i_2, and the outputs y^i are always equal to the summation of all symbols $\sum_{j=1}^{V} s^i_j$.

4 Experimental Results

We seek to answer the following questions in our experiments: (1) Is it feasible to jointly learn the symbolic representations from high-dimensional perceptual inputs and synthesize the programs that operate on them (Sec 4.2)? (2) Are the techniques we introduce beneficial (Sec 4.2.1)? Specifically, what are the effects of multitasking with amortized inference, overparametrization, and regularization? (3) Can the generalizability of models be improved by learning interpretable symbolic representations with the programmatic prior (Sec 4.3)?

4.1 Experimental Setup

Task and Dataset. As illustrated in Figure 1, in the neurosymbolic program synthesis problem, we are given dataset containing samples as I/O pairs, $\left(x^{t,i}_1, x^{t,i}_2, \cdots, x^{t,i}_V, y^{t,i}\right)$, where t, i index tasks and samples within tasks, respectively, and we assume the I/O pairs are generated using gold standard symbolic representations, $s^{*,t,i}_j = g^*(x^{t,i}_j)$, and gold standard programs $z^{*,t}$ as $y^{t,i} = [\![z^{*,t}]\!]\left(s^{*,t,i}_1, s^{*,t,i}_2, \cdots, s^{*,t,i}_V\right)$. We want to train models to jointly infer the symbolic representations $s^{t,i}_j = g(x^{t,i}_j)$ and the program $\hat{z}^t = \arg\max_z q(z|r(\{s^{t,i}_j, y^{t,i}_j\})$ only given distant supervision via the I/O pairs. Success means that the symbolic representations are close to the gold standard ones, $s^{t,i}_j \simeq s^{*,t,i}_j$, and the synthesized programs are equivalent to the gold standard ones, $[\![\hat{z}^t]\!](s) = [\![z^{*,t}]\!](s), \forall s \in \mathbb{R}^V$, as shown in Figure 2.

We build the dataset by firstly drawing gold standard symbols, $s^{*,\cdot,i}_j$, uniformly from $\{1, 2, \cdots, 10\}$ for all $i \leq N$ and $j \leq V$, where N is the number of I/O pairs for each task and $V = 3$ is the number of symbols for each input. The placeholder $\cdot$ denotes that we share the same set of symbols $s^{*,\cdot,i}_j$ and later inputs $x^{\cdot,i}_j$ for all tasks t, for computational efficiency issues. The perceptual inputs $x^{\cdot,i}_j$ are then drawn randomly using the CIFAR-10 dataset [23]. Specifically, CIFAR-10 dataset contains images in 10 classes, $\{(\text{img}^k, l^k)\}$, where $\text{img}^k \in \mathbb{R}^{32\times32\times3}$ are images and $l^k \in \{1, 2, \cdots, 10\}$ are their labels. The perceptual inputs $x^{\cdot,i}_j$ are then drawn uniformly from all images img^k with label l^k equal to $s^{*,\cdot,i}_j$. The programs $z^{*,t}$ are drawn uniformly from all the semantically distinct arithmetic formulas with at most 3 variables and 3 operators, such as s_1, $s_2 + s_3$, and $(s_1 + s_3) \times (s_1 - s_3)$. The outputs $y^{t,i}$ are then calculated by executing programs on the corresponding inputs as $y^{t,i} = [\![z^{*,t}]\!](s^{*,\cdot,i})$, for all $t \leq T$, where $T = 500$ is the number of tasks.

Evaluation Metrics. We evaluate the abilities of models on grounding symbols and synthesizing programs using the following metrics:

- Program success rate: the ratio that the synthesized programs $\hat{z}^t$ are equivalent to the gold standard programs $z^{*,t}$. In practice, we empirically evaluate the equivalence of programs using the samples in the dataset, i.e., $\sum_{t\leq T} \mathbf{1}([\![\hat{z}^t]\!](s^{*,\cdot,i}) = y^{t,i}, \forall i \leq N)\ /\ T$.
- Symbolic loss: the mean squared error (MSE) between the grounded symbols $s^{\cdot,i}_j$ and the gold standard symbols $s^{*,\cdot,i}_j$, i.e., $\frac{1}{NV}\sum_{i\leq N}\sum_{j\leq V}\left(g(x^{\cdot,i}_j) - s^{*,\cdot,i}_j\right)^2$.
- Test loss: the mean squared error of applying the whole model to predict the outputs, i.e., $\frac{1}{TN}\sum_{t\leq T}\sum_{i\leq N}\left([\![\hat{z}^t]\!](g(x^{t,i})) - y^{t,i}\right)^2$.

