Probabilistic Neural Programs

Anonymous Author(s)
Affiliation
Address
email

Abstract

We present probabilistic neural programs, a framework for program induction that permits flexible specification of both a computational model and inference algorithm while simultaneously enabling the use of deep neural networks. Probabilistic neural programs combine a computation graph for specifying a neural network with an operator for weighted nondeterministic choice. Thus, a program describes both a collection of decisions as well as the neural network architecture used to make each one. We evaluate our approach on a challenging diagram question answering task where probabilistic neural programs correctly execute nearly twice as many programs as a baseline model.

1 Introduction

In recent years, deep learning has produced tremendous accuracy improvements on a variety of tasks in computer vision and natural language processing. A natural next step for deep learning is to consider *program induction*, the problem of learning computer programs from (noisy) input/output examples. Compared to more traditional problems, such as object recognition that require making only a single decision, program induction is difficult because it requires making a sequence of decisions and possibly learning control flow concepts such as loops and if statements.

Prior work on program induction has described two general classes of approaches. First, in the noise-free setting, program synthesis approaches pose program induction as completing a program "sketch," which is a program containing nondeterministic choices ("holes") to be filled by the learning algorithm [13]. Probabilistic programming languages generalize this approach to the noisy setting by permitting the sketch to specify a *distribution* over these choices as a function of prior parameters and further to condition this distribution on data, thereby training a Bayesian generative model to execute the sketch correctly [6]. Second, neural abstract machines define continuous analogues of Turing machines or other general-purpose computational models by "lifting" their discrete state and computation rules into a continuous representation [9, 11, 7, 12]. Both of these approaches have demonstrated success at inducing simple programs from synthetic data but have yet to be applied to practical problems.

We observe that there are (at least) three dimensions along which we can characterize program induction approaches:

1. Computational Model – what abstract model of computation does the model learn to control? (e.g., a Turing machine)
2. Learning Mechanism – what kinds of machine learning models are supported? (e.g., neural networks, Bayesian generative models)
3. Inference – how does the approach reason about the many possible executions of the machine?

Submitted to Neural Abstract Machines & Program Induction (NAMPI 2016) Workshop. Do not distribute.

```
def mlp(v: Tensor):
  Pp[CgNode] =
  for {
    w1 <- param("w1")
    b1 <- param("b1")
    h1 = ((w1 * v) + b1).tanh
    w2 <- param("w2")
    b2 <- param("b2")
    out = (w2 * h1) + b2
  } yield {
    out
  }
```

```
val dist: Pp[Int] = for {
  s <- mlp(new Tensor(...))
  v <- choose(Array(0, 1), s)
  y <- choose(Array(2, 3), s)
} yield {
  v + y
}
// tensor parameters initialized to 0
val params: NnParams
println(dist.beamSearch(10, params))
// output: 2 (0.25), 3 (0.25),
//         3 (0.25), 4 (0.25)
```

Figure 1: Probabilistic neural programs defining a multilayer perceptron (left) and applying it to create a probability distribution over program executions (right).

Neural abstract machines conflate some of these dimensions: they naturally support deep learning, but commit to a particular computational model and approximate inference algorithm. These choices are suboptimal as (1) the bias/variance trade-off suggests that training a more expressive computational model will require more data than a less expressive one suited to the task at hand, and (2) recent work has suggested that discrete inference algorithms may outperform continuous approximations [5]. In contrast, probabilistic programming supports the specification of different (possibly task-specific) computational models and inference algorithms, including discrete search and continuous approximations. However, these languages are restricted to generative models and cannot leverage the power of deep neural networks.

We present probabilistic neural programs, a framework for program induction that permits flexible specification of the computational model and inference algorithm while simultaneously enabling the use of deep neural networks. Our approach builds on computation graph frameworks [1, 3] for specifying neural networks by adding an operator for weighted nondeterministic choice that is used to specify the computational model. Thus, a program sketch describes *both* the decisions to be made and the architecture of the neural network used to score these decisions. Importantly, the *computation graph interacts with nondeterminism*: the scores produced by the neural network determine the weights of nondeterministic choices, while the choices determine the network's architecture. As with probabilistic programs, various inference algorithms can be applied to a sketch. Furthermore, a sketch's neural network parameters can be estimated using stochastic gradient descent from either input/output examples or full execution traces.

