Semantics-aware program sampling

Pratiksha Thaker
Stanford University
prthaker@stanford.edu

Daniel Tarlow
Google Brain
dtarlow@google.com

Marc Brockschmidt
Microsoft Research
mabrocks@microsoft.com

Alexander Gaunt
Microsoft Research
algaunt@microsoft.com

Abstract

We present an algorithm for specifying and sampling from distributions over programs via a perturb-max approximation. Prior work is generally limited to sampling from priors over program *syntax* (for example, assigning lower probability to longer programs). We demonstrate a simple modification to our sampling algorithm that allows interpolation between such syntactic priors and priors over *semantics* (that is, the behavior of the program on its inputs, or the set of class labels when the program is viewed as a statistical model).

1 Introduction

Machine learning has recently seen a surge of interest in model *interpretability*: understanding the structure and parameters of a learned model to understand data, rather than relying only on the model's predictions. One approach to increasing interpretability is to build additional structure into the model, which can be accomplished via frameworks including probabilistic graphical models [Koller and Friedman, 2009] or, more recently, rule lists [Wang et al., 2016] or probabilistic programs [Gordon et al., 2014, Gaunt et al., 2016]. Programs are particularly appealing models because they are typically designed to be human-readable, with instructions that carry meaningful semantics.

In identifiable model spaces, such as linear regression models, a distribution over syntax (parameters) corresponds directly to a distribution over semantics. Unfortunately, programs – as well as other structured models, including some graphical models – are not identifiable. For instance, a natural programming language might have many ways of representing the zero program, so that these programs are overrepresented in the posterior. Therefore, while priors over program syntax are relatively easy to define, they might cause unintended biases in the resulting marginal distribution over semantics.

More generally, given two sets of variables a and b, it might be easy to define a prior over a, $P(a)$, and a model that describes $P(b|a)$. These then imply $P(b) = \sum_a P(b|a)P(a)$. If this sum is intractable to compute, $P(b)$ may become difficult to reason about, and intuitive priors on $P(a)$ may lead to unintended consequences for this marginal (in the program synthesis example, overweighting trivial programs). A simple way to correct for such inaccuracies in the prior is to interpolate between the implied distribution $P(b)$ and a simple but explicit distribution over b, such as a uniform distribution. However, given a procedure for sampling only from $P(a)$, sampling a uniform distribution over b would itself naïvely require an intractable marginalization over a.

In this paper, we develop a method to shift this implied distribution over b toward uniform without having to explicitly marginalize over the remaining variables in the model, using a low-dimensional approximation to the perturb-max sampling procedure [Papandreou and Yuille, 2011]. While the exact sampling procedure places independent perturbations on joint configurations of variables, the

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

approximation factors these perturbations into a sum of perturbations on each marginal distribution. We modify this factorization with a weighted average that allows us to *interpolate* between distributions defined over disjoint subsets of variables in the model.

2 Perturb-max interpolation

2.1 The Gumbel-max trick

A Gumbel-distributed random variable γ with location parameter m has a CDF defined as

$$P(\gamma \leq g) = \exp\{-\exp\{-g + m\}\}.$$

The perturb-max method for sampling from categorical distributions (also known as the "Gumbel-Max trick") uses random perturbations over a distribution to turn a sampling problem into an optimization problem. Specifically, given a distribution $P(x), x \in [N]$, such that $P(x)$ follows a Gibbs distribution $P(x) \propto \exp\{\theta(x)\}$, perturb-max returns an exact sample from $P(x)$ by adding independent Gumbel-distributed noise to each element of $\theta(x)$ and finding the argmax over the perturbed distribution.

Formally, for $\gamma(x)$ i.i.d. Gumbel(0),

$$\max_x \{\theta(x) + \gamma(x)\} \sim \text{Gumbel}\left(\log \sum_x \exp\{\theta(x)\}\right),$$

$$\arg\max_x \{\theta(x) + \gamma(x)\} \sim \frac{\exp\{\theta(x)\}}{\sum_x \exp\{\theta(x)\}}.$$

2.2 Approximate averaging

We now illustrate our modifications to the perturb-max procedure via a simplified example. We wish to sample from a distribution over two random variables $A \in \{1, 2, 3, 4\}$ and $B \in \{0, 1, 2, 3, 4\}$, with the joint parameters

$$\theta(a, b) = \begin{cases} 1 & a \leq b \\ 1 & b = 0 \\ 0 & \text{otherwise.} \end{cases}$$

These parameters give 14 configurations with nonzero probability. To generate one sample uniformly from the joint, we can take

$$\arg\max_{(a,b)} \{\log \theta(a, b) + \gamma(a, b)\}.$$

This requires instantiating 14 independent Gumbel variables (assuming we apply noise only over nonzero configurations). We could reduce the number of independent Gumbel variables required with an approximation, adding noise separately to a and b and taking the sum:

$$\arg\max_{(a,b)} \{\log \theta(a, b) + \gamma(a) + \gamma(b)\}.$$

Such low-dimensional perturbations have been studied, for example, by Hazan and Jaakkola [2012] in the context of approximating the log-partition function in graphical models. In this case, the approximation requires instantiating only 9 independent Gumbel variables.

Our original goal was to develop a method to *interpolate* between two distributions. To incorporate an interpolation into the simple example, we turn the additive noise into a weighted average:

$$\arg\max_{(a,b)} \{\log \theta(a, b) + (1 - \alpha)\gamma(a) + \alpha\gamma(b)\}. \tag{1}$$

Figure 1 gives an empirical intuition for this weighted average by varying the value of α in the example, and plotting the mass placed on $b = 0$ in the resulting sample. When $\alpha = 0$, Equation 1 corresponds to sampling from the marginal distribution over a; when $\alpha = 1$, Equation 1 corresponds to sampling from the marginal over b. The former gives a marginal probability $P(b = 0) \approx 0.32$ while the latter gives $P(b = 0) = 0.2$. Figure 1 demonstrates that varying α interpolates smoothly between these two distributions.

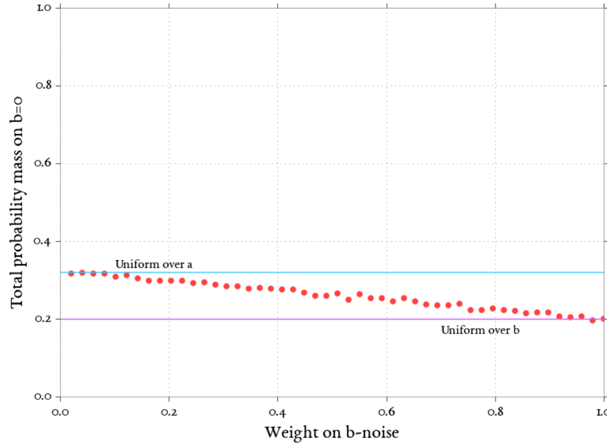

Figure 1: Behavior of shared-noise approximation as α varies.

3 Approximating a posterior over programs

Having described our sampling procedure in the abstract, we demonstrate its application to a structured modeling paradigm: programming languages.

3.1 Program formalism

In this section we formalize the notion of a program in our sampling algorithm.

Each program is a model that maps inputs $\mathcal{X} \subseteq [N]^d$ to labels $\mathcal{Y} \subseteq [N]^k$.

A program $f(x) : \mathcal{X} \to \mathcal{Y}$ is defined by its *syntax*, or its representation as a sequence of instructions, and its *semantics*, or the complete set of labels for all inputs:

$$h_f(\mathcal{X}) := (f(x_1), f(x_2), \ldots, f(x_{|\mathcal{X}|})).$$

(An instruction is a program of length one.) We denote the set of all possible syntactic representations in our language as $\mathcal{F}$, and the set of all possible semantic configurations as $\mathcal{H}$.

The mapping from syntax to semantics is many-to-one: there are multiple syntactic representations for the same semantics. Each semantics $h \in \mathcal{H}$ defines an *equivalence class* over syntactic representations:

$$C_h(\mathcal{X}) := \{f \in \mathcal{F} : h_f(\mathcal{X}) = h\}.$$

We denote the set of all equivalence classes by $\mathcal{C} := \{C_h : h \in \mathcal{H}\}$. Note that $|\mathcal{C}| = |\mathcal{H}|$, but $\cup_{h \in \mathcal{H}} |C_h| = |\mathcal{F}|$.

We can view the program syntax as a flexible model parameterization. In our framework, programs are deterministic, but the framework can be extended to probabilistic programs.

3.2 A simple prior

The end-user defining the program synthesis task has a prior over programs that is difficult to specify, but essential for making search over programs efficient. Existing synthesis systems define this prior in various ways, from a simple length prior that penalizes longer programs [Ellis et al., 2016] to structured priors with learned parameters [Liang et al., 2010, Menon et al., 2013]. Defining a prior over syntax can be relatively straightforward and intuitive, especially to prioritize simple and interpretable programs over complex programs with the same semantics. For example, a prior that penalizes longer programs can be defined as

$$\theta(f) = -|f|.$$

Algorithm 1 Basic enumerative sampling for programs.

Input: Labeled examples $(X, Y) = ((x_1, y_1), \dots, (x_k, y_k))$
Input: Number of samples required n
  $(m_1, \dots, m_n) \leftarrow -\infty$ ▷ Track maximum perturbed value
  $(f_1, \dots, f_n) \leftarrow \text{null}$ ▷ Track argmax
  for each $f \in \mathcal{F}$ **do**
    for each $i \in \{1, \dots, n\}$ **do**
      Draw noise $\gamma(f_i) \sim \text{Gumbel}(0, 1)$
      if $\gamma(f_i) + \theta(f, (X, Y)) > m_i$ **then**
        $m_i \leftarrow \gamma(f_i) + \theta(f, (X, Y))$
        $f_i \leftarrow f$
Output: samples $f_1, \dots, f_n$

Sampling from this prior is straightforward using the perturb-max method when enumerating programs, as described in Algorithm 1. Taking

$$\arg\max_f \{\theta(f) P(f \mid (X, Y)) + \gamma(f)\} \sim \frac{\exp\{\theta(f)\}}{\sum_f \exp\{\theta(f)\}} \tag{2}$$

produces a single sample from $\theta(f)$. We can take the argmax over the perturbed values without storing all of the values by maintaining a running maximum while streaming over the programs. Storing n such values independently, with independent noise values, generates n samples.

3.3 Accounting for semantics

Unfortunately, a prior defined purely over syntax provides little control over the induced distribution over semantics. Models expressed in programming languages are typically not identifiable: for instance, a natural programming language might have many ways of representing the zero program, so that these programs are overrepresented in the posterior.

As a simple way to control the impact of the syntactic prior, we introduce a scheme to interpolate between an arbitrary distribution over syntax and a uniform distribution over semantics.

Assume the distribution over syntax is specified in terms of unnormalized weights, $\theta(f)$. Normalizing, we have

$$P_f(f) = \frac{\exp\{\theta(f)\}}{\sum_{f' \in \mathcal{F}} \exp\{\theta(f')\}}. \tag{3}$$

Under a uniform distribution over semantics, the probability of each syntactic representation is dependent on the size of the equivalence class it participates in:

$$P_c(f) = \frac{1}{|C_h| \times |\mathcal{C}|}. \tag{4}$$

However, we cannot directly sample from this distribution because $|\mathcal{C}|$ and $|C_h|$ are expensive to compute.[1]

Intuitively, when the distribution is closer to Equation 3, we sample according to the specified distribution over semantics; when the distribution is closer to Equation 4, we first choose an equivalence class, then sample from the distribution over semantics conditioned on the equivalence class. An interpolation between the two would allow us to reduce the skew in the distribution over equivalence classes while maintaining some information provided by the syntactic prior.

Evaluating these probabilities requires computing *three* unknown normalizing constants. We would like to incorporate the interpolation into our sampling procedure without explicitly computing these normalizing terms. To accomplish this, we will rely on an approximation to the perturb-max procedure.

[1] In practice, we enumerate the entire space of programs, so we could compute the normalizing constants exactly. However, this would require two passes over the program space, which is also expensive, and storing each $|C_h|$ may require a prohibitive amount of memory if $|\mathcal{H}|$ is large.

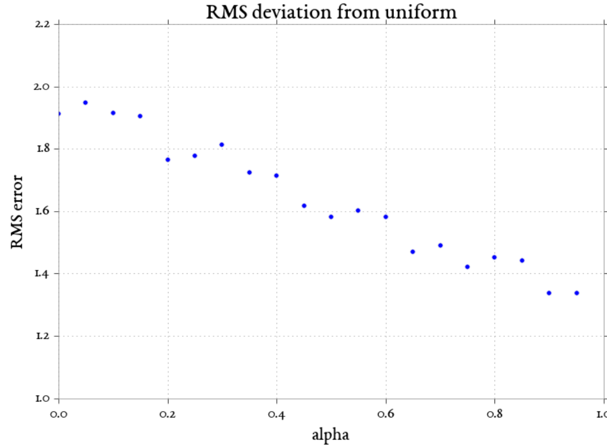

Figure 2: RMS deviation of empirical semantics distribution from uniform

3.4 Extending to programs

To extend the weighted-average interpolation scheme to our program setting, we modify Equation 2 as follows:

$$\arg\max_f \{(1-\alpha)(\theta(f)P(f|X,Y)+\gamma(f))+\alpha\sum_{i=1}^{k}\gamma(y_i)\}.$$

We have added a second noise term to the perturbation, which corresponds to adding an independent perturbation to the outputs for each example. In order to place uniform probability over equivalence classes, the noise is *shared* across identical output values. Naively implementing this sharing might require caching independent noise values for every unique equivalence class, which could become prohibitively expensive. Instead, we sample a subset of inputs and perturb the output values, sharing noise across shared output values, and approximating independent noise by a sum over the noise on each input.[2]

4 Implementation

4.1 Programming language

We use the same domain-specific language (DSL) as Balog et al. [2016], a SQL-inspired language that defines array manipulation primitives. Specifically, the DSL provides first-order functions including `max`, `min`, `head`, `last`, `sum`, `take`, `drop`, `access`, `reverse`, and `sort`, higher-order functions including `map`, `filter`, `count`, `zipwith`, and `scanl`, and basic arithmetic operators and boolean predicates. We refer the reader to [Balog et al., 2016] for the precise semantics of the language for lack of space. In total, the language contains 63 instructions which take integer or array-valued arguments and output integer or array values.

4.2 Enumeration and sampling

We enumerate programs with array inputs to depth 3 in a depth-first fashion. For efficiency, the enumerator prunes programs that are not type-correct, but does not perform additional pruning of the search space.

In Figure 2, we sample 500 programs for varying values of α weight, and measure the distance of the resulting distributions over semantics from the uniform distribution. As with the toy example,

[2]To sample from a purely uniform distribution over semantics, one could use a min-wise independent hash family [Indyk, 2001] to hash semantics to noise values, thereby both avoiding the summed approximation and sampling in a space-efficient way.

as weight is moved from syntax to semantics, the distribution over semantics becomes closer to uniform.

Acknowledgements

Thanks to Moses Charikar, Stefano Ermon, and Percy Liang for helpful discussions, and to Paroma Varma and Bryan He for feedback that greatly improved the presentation. PT is supported by an NDSEG fellowship.

References

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Sampling for bayesian program learning. In *Advances In Neural Information Processing Systems*, pages 1289–1297, 2016.

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.

Andy Gordon, Thomas A. Henzinger, Aditya Nori, and Sriram Rajamani. Probabilistic programming. IEEE, May 2014. URL `https://www.microsoft.com/en-us/research/publication/probabilistic-programming/`.

Tamir Hazan and Tommi Jaakkola. On the partition function and random maximum a-posteriori perturbations. *arXiv preprint arXiv:1206.6410*, 2012.

Piotr Indyk. A small approximately min-wise independent family of hash functions. *Journal of Algorithms*, 38(1):84–90, 2001.

Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 639–646, 2010.

Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 187–195, 2013.

George Papandreou and Alan L Yuille. Perturb-and-map random fields: Using discrete optimization to learn and sample from energy models. In *2011 International Conference on Computer Vision*, pages 193–200. IEEE, 2011.

Tong Wang, Cynthia Rudin, Finale Doshi, Yimin Liu, Erica Klampfl, and Perry MacNeille. Bayesian or's of and's for interpretable classification with application to context aware recommender systems. In *International Conference on Data Mining (ICDM)*, 2016.