Probabilistic Programming

Ben Ayad, Mohamed Ayoub

SCCS
Mohamed 6 Polytechnic University
Benguerir, Morocco

Contents

Project context

Deep Learning has integrated production level systems, and currently, the list of its applications also includes some critical ones, e.g., self-driving cars. The down side of this use is the uncertainty of deep neural networks behavior and the lack of its interpretability, which lately raised a lot of concerns in the computer science community. Therefore, the verification of Deep Learning based models remains a widely open and a very active research topic.

For this module, we had the chance to discover a wide range of topics that couple both worlds of formal methods and deep learning, more specifically, we were introduced to a new programming paradigm, called Probabilistic Programming.

This report is structured in the following manner, it includes 2 main parts. At first, we'll define Probabilistic Programming, preview its related tools and cite some examples of probabilistic programs. Finally, we'll discuss the different efforts that has been made to verify this type of programs.

Probabilistic programming

Definition

Probabilistic programs (PPs) [9] are usual functional or imperative programs with two additional constructs:

1. The ability to draw values at random from distributions
2. The ability to constrain (condition) values of variables via observations

The key idea of PPs is to view programs as **distributions over executions**. Semantically speaking, a PPL will be a map from programs to distributions. When the program halts with probability one, this induces a proper distribution over return values [5]. On the other hand, and using Church [7], it was shown that any any computable distribution can be represented as the distribution induced by a Church program [16].

Probabilistic programming Languages, The way to Rapid Bayesian Inference:

In fact, the main feature behind the development of Probabilistic Programming Languages (PPLs) is to enable rapid Bayesian inference. Following this philosophy, probabilistic programming could be seen as a way to write your probabilistic model while abstracting how Bayesian inference is done, the PPL deals with it in the background. Analogously speaking, when deep learning started, every programmer had to code the back-propagation by himself, until some libraries introduced auto-differentiation, which takes care of all the hard maths (optimization side), and the programmer only focuses on the model design (forward propagation), this played a huge role in the "democratization" of ML, nowadays, you can just specify your layers one by one and call the "fit" method without even thinking of the back-propagation. In this sense, PPL has this objective, to automatize the Bayesian inference, which is the hardest part in dealing with probabilistic programs, **by hiding the details of inference inside the compiler and runtime**.

Let's end this section with a critical quote from N. D. Goodman, one of the creators of WebPPL [8]:

> The critical obstacle to probabilistic programming as a practical tool is efficient implementation of the **inference operator**.

Goals of Probabilistic Programming:

Many consider PP as a futuristic tool for programmers with limited machine learning expertise, in a sense that it will allow fast prototyping of probabilistic models without worrying much about the inference of models, which is ,usually, the most difficult part when dealing with probabilistic modeling problem.

Inference algorithms should be implemented as a feature within the probabilistic language, and should be run in the background. Moreover, this will allow also the ability to separate the model development from the inference.

The following is a non-exhaustive list of what we could achieve with proper implementation of PPLs:

- Increase programmer productivity
- Not worrying about the inference methods
- Define new kinds of complex models

Simple Examples:

The next two examples follow the syntax of conditional probabilistic guarded command language (cpGCL), which the one used in J.P Katoen work, its syntax is expressed as follows:

$\Rightarrow$ skip empty

$\Rightarrow$ abort

$\Rightarrow$ x := E

$\Rightarrow$ observe (G)

$\Rightarrow$ prog1 ; prog2

$\Rightarrow$ if (G) prog1 else prog2

$\Rightarrow$ prog1 [p] prog2

$\Rightarrow$ while (G) prog

Listing 1: Geometric Distribution in cpGCL

```
bool c = true ;
int i = 0 ;
while(c) {
    i++ ;
    (c = false [p] c = true)
}
```

Listing 2: Variant of a Geometric Distribution in cpGCL

```
bool c = true ;
int i = 0 ;
while(c) {
    i++ ;
    (c = false [p] c = true)
}
observe(odd(i))
```

The first program obviously models a geometric distribution with a parameter p, such that: P[i = N] = p(1-p)$^{N-1}$ $\quad \forall N > 0$