Note that, although we are evaluating models using the gold standard symbols $s^{*,\cdot,(i)}_j$, they are not available during training and we must learn to ground them in order to achieve good performances on all metrics. Meanwhile, we use different images and random symbolic inputs for generating the training dataset and the testing dataset in order to evaluate models' robustness against the perceptual and sampling noises (CIFAR-10 dataset is split into 50000 samples for training and 10000 samples for testing). All the models and metrics are evaluated on the testing dataset.

We also introduce another metric for evaluating the improvements of models' generalizability from learning interpretable symbolic representations with the programmatic prior. In particular, for a new task t', we can directly apply our learned symbol grounder g to get the symbolic representation $s^{t',i}_j$ of high-dimensional images $x^{t',i}_j$, and synthesize programs using the symbol-output pairs. To evaluate the out-of-distribution generalizability of such models, we train them on dataset with symbols smaller or equal to 5, $s^{t',i}_j \leq 5, \forall i, j$, but test them on dataset with symbols larger than 5, $s^{t',i}_j > 5, \forall i, j$. The evaluation metric is then

- Generalization loss: the mean squared error of applying the model trained on smaller digits to predict outputs on larger digits, i.e., $\frac{1}{N}\sum_{i\leq N}\left([\![\hat{z}^{t'}]\!](s^{t',i}) - y^{t',i}\right)^2$.

Notice that when given the learned symbol grounder, the neurosymbolic program synthesis problem reduces to the classical symbolic program synthesis problem. Any classical algorithms can be applied. In practice, we use the bottom-up enumerative search [30] to find the program for the new task.

Implementation Details. The framework of our models is illustrated in Figure 2 and described in Sec 3. We apply the standard 18-layer ResNet [18], which is a classical convolutional neural network, to build the symbol grounder[2]. All MLPs used in the program synthesizer model have two hidden layers of size 256. Layer normalization [1] is applied before each activation function, ReLU, to improve the training stability [3]. The program sketch contains $L = 30$ lines of code.

Given the large and complex search space of models, there are non-negligible probabilities that a run of the experiment would fail with loss=infinity and gradients equal to nan due to the numerical issues[4]. To tackle this numerical issue, we instead squash the range of symbols from $\{1, 2, \cdots, 10\}$ to $\{0.1, 0.2, \cdots, 1.0\}$ by dividing by ten. Also, for each model, we run the experiments for multiple times (~ 3) and report the best one. We select the best learned model using the mean square error of applying the learned model, $g(\cdot), \hat{z}$, to predict outputs in the training dataset.

We train models using the Adam [21] optimizer with a learning rate equal to 3e-4 and ϵ =1e-5 for 20 epochs. The dataset contains 1e6 I/O pairs for each task for training and 1000 I/O pairs for each task for testing. The program length regularizer is not applied until epoch 10 with ratio λ =2e-4. The temperature for gumbel softmax is set to 1 in the beginning and changed to 3 from epoch 15 to minimize the error gap from the continuous approximations of programs near the end of training.

Baselines. To analyze the effects of the programmatic prior and the other techniques as described in Section 3, we compare our model with the following baselines:

- without program structure: These models use neural networks (typically MLPs) instead of programs to predict outputs from symbols as $\hat{y} = \text{MLP}(s)$. We implement two types of models to incorporate the task information and report the best of them: (1) disjoint MLPs for each task, i.e., $\hat{y}^t = \text{MLP}^t(s)$; and (2) one-hot embedding of the task id as features, i.e., $\hat{y}^t = \text{MLP}([s; \text{one-hot}(t)])$.
- without amortized inference: These models drop the program synthesizer r and instead take z^t as free independent parameters. Therefore, there will be no information shared by amortized inference among the tasks.
- without gumbel softmax: These models replace gumbel softmax with softmax. So, there is no stochastic sampling, $z \sim q(z|r(y, s))$, in these models.
- with Syntax-Tree: These models use the syntax tree execution model instead of straight-line code. More details of the syntax tree execution model are stated in Appendix B. Briefly speaking, the syntax tree execution model has fewer optimal solutions in its solution space and, therefore, is less overparameterized.
- with Depth-D: The program sketch contains $L = D$ lines of code, instead of $L = 30$. Larger depths correspond to more overparameterized models.