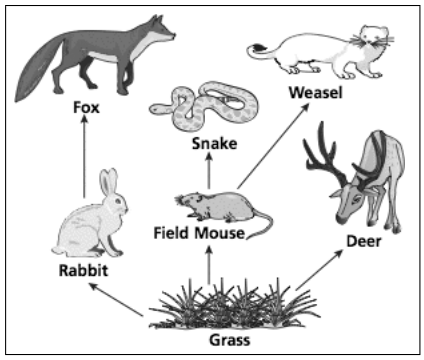
We evaluate our approach on a challenging diagram question answering task, which recent work has demonstrated can be formulated as learning to execute a certain class of probabilistic programs. On this task, we find that the enhanced modeling power of neural networks improves accuracy.

2 Probabilistic Neural Programs

Probabilistic neural programs build on computation graph frameworks for specifying neural networks by adding an operator for nondeterministic choice. We have developed a Scala library for probabilistic neural programming that we use to illustrate the key concepts.

Figure 1 (left) defines a multilayer perceptron as a probabilistic neural program. This definition resembles those of other computation graph frameworks. Network parameters and intermediate values are represented as computation graph nodes with tensor values. They can be manipulated with standard operations such as matrix-vector multiplication and hyperbolic tangent. Evaluating this function with a tensor yields a program sketch object that can be evaluated with a set of network parameters to produce the network's output.

Figure 1 (right) shows how to use the `choose` function to create a nondeterministic choice. This function nondeterministically selects a value from a list of options. The score of each option is given by the value of a computation graph node that has the same number of elements as the list. Evaluating this function with a tensor yields a program sketch object that represents a function from neural network parameters to a *probability distribution* over values. The log probability of a value is proportional to the sum of the scores of the choices made in the execution that produced it.



What happens to the snake population if the field mouse population decreases?
$\lambda f.\texttt{cause}(\texttt{decrease}(\texttt{mice}), f(\texttt{snakes}))$

Does the deer eat the grass?
$\texttt{eats}(\texttt{deer}, \texttt{grass})$

How many organisms eat the grass?
$\texttt{count}(\lambda x.\texttt{eats}(x, \texttt{grass}))$

Are rabbits a tertiary consumer?
$\texttt{tertiary-consumer}(\texttt{rabbits})$

Figure 2: A food web diagram with annotations generated from a computer vision system (left) along with related questions and their associated program sketches (right).

Performing (approximate) inference over this object – in this case, using beam search – produces an explicit representation of the distribution. Multiple nondeterministic choices can be combined to produce more complex sketches; this capability can be used to define complex computational models, including general-purpose models such as Turing machines. The library also has functions for conditioning on observations.

Although various inference algorithms may be applied to a program sketch, in this work we use a simple beam search over executions. This approach accords with the recent trend in structured prediction to combine greedy inference or beam search with powerful non-factoring models [2, 10, 4]. The beam search maintains a queue of partial program executions, each of which is associated with a score. Each step of the search continues each execution until it encounters a call to `choose`, which adds zero or more executions to the queue for the next search step. The lowest scoring executions are discarded to maintain a fixed beam width. As an execution proceeds, it may generate new computation graph nodes; the search maintains a single computation graph shared by all executions to which these nodes are added. The search simultaneously performs the forward pass over these nodes as necessary to compute scores for future choices.

The neural network parameters are trained to maximize the loglikelihood of correct program executions using stochastic gradient descent. Each training example consists of a pair of program sketches, representing an unconditional and conditional distribution. The gradient computation is similar to that of a loglinear model with neural network factors. It first performs inference on both the conditional and unconditional distributions to estimate the expected counts associated with each nondeterministic choice. These counts are then backpropagated through the computation graph to update the network parameters.

3 Diagram Question Answering with Probabilistic Neural Programs

We consider the problem of learning to execute program sketches in a food web computational model using visual information from a diagram. This problem is motivated by recent work [8], which has demonstrated that diagram question answering can be formulated as translating natural language questions to program sketches in this model, then learning to execute these sketches. Figure 2 shows some example questions from this work, along with the accompanying diagram that must be interpreted to determine the answers. The diagram (left) is a food web, which depicts a collection of organisms in an ecosystem with arrows to indicate what each organism eats. The right side of the figure shows questions pertaining to the diagram and their associated program sketches.