The second example adds an **observe statement** at the end of the program. This one line messes up with the whole output of the program, now the runs that happen to have i as an even number are blocked. Not only that they are blocked, also, the probabilities are re-scaled by the probability of feasible runs. In this case:

The probability of the feasible runs are the ones when i is an odd, their sum is equal to: 1/(2-p)

And hence, the probability of each possible run is rescaled by it (meaning of the first factor below): P[i = 2N + 1] = (2-p)p(1-p)^{2N} $\quad \forall N \geq 0$

Applications and tools

Probabilistic programming is widely used in different domains, including:

- Quantum computing
- Randomized algorithms
- Bayesian Networks
- Approximate computing

Many of its applications have already seen light in real life situations, for instance, **TrueSkill** [10] is already implemented in **XBOX** to match players with similar skills. It actually tracks the uncertainty about player skills, explicitly draws models and can deal with any number of competing entities and eventually infer individual skills from team results. The inference is performed by approximate message passing on a factor graph representation of the model.

Many tools are currently available, mainly concerning probabilistic graphical models, **TrueSkill** is for instance is using **Infer.Net** [14], the list of tools also includes:

- For Graphical models
  - BUGS [6]
  - STAN [18]
- For Factor graphs
  - Factorie
  - Infer.Net [14]

Probabilistic Programming & Machine Learning

In this section, we'll detail two examples of the use of probabilistic programs in Machine Learning. The first one is concerning learning the parameters of a Bayesian Network and the second example addresses their use in Bayesian Neural Networks.

Bayesian Networks

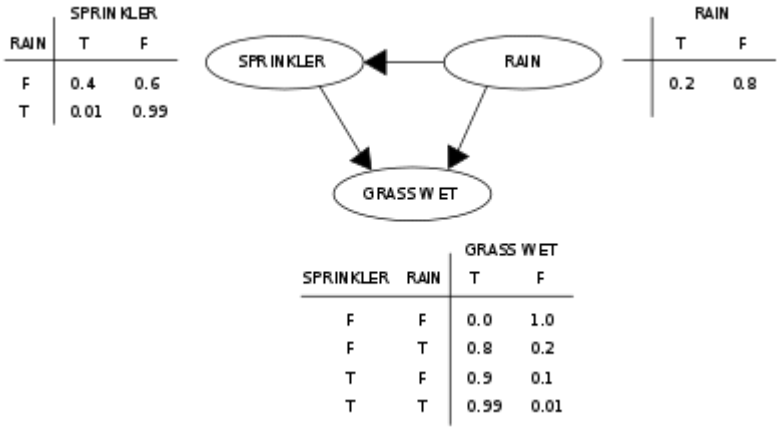
Definition: A Bayesian network is a directed acyclic graph in which:

- Each edge corresponds to a conditional dependency.
- Each node corresponds to a unique random variable.

Formally, if an edge (A, B) exists in the graph connecting random variables A and B, it means that P(B/A) is a factor in the joint probability distribution, so we must know P(B/A) for all values of B and A in order to conduct inference (which is usually summed up in a table).

Why PPs suits Bayesian networks perfectly: The main feature in Bayesian networks is the conditioning between random variables (described graphically as edges). This conditioning between random variables could naturally be implemented in a PPL using the second construct of probabilistic programs which is the observe statement, the whole idea of conditioning between random variables could be transmitted in the probabilistic programming world in one single line.

Example: For instance, let's take at the simple Bayesian Network presented below and see its implementation using a cpGCL:



Its implementation in cpGCL would simply look like the following, where every table is mapped to probabilistic program, for instance the last table is mapped to the following PP:

Listing 3: Bayesian Net As a Probabilistic Program

```
if (S=0 && R=0) {
    GW = 0 [0] GW = 1
} else if (S=1 && R=1) {
    GW = 0 [0.99] GW = 1
} else if (S=0 && R=1) {
    GW = 0 [0.8] GW = 1
} else if (S=1 && R=0) {
    GW = 0 [0.9] GW = 1
}
```

The interest of the observe statement would come up if we were asking for a conditional probability, like P(GW / S=1), then we would easily put **observe(S=1)** before the code stated above.

Important result: Generally speaking every Bayesian network could be implemented using a loop-free probabilistic program.

Bayesian Neural Networks

Hypothetical set up: Training a network that classifies images of road signs, and testing it with an image that doesn't make sense to the classifier (e.g., Using a picture of a human, or any class of objects that was not a part of the training set).

Problematic: As ironically as it sounds, the model will eventually choose the road sign with highest probability (suppose we have a softmax layer at the end). I.e., the model can't express that he's not sure of its prediction, or even express that he has never seen this example before. And When we train a model the classical way, we will always have this problem, the model always gives an answer without any additional details on the confidence of its choices. It behaves the same way either the decision was the right or completely wrong.

Potential solution: Bayesian ML could solve this problem, its main interest is that we can visualize how much the model is sure of each prediction, by initially "doubting" the parameters. In other words, parameters that we learn in a Bayesian set up are distributions and NOT values.

Below is an example of a Bayesian NN:

Listing 4: Training of a Bayesian NN:

```
from edward.models import Normal

def neural_network(X):
  h = tf.tanh(tf.matmul(X, W_0) + b_0)
  h = tf.tanh(tf.matmul(h, W_1) + b_1)
  h = tf.matmul(h, W_2) + b_2
  return tf.reshape(h, [-1])


# DATA
X_train, y_train = build_toy_dataset(FLAGS.N)

# MODEL
with tf.name_scope("model"):
  W_0 = Normal(loc=tf.zeros([FLAGS.D, 10]), scale=tf.ones([FLAGS.D, 10]),
               name="W_0")
  W_1 = Normal(loc=tf.zeros([10, 10]), scale=tf.ones([10, 10]), name="W_1")
  W_2 = Normal(loc=tf.zeros([10, 1]), scale=tf.ones([10, 1]), name="W_2")
  b_0 = Normal(loc=tf.zeros(10), scale=tf.ones(10), name="b_0")
  b_1 = Normal(loc=tf.zeros(10), scale=tf.ones(10), name="b_1")
  b_2 = Normal(loc=tf.zeros(1), scale=tf.ones(1), name="b_2")

  X = tf.placeholder(tf.float32, [FLAGS.N, FLAGS.D], name="X")
  y = Normal(loc=neural_network(X), scale=0.1 * tf.ones(FLAGS.N), name="y")
```

We can clearly see here that each weight **w** and each bias **b** is a normal distribution, this was only the forward propagation (or model definition), appropriate inference method should be used next to learn the parameters.

The following code snippet sums up the inference side (variational inference [11] is used in this case), the programmer has to only define the inference method, and everything is tracked automatically by Edward. (technically the coder only has to specify the weights before as placeholders so that the optimizer can track them, similar to PyTorch)

Listing 5: Inference of a Bayesian NN:

```
# INFERENCE

inference = ed.KLqp({W_0: qW_0, b_0: qb_0,
                     W_1: qW_1, b_1: qb_1,
                     W_2: qW_2, b_2: qb_2}, data={X: X_train, y: y_train})

inference.run(logdir='log')
```

Verification of probabilistic programs

Probabilistic Programs are typically small, and help abstract and simplify complex models in clearer forms. In contrast, they are harder to understand and analyse. For instance, the elementary question of almost-sure termination is as hard as the universal halting problem. Moreover, bugs could occur easily. Hence the need to develop program analysis techniques, based on static program analysis, and model checking to make probabilistic programming more reliable. Listed bellow is a list of some papers that captured our attention during this bibliographic study:

Bayesian inference using data flow analysis [2]:

The proposed technique performs probabilistic inference using data flow analysis, using ADDs as a data structure. They also showed that the method indeed computes the posterior probability of a probabilistic program. The "data flow facts" are probability distributions, and the analysis merges data flow facts at join points, and computes fixpoints in the presence of loops. Moreover, they presented some heuristics to scale the inference.

Weakest precondition reasoning for expected run times of probabilistic programs [12]:

This paper has studied a wp–style calculus for reasoning about the expected run-time and positive almost-sure termination of probabilistic programs. They could also determine the expected termination time of a probabilistic program (when it's possible) and also prove positive almost-sure termination.They provided several sound and complete proof rules for obtaining upper as well as lower bounds on the expected run–time of loops.

Reasoning about recursive probabilistic programs [15]:

This paper presents a wp–style calculus for obtaining expectations on the outcomes of recursive probabilistic programs. They provided several proof rules to derive one and two–sided bounds for such expectations, and show the sound-

ness of the wp–calculus with respect to a probabilistic pushdown automaton semantics. They also give a wp–style calculus for obtaining bounds on the expected runtime of recursive programs that can be used to determine the time until termination of such programs.

Probabilistic program analysis with martingales [1]:

In this paper, the authors extended the quantitative invariants approach of McIver and Morgan [13] to a wider class of probabilistic programs through martingale and super-martingale program expressions using concentration of measure inequalities (Azuma-Hoeffding theorem) to generate probabilistic assertions. They also defined super martingale ranking functions (SMRFs) to prove almost sure termination of probabilistic programs. And finally presented constraint-based techniques to generate (super) martingale expressions.

Inferring whole program properties from finitely many paths [17]:

In this paper, the authors presented a static analysis approach that provides guaranteed interval bounds on the values (assertion probabilities) of the correctness properties of PPs (named queries). First, they conclude facts about the behavior of the entire program by choosing a finite, adequate set of its paths. The queries are then evaluated over each path by a combination of symbolic execution and probabilistic volume bound computations. Each path yields interval bounds that can be summed up with a "coverage" bound to yield an interval that encloses the probability of assertion for the program as a whole.

Probabilistic abstract interpretation [3]:

They presented in this paper a new probabilistic abstraction framework that allows to systematically lift any classical analysis or verification method to the probabilistic setting by separating in the program semantics the probabilistic behavior from the (non-)deterministic behavior. The separation provides new insights for designing novel probabilistic static analyses and verification methods. They also defined the concrete probabilistic semantics and propose different ways to abstract them.

PSI: Exact symbolic inference for probabilistic programs [4]:

This paper presents a novel symbolic analysis system for exact inference in probabilistic programs with both continuous and discrete random variables. PSI computes succinct symbolic representations of the joint posterior distribution represented by a given probabilistic program. PSI could also compute answers to various posterior distribution, expectation and assertion queries using its own back-end for symbolic reasoning. The evaluation shows that PSI is more effective than the other existing exact inference approaches.

Bibliography

[1] Aleksandar Chakarov and Sriram Sankaranarayanan. Probabilistic program analysis with martingales. In *International Conference on Computer Aided Verification*, pages 511–526. Springer, 2013.

[2] Guillaume Claret, Sriram K Rajamani, Aditya V Nori, Andrew D Gordon, and Johannes Borgström. Bayesian inference using data flow analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 92–102, 2013.

[3] Patrick Cousot and Michael Monerau. Probabilistic abstract interpretation. In *European Symposium on Programming*, pages 169–193. Springer, 2012.

[4] Timon Gehr, Sasa Misailovic, and Martin Vechev. Psi: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*, pages 62–83. Springer, 2016.

[5] Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459, 2015.

[6] W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex bayesian modelling. *Journal of the Royal Statistical Society. Series D (The Statistician)*, 43(1):169–177, 1994.

[7] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

[8] Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2020-9-13.

[9] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. *Future of Software Engineering, FOSE 2014 - Proceedings*, pages 167–181, 2014.

[10] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill(tm): A bayesian skill rating system. In *Advances in Neural Information Processing Systems 20*, pages 569–576. MIT Press, January 2007.

[11] Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. Stochastic variational inference. *The Journal of Machine Learning Research*, 14(1):1303–1347, 2013.

[12] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected run–times of probabilistic programs. In *European Symposium on Programming*, pages 364–389. Springer, 2016.

[13] Annabelle McIver, Carroll Morgan, and Charles Carroll Morgan. *Abstraction, refinement and proof for probabilistic systems*. Springer Science & Business Media, 2005.

[14] T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. /Infer.NET 0.3, 2018. Microsoft Research Cambridge. http://dotnet.github.io/infer.

[15] Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. Reasoning about recursive probabilistic programs. In *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–10. IEEE, 2016.

[16] Daniel Murphy Roy. *Computability, inference and modeling in probabilistic programming*. PhD thesis, Massachusetts Institute of Technology, 2011.

[17] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 447–458, 2013.

[18] Stan Development Team. The Stan Core Library, 2018. Version 2.18.0.