[2]The output dimensionality of the last layer of ResNet is changed to predict symbols instead of image labels.
[3]BatchNorm [20] can not be easily applied in the multi-tasking setting.
[4]For example, when the symbols are predicted as 10 and the currently synthesized program is s_0^{20}, it is easy for the models' outputs to be overflow and fail the training

4.2 Joint Learning of Symbols and Programs

We demonstrate that it is feasible to jointly learn interpretable symbolic representations and synthesize programs that operate on them using our model. As shown in Table 1, our model successfully learns the symbolic representations and correctly synthesizes the programs for most tasks. The symbolic loss, 0.00086, is negligible for distinguishing symbols in the set $\{0.1, 0.2, \cdots, 1.0\}$. And we correctly find the program for 459 of 500 tasks, which corresponds to the program success rate as 91.8%.

4.2.1 Ablation Studies.

We analyze the effectiveness of techniques by performing the ablation studies and comparing our model with baselines. The experimental results show that all of our techniques, including multi-tasking with amortized inference, overparameterization, gumbel-softmax, and the program length regularizer, improve performances of models in the neurosymbolic program synthesis problem.

Effectiveness of multi-tasking with amortized inference. As shown in Table 1, amortized inference with multi-tasks largely improves the performance of program synthesis and then the learning of interpretable symbolic representations. Without amortized inference, the model can only synthesize programs correctly for 136/500 tasks. The symbolic loss is also non-negligible as 0.059. It is because amortized inference can share information while synthesizing programs for multiple tasks and, therefore, alleviates the difficulties of program synthesis and then joint learning. More correctly synthesized programs can drive better symbolic representation learning by triangulating a good symbolic basis. Better symbolic representations, in return, enable even better program synthesis by providing more accurate program inputs. We also compare the performances of joint learning in one task, i.e., $T = 1$, and 500 tasks in Table 2. We run each experiment more than 10 times, each time with randomly drawn programs, and calculate the success rate metric as the ratio of runs that learn the symbolic representations successfully, i.e., symbolic loss ≤ 0.05. We also report the best symbolic loss of both settings in Table 2. The results suggest that multi-tasking can largely improves the success rate of joint learning and also their performances.

Table 1. The performances of models for the neurosymbolic program synthesis problem. Our model (ROAP) successfully learned the interpretable symbolic representations (negligible symbolic loss) and the programs that operate on them (high program success rate). The learned model can predict outputs from input images with small errors on both within-distribution test dataset (test loss) and out-of-distribution test dataset with unseen larger digits (generalization loss). All techniques, including the programmatic prior, multi-tasking with amortized inference, and overparameterization, are beneficial in terms of the joint learning performances by comparing our model with the baselines.

	program success rate	symbolic loss	test loss	generalization loss
ROAP (our model)	**459** / 500	**0.00086**	**0.11**	**0.07**
w/o program, i.e., CNN+MLP	0 / 500	0.95	**0.11**	1.03
w/o amortized inference	136 / 500	0.059	0.20	0.19
w/o gumbel-softmax	8 / 500	0.52	0.33	4.40
w/ Syntax-Tree	345 / 500	0.04	0.16	0.22
w/ depth=10	58 / 500	0.90	0.14	2.12
w/ depth=3	56 / 500	1.10	0.16	2.08

Table 2. Effects of multitask learning for joint learning.

	success rate	symbolic loss
500 tasks	**33.3%**	**0.00086**
1 task	10.5%	0.039

Effectiveness of overparameterization. Baseline models, with Syntax-Tree, with depth=10, and with depth=3, are all less overparameterized than our model. As listed in Table 1, our overparameterized model largely outperforms these baselines in terms of all metrics. These results suggest that overparameterization helps improve the performances of joint learning, probably by providing more global optimal solutions in the solution space and avoiding converging to local optimums. We further modify the degrees of overparameterization by changing the depth L of program sketches and report their performances for tasks with different difficulties in Figure 5. The results first show that more overparameterized models perform better than less overparameterized models. Moreover, less overparameterized models perform better on easier tasks, i.e., when the relative overparameterization degree (the size of paragram sketch versus the size of correct programs) becomes larger, which also demonstrates the effectiveness of overparameterization.

Effectiveness of gumbel softmax. As shown in Table 1, models without gumbel softmax perform much worse than ours. It is because the stochastic sampling noises introduced by gumbel softmax help models to explore a larger solution space and then find a better converged point while using the local optimizers such as Adam.

Effectiveness of program length prior. We demonstrate the effects of our program length regularizer in three scenarios in Figure 6. When the model has already learned good symbols and program synthesizers (blue lines), applying the program length prior will not hurt the models' performances

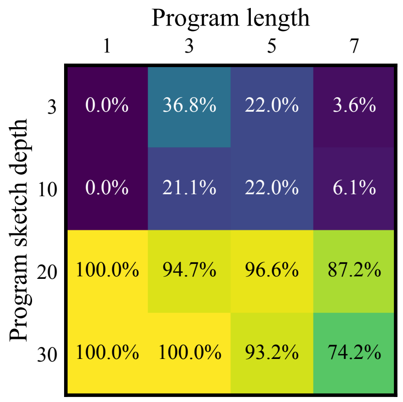

Figure 5. Overparameterization is necessary for robust program recovery. Programs get depth corresponds to the number of possible lines of code: increasing sketch depth increases overparameterization. In each cell, we show the percentage of programs of a given size successfully synthesized

while can largely improve the models' interpretability by inducing synthesizing shorter programs. The averaged length of synthesized programs reduces from 21.6 to 9.6 after applying the regularization, while the program success rate increases from 431 to 455 / 500. For runs with poor performances, applying the program length regularization can largely boost the performances of models by providing more programmatic prior for salvageable runs (the orange lines), but not always (the green lines). Specifically, for salvageable runs, the program success rate increases from 4 to 412 / 500 and the symbolic loss decreases from 1.23 to 0.066.

4.3 Out-of-Distribution Generalizability by Programs

We demonstrate the improvements of out-of-distribution generalizability of models from the programmatic prior using

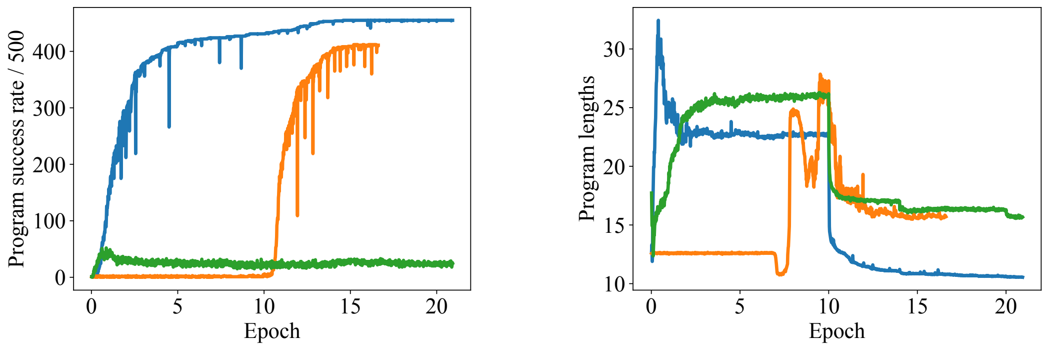

Figure 6. Effects of the program length prior on model convergence. The regularizer is applied starting from epoch 10. Left: number of program synthesis tasks solved during training. Right: average program length during training. On successful runs (blue), the regularizer maintains performance while dramatically decreasing program length. On 'salvageable' runs (orange) the regularizer pushes the model into successful solutions, but not always (green)

the generalization loss metric. The generalization loss metric evaluates the models' performances on unseen symbols (0.6 to 1.0) that are larger than the symbols (0.1 to 0.5) used during training for a *new* task, having already seen those symbols used in *old* tasks. As shown in Table 1, without the programmatic prior, models cannot learn the correct symbolic representations and synthesize the correct programs even though training and test losses are low. Because of the large flexibility of neural networks, models without a programmatic prior can easily overfit and achieve low training losses without learning the correct representations and programs. On the other hand, our model with the programmatic prior can successfully learn the symbolic representations and programs while still achieving the same test loss. The symbol grounder is shared among multiple tasks so that it can learn to ground unseen symbols for the new task. More importantly, due to the generalizability of programs, our programmatic model can generalize to unseen domains (I/O pairs with larger digits) and achieve generalization loss as low as 0.07 (compared with 1.03 for models without the programmatic prior).

5 Conclusion

Our goal is to make progress on basic neurosymbolic problems: starting from perception, and absent symbol-level supervision, how can we discover grounded symbols and the symbolic routines which manipulate them? Our experiments examine a small-scale arithmetic domain, and assumed the premise that end-to-end deep learning could be made to work here. Even in this simplistic domain, joint perceptual learning and inductive reasoning proved difficult without additional tricks.

Many directions remain open. One low hanging fruit is pretraining the perceptual frontend $g(\cdot)$ with self supervised representation learning. But we see the main open question as whether a pure gradient-guided search is actually superior to a hybrid alternating maximization scheme, or a pure symbolic search.

To what extent do the ROAP tricks generalize? This is an empirical question, and we are actively exploring new application areas to answer exactly that question. But we are optimistic of the general applicability of the ROAP recipe, because we believe that each trick can be motivated from first principles: (1) Multitask learning through amortization allows sharing of statistical strength from easy tasks to hard tasks, which should facilitate bootstrap learning. Multitasking also constrains the symbolic representation from different directions, 'triangulating' the correct symbols for the systems to be using. (2) Overparameterization is necessary for reliable neural network convergence, and unsurprisingly also necessary for gradient-guided program search. (3) Regularizing program length has solid theoretical underpinnings [27]. Although our main demonstration is small-scale, we hope that the ideas we pilot could apply more broadly.

6 Broader Impact

Joint learning of symbolic representations and symbolic programs that operate on them, once achieved, can largely improve models' interpretability and generalizability due to the programmatic prior, while still maintaining scalability to handle noisy high-dimensional perceptual data using neural networks with gradient descent. Human interventions and safety constraints/verification can be further introduced through the programmatic prior. Therefore, efforts on enabling joint learning of symbols and programs, such as ours, can lead to more usages of machine learning in high-stakes and complex environments such as the medical settings, road, and commerce, where sensory inputs are noisy and high-dimensional, and interpretability and safety are critical. Joint inference of symbols and programs can also help scientific

$$
\begin{aligned}
[\![z]\!](s) &= \mathrm{Exec}(z, s, 0) \quad \textit{execute program and extract output on the root node with } d = 0 \\
\mathrm{Exec}(z, s, path) &= \sum_o z^{path} \times F_o\left(s, \quad \mathrm{Exec}(z, s, path.0), \quad \mathrm{Exec}(z, s, path.1)\right), \text{ whenever } \mathrm{len}(path) < D \\
\mathrm{Exec}(z, s, path) &= \sum_{1 \le j \le V} z_j^{path} \times s_j, \text{ whenever } \mathrm{len}(path) = D \quad \textit{load symbolic variables in leaf nodes} \\
&\text{where } z \text{ is a tuple of } (z^0, z^{0.0}, z^{0.1}, z^{0.0.0}, z^{0.0.1}, \cdots) \\
F_1(s, A, B) &= A + B \quad \textit{add} \\
F_2(s, A, B) &= A - B \quad \textit{subtraction} \\
F_3(s, A, B) &= A \times B \quad \textit{multiplication} \\
F_4(s, A, B) &= A \quad \textit{no-op/skip connection}
\end{aligned}
$$

Figure 7. The syntax-tree execution model for a program sketch with a depth-D syntax tree. z parametrizes the program by specifying the operator for all internal nodes in the syntax tree, z^{path} when $\mathrm{len}(path) < D$, and specifying the symbolic variables for all leaf nodes, z^{path} when $\mathrm{len}(path) = D$.

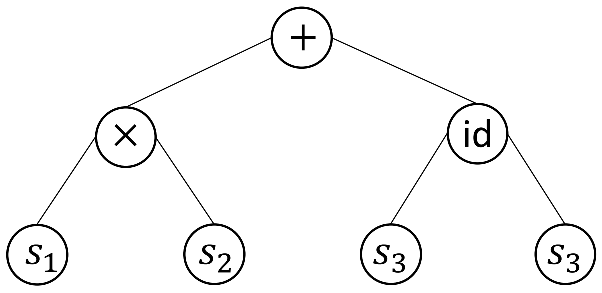

Figure 8. An example of the syntax trees for $y = s_1 \times s_2 + s_3$. The symbolic variables are loaded in the leaf nodes and the operators are assigned in the internal nodes.

discovery by searching for interpretable and generalizable features to explain nature. On the other hand, interpretable feature extraction and convenient human intervention can be exploited to collect and analyze personal information and for malicious rumor spreading.

Acknowledgments

We are grateful for helpful discussions with Owen Lewis about overparameterization.

A PointNet [24]

PointNet [24] learns discriminative features of a set of vectors $\{v^i\}$ by firstly individually transforming the vectors into features using a Multi-Layer Perceptron (MLP) network and then outputting the maximum of them:

$$h = \max_i \mathrm{MLP}\left(v^i\right).$$

Given the properties of max, PointNet is permutation invariant to the indexing i of the set elements v^i and can handle input sets of any size. Together with the powerful feature learning abilities of MLPs, PointNet then becomes a popular choice for learning features of sets of vectors.

B The Syntax-Tree execution model

As shown in Figure 7 and Figure 8, we implement a differentiable execution model using syntax trees as a baseline. At each internal node, the syntax model only uses z^{path} to parameterize the choice of operators since it has fixed choices of the left and right arguments as its left and right nodes' outputs. On the other hand, our straight-line code model has extra freedom to select each node/line's arguments using the parameterization, z^L and z^R. Our model is thus more overparameterized than the baseline, with syntax-tree, when $L \simeq 2^D$. In practice, we evaluate baseline models with depth $D = 2, 3, 5$ and report the best performances among them.

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).
[2] Oscar Chang, Lampros Flokas, Hod Lipson, and Michael Spranger. 2020. Assessing SATNet's ability to solve the symbol grounding problem. *Advances in Neural Information Processing Systems* 33 (2020), 1428–1439.
[3] Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, and Yisong Yue. 2021. Neurosymbolic Programming. *Foundations and Trends® in Programming Languages* 7, 3 (2021), 158–243. https://doi.org/10.1561/2500000049
[4] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. *ICML* (2017).
[5] Simon Du, Jason Lee, Haochuan Li, Liwei Wang, and Xiyu Zhai. 2019. Gradient descent finds global minima of deep neural networks. In *International conference on machine learning*. PMLR, 1675–1685.
[6] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. *NIPS* (2018).
[7] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. *DreamCoder: Bootstrapping Inductive*

Program Synthesis with Wake-Sleep Library Learning. Association for Computing Machinery, New York, NY, USA, 835–850. https://doi.org/10.1145/3453483.3454080

[8] Richard Evans, Matko Bošnjak, Lars Buesing, Kevin Ellis, David Pfau, Pushmeet Kohli, and Marek Sergot. 2021. Making sense of raw input. *Artificial Intelligence* 299 (2021), 103521. https://doi.org/10.1016/j.artint.2021.103521

[9] Richard Evans and Edward Grefenstette. 2018. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research* 61 (2018), 1–64.

[10] Jonathan Frankle and Michael Carbin. 2018. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635* (2018).

[11] Alexander L Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. 2017. Differentiable programs with neural libraries. In *International Conference on Machine Learning*. PMLR, 1213–1222.

[12] Alexander L Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. 2016. TerpreT: A Probabilistic Programming Language for Program Induction. *arXiv preprint arXiv:1608.04428* (2016).

[13] Samuel Gershman and Noah Goodman. 2014. Amortized inference in probabilistic reasoning. In *Proceedings of the annual meeting of the cognitive science society*, Vol. 36.

[14] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural turing machines. *arXiv:1410.5401* (2014).

[15] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[16] Moritz Hardt and Tengyu Ma. 2016. Identity matters in deep learning. *arXiv preprint arXiv:1611.04231* (2016).

[17] Stevan Harnad. 1990. The symbol grounding problem. *Physica D: Nonlinear Phenomena* 42, 1-3 (1990), 335–346.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[19] Luke Hewitt and Joshua Tenenbaum. 2019. Learning structured generative models with memoised wake-sleep. *under review* (2019).

[20] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*. PMLR, 448–456.

[21] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[22] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[23] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[24] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. 2017. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 652–660.

[25] Ameesh Shah, Eric Zhan, Jennifer Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. 2020. Learning differentiable programs with admissible neural heuristics. *Advances in neural information processing systems* 33 (2020), 4940–4952.

[26] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *ACM Sigplan Notices*, Vol. 41. ACM, 404–415.

[27] Ray J Solomonoff. 1964. A formal theory of inductive inference. *Information and control* 7, 1 (1964), 1–22.

[28] Sever Topan, David Rolnick, and Xujie Si. 2021. Techniques for Symbol Grounding with SATNet. *Advances in Neural Information Processing Systems* 34 (2021).

[29] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 287–296. https://doi.org/10.1145/2491956.2462174

[30] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. *ACM SIGPLAN Notices* 48, 6 (2013), 287–296.

[31] Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. 2018. Houdini: Lifelong learning as program synthesis. In *Advances in Neural Information Processing Systems*. 8687–8698.