The possible executions of each program sketch are determined by a domain-specific computational model that is designed to reason about food webs. The nondeterministic choices in this model correspond to information that must be extracted from the diagram. Specifically, there are two functions that call `choose` to nondeterministically return a boolean value. The first function, $\texttt{organism}(x)$, should return true if the text label x is an organism (as opposed to e.g., the image title). The second function, $\texttt{eat}(x, y)$, should return true if organism x eats organism y. These functions do influence program control flow. The food web model also includes various other functions, e.g., for reasoning about population changes, that call `organism` and `eat` to extract

Method	Execution Accuracy	`choose` Accuracy
LOGLINEAR	8.6%	78.2%
2-LAYER PNP	12.5%	78.7%
MAXPOOL PNP	**14.9%**	**82.5%**

Table 1: Result of Probabilistic Programs on the executions of sketches from the FOODWEBS dataset. "`choose` Accuracy" refers to overall nondeterminism accuracy, whereas "Execution Accuracy" determines if the entire program was executed correctly.

information from the diagram. [8] has a more thorough description of the theory; our goal is to learn to make the choices in this theory.

We consider three models for learning to make the choices for both `organism` and `eat`: a non-neural (LOGLINEAR) model, as well as two probabilistic neural models (2-LAYER PNP and MAXPOOL PNP). All three learn models for both `organism` and `eat` using outputs from a computer vision system trained to detect organism, text, and arrow relations between them. [8] defines a set of hand-engineered features heuristically created from the outputs of this vision system. LOGLINEAR and 2-LAYER PNP use only these features, and the difference is simply in the greater expressivity of a two-layer neural network. However, one of the major strengths of neural models is their ability to learn latent feature representations automatically, and our third model also uses the direct outputs of the vision system not made into features. The architecture of MAXPOOL PNP reflects this and contains additional input layers that maxpool over detected relationships between objects and confidence scores. The expectation is that our neural network modeling of nondeterminism will learn better latent representations than the manually defined features.

4 Experiments

We evaluate probabilistic neural programs on the FOODWEBS dataset introduced by [8]. This data set contains a training set of ~2,900 programs and a test set of ~1,000 programs. These programs are human annotated gold standard interpretations for the questions in the data set, which corresponds to assuming that the translation from questions to programs is perfect. We train our probabilistic neural programs using correct execution traces of each program, which are also provided in the data set.

We evaluate our models using two metrics. First, execution accuracy measures the fraction of programs in the test set that are executed completely correctly by the model. This metric is challenging because correctly executing a program requires correctly making a number of `choose` decisions. Our 1,000 test programs had over 35,000 decisions, implying that to completely execute a program correctly means getting on average 35 `choose` decisions correct without making any mistakes. Second, `choose` accuracy measures the accuracy of each decision independently, assuming all previous decisions were made correctly.

Table 1 compares the accuracies of our three models on the FOODWEBS dataset. The improvement in accuracy between the baseline (LOGLINEAR) and the probabilistic neural program (2-LAYER PNP) is due to the neural network's enhanced modeling power. Though the `choose` accuracy does not improve by a large margin, the improvements translate into large gains in entire program correctness. Finally, as expected, the inclusion of lower level features (MAXPOOL PNP) not possible in LOGLINEAR significantly improved performance. Note that this task requires performing computer vision, and thus it is not expected that any model achieve 100% accuracy.

5 Conclusion

We have presented probabilistic neural programs, a framework for program induction that permits flexible specification of computational models and inference algorithms while simultaneously enabling the use of deep learning. A program sketch describes a collection of nondeterministic decisions to be made during execution, along with the neural architecture to be used for scoring these decisions. The network parameters of a sketch can be trained from data using stochastic gradient descent. We demonstrate that probabilistic neural programs improve accuracy on a diagram question answering task which can be formulated as learning to execute program sketches in a domain-specific computational model.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. Globally normalized transition-based neural networks. *CoRR*, abs/1603.06042, 2016.

[3] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A CPU and GPU math compiler in Python.

[4] Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. Recurrent neural network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 199–209, San Diego, California, June 2016. Association for Computational Linguistics.

[5] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016.

[6] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Proc. of Uncertainty in Artificial Intelligence*, 2008.

[7] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[8] Jayant Krishnamurthy, Oyvind Tafjord, and Aniruddha Kembhavi. Semantic parsing to probabilistic programs for situated question answering. *EMNLP*, 2016.

[9] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *CoRR*, abs/1511.04834, 2015.

[10] Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülsen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13:95–135, 2007.

[11] Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. *CoRR*, abs/1511.06279, 2015.

[12] Sebastian Riedel, Matko Bošnjak, and Tim Rocktäschel. Programming with a differentiable forth interpreter. *arXiv preprint arXiv:1605.06640*, 2016.

[13] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM.