A Survey on Probabilistic Programming in the Context of Neural Networks

Xingyu Li, Zekun Zhao

Department of Computer Science, BSOE,
University of California, Santa Cruz

{*xli279, zzhao99*}*@ucsc.edu*

Abstract

This report concerns the cooperation of Probabilistic Programming with Deep Neural Network, which is an emerging application of Probabilistic Programming. First, the basic ideas and methods of these two fields are briefly reviewed. Then, we turn to the general discussion of the cooperation, including its benefits, ways of implementation and the requirements thus posted on the Probabilistic Programming side. Finally, we show more details about two popular schemes of implementing the cooperation as well as describing and comparing existing Probabilistic Programming frameworks that are able to implement such cooperation.

1 Introduction

As an emerging area in the field of Programming Language, Probabilistic Programming[1], gains increasing popularity recently and lots of related frameworks have been developed, such as Stan [2, 3], pyro [4] and Edward [5, 6] just to name a few. In Probabilistic Programming, one adopts a distinct view towards the probabilistic components in a program, treating the probabilistic distribution as a basic building block and providing a concise syntax to define generative models and to do inference. In this way, Probabilistic programming systems abstract away many difficulties in implementing sampling and inference algorithms and enables people to talk about statistic modeling more intuitively.

Also, we observe that the celebrated Deep Neural Networks models are highly related to the probabilistic inference. For example, in Variational Auto-Encoder (VAE) [7] model, the neural network learns the probabilistic distributions of each categories and the objective is practically the K-L divergence between the learned posterior and the true one. Hence, it would be beneficial to use Probabilistic Programming Language in describing neural network models.

In this project we surveyed Probabilistic Programming in the context of Deep Neural Networks. We found these two highly related fields do not share an inclusive relation, rather they are complementary to each other. We definitely realized that this short report can never serve as a comprehensive survey on this two broad field. Actually a quarter is far from enough for us to go through even just a part of related materials. We then decided to focus more on the high level ideas and principles, aiming to illustrate the underlying picture of how Probabilistic Programming and the Neural Network models can help to augment each other.

This report is organized as follow. In the next two sections, we briefly review the present state of Probabilistic Programming and Deep Neural Networks, reviewing their focus, pros and cons. Then, section 4 is devoted to the topics of why this two field can help each other and how can one combines the two. We also include a simple discussion on the requirements posted on the design of syntax when the Probabilistic Programming frame work is adapted to describe the Deep Neural Network models. In section 5, we present the two most popular and promising schemes of combining Probabilistic Programming with Deep Neural Networks, and

use examples to illustrated how they work. Finally, we reflect and conclude all the knowledge we learned during this project.

2 Probabilistic Programming in a nutshell [1]

Bayesian inference lies at the heart of probabilistic modeling, and Probabilistic Programming provides a framework that hides the complexity of doing Bayesian inference. There are several aspects of this framework, including the design of syntax, evaluator (the sampling and inference algorithms) and interpreter or compiler. In this report, we only investigate the first two.

From a user's viewpoint, the magic of Probabilistic Programming owns directly to the combination of concise syntax and sophisticated evaluator, which makes the tasks of complex sampling and inference as simple as pressing a button. Note that, in order to make Probabilistic Programming practical, it is naturally to require the sampling and inference algorithms to be general and applicable to any possible case in practice.

2.1 Syntax

Instead of building the whole framework from scratch, most existing Probabilistic Programming Languages extend other sophisticated Programming Languages [], such as C and Python, by adding features to the original syntax.

There are two key features that must be included in the definition of the syntax for a Probabilistic Programming Language, namely the `sample` and `infer`. The `sample` takes a distribution object as input and returns a sampled value; The `infer` accepts two input arguments, one is a distribution object, and the other is a (or an array of) observed values. It evaluates the posterior distribution given those observed values. For example (see ref [1] for details)

```
(let [data [1.1 2.1 2.0 1.9 0.0 -0.1 -0.05]
      likes (foreach 3 []
              (let [mu (sample (normal 0.0 10.0))
                    sigma (sample (gamma 1.0 1.0))]
                (normal mu sigma)))
      pi (sample (dirichlet [1.0 1.0 1.0]))
      z-prior (discrete pi)]
  (foreach 7 [y data]
    (let [z (sample z-prior)]
      (observe (get likes z) y)
      z)))
```

Figure 1: An example of a Gaussian Mix Model in a First Order Probabilistic Programming Language.

In above example, we see that one can easily sample from any distribution using an intuitive syntax, `sample (dist params)`, where `(dist params)` defines the target distribution with parameters. For the inference part, `observe` does the same job as the `infer`, and the syntax is `observe (dist params) vals`, where `vals` indicates the observed values that we are inferring on.

2.2 Sampling and Inference Algorithms

In this subsection, we briefly describe the basic ideas of some widely used sampling and inference algorithms in Probabilistic Programming. One can see that the sampling and inference algorithms are intrinsically interconnected. Actually, to do inference, one needs to estimate the present distribution somehow, which in turn requires some sorts of sampling algorithms or their equivalent.

The popular inference algorithms mainly fall into the following two categories:

- **Sample-based Inference** The sample-based methods exploit the empirical estimate of expectations, i.e.

$$\mathbb{E}[f(x)] = \int f(x)P(x)\,dx \approx \frac{1}{n}\sum_{i}^{n} f(x_i)$$

  where f is the target function and $\{x_i\}$ are samples from distribution $P(x)$. The naive sampling methods (e.g. reject sampling) are very inefficient. One main alternative to them is the Markov Chain Monte Carlo (MCMC) sampling methods, among which the Metropolis Hastings sampling, Gibbs sampling and Hamiltonian Monte Carlo sampling are well-known examples. In MCMC, a Markov chain is used to generate the next sample based on the current one, making it more likely to stay in densely packed probability regions. Even though it can provide accurate estimation, the MCMC methods are often too slow for problems with rich structure or internet-scale data.

- **Variational Inference** The Variational methods are usually much faster than the MCMC methods, nonetheless they provide only approximate estimates. The idea is that instead of the whole function space, one choose a smaller family of functions (usually paremeterized in a way amenable to efficient optimization procedures), and approximate the true posterior within this function family. One key aspect of variational inference methods is how to measure the "distance" between the approximate posterior and the true one. In practice, one usually adopts a distance of f-divergence form, e.g. K-L divergence.

In practice, those sampling and inference algorithms are sealed by carefully designed syntax, such as the one shown in the example of the previous section. Once you use `infer` on target distribution, the program will automatically call the build-in inference algorithms for you. This kind of auto-inference technique plays exactly the same role as the auto-differentiation in the Deep Neural Network frameworks.

3 Deep Neural Network in a nutshell

In Fig. 2 below, we illustrate the typical structure of a Deep Neural Network model. Generally, they consist of three components: the training data, which is samples from the target distribution; a predefined network architecture (we use fully-connected layer in the figure, but it can be way more complex); and the Loss function, which is the optimization objective.

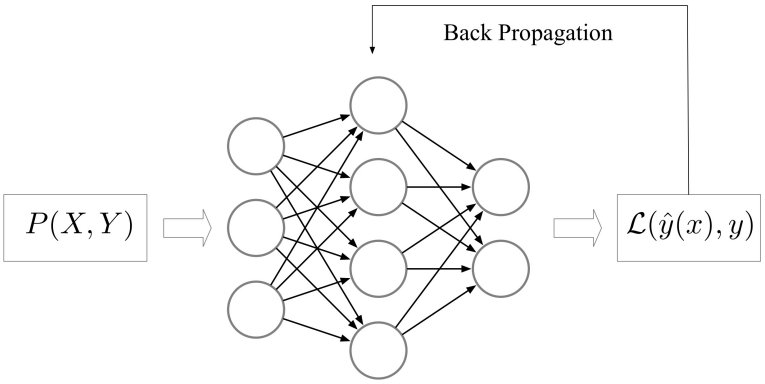


Figure 2: Illustration of a typical Deep Neural Network model.

The power of Deep Neural Network models comes from their ability of learning (or training). There are a bunch of learnable parameters in the network, and their size could be even much bigger than the training data size. This is the base of the Neural Network being able to adapt to the training dataset. During the training process, the data is fed into the network and the loss function is calculated. Then, the tuning of those parameters is guided by the optimization of the loss function. Through back propagation, the amount of update for each parameter is

passed back to the network, and thus complete the circle of one training step. In practice, the training step is executed repeatedly until the loss converges.

Though, at first look, the Deep Neural Network is quite different from the probabilistic model, we can in fact observe many similarities. The Perceptron model in Deep Neural Network turns out to be equivalent to the Logistic Regression model in statistic, and minimizing the mean squared error loss function is nothing but maximizing the likelihood of observing the training data. For more complex models, the relations are not so obvious, however, in general, one may identify:

- the learning process of Deep Neural Network is a kind of inference process. That is, based on the observations (training data), back propagation helps to infer the underlying model (update parameters).
- in many cases, the target of training can be interpreted as the effort to grasp the training data's distribution $P(X, Y)$, so that one can make accurate predictions. In this sense, the network itself can be viewed as modeling a conditional distribution $P(\hat{Y}|X)$, where $\hat{Y}$ is the output of the Deep Neural Network model.

Of course, there are also striking difference between the Deep Neural Network and Probabilistic models. For example, the Gradient Descent is pretty much irrelevant to the Bayesian inference. And, in Deep Neural Network, the prediction are always a point estimate, whereas in Probabilistic models one can talk about the uncertainty of the predictions.

4 Cooperation of the Probabilistic Programming and Deep Neural Network

From the brief reviews about the Probabilistic Programming and Deep Neural Network, we saw that both of them concern about inference and probabilistic distributions. One may expect that it is possible to combine the two fields together despite their distinctions. A close look reveals such a cooperation might even bring a kind of more powerful model that possesses the advantages of both sides.

4.1 What is the benefit of the cooperation? [8]

It is worthy to list the pros and cons of Probabilistic Programming and the Deep Neural Network for further comparison. For the Probabilistic Programming side, we see

- the predefined probabilistic model guide the inference on the observed data, which results in a structured representation. In this way, the model can exploit the data more efficiently and its results are more interpretable.
- another desirable feature of probabilistic models is they explicitly model the uncertainty of predictions, which tells how sure or unsure the model is.
- on the other hand, since probabilistic models adopt rigid assumptions, they may not be able to adapt to the data set they are working on. In practice, people often need feature engineering to help improve the performance of probabilistic models.

For the Deep Neural Network side

- the biggest defect of Deep Neural Network models is they are largely black box. The lack of explainablity has limited their application in many areas, such as Medicine. Further, it usually requires very large amount of data to train the Deep Neural Network model.
- we have mentioned in previous section that the Deep Neural Network models normally only provide point estimate. Unable to capture the uncertainty of the model has limited their generalizability.

- despite those defects, the Deep Neural Network models have very high capacity, and their ability of auto feature extraction and model learning have won them the popularity in both industry and academia.

Above lists are not a complete summary, however one is able to sense a close relation between the Probabilistic Programming and Deep Neural Network: they seems to be just complementary. Their advantages and disadvantages are corresponding to each other. So that it is possible to combine them to form a new model that not only explainable but also flexible; not only can automatically extract features from data but also being able to grasp the uncertainty of the prediction.

4.2 How to implement the cooperation? [9, 10, 11, 12, 8]

The next question is how can one realize the cooperation. People have already done a lot works in this direction. From literature, we find the methods mainly fall into the following two frameworks:

1 one simply turns all the parameters (weights) of a neural network into random variables. In this way, the conventional Deep Neural Network model is transformed into a probabilistic model, on which one can do inference using existing Probabilistic Programming frameworks. This method is known as the Bayesian Neural Network (BNN). As a probabilistic model, it naturally assign uncertainty to its predictions. However, since now we are learning the distribution of each parameter rather than simply a value, it is harder to train the Bayesian Neural Network models.

2 one can also "concatenate" the two to form a new kind of models. The basic idea is to use neural network to process data, then the automatically extracted features are passed to a Probabilistic model which guides the inference. In practice, a framework of this kind has been developed, which is called the Structured Variational AutoEncoder (SVAE).

We will dive into more details about this two frameworks in section 5.

4.3 Requirements on syntax and inference algorithms

It is noticed [6] that in order to combine the Probabilistic Programming with the Deep Neural Network framework, there posts at least the following two requirements on the syntax and inference algorithms of Probabilistic Programming:

1 **compositionality** on syntax for both model define and doing inference. This requirement comes from the fact that, in practice, there is no constraint on the size and type of a specific Deep Neural Network model. The only way out is to specify the model define and inference on model "building blocks", and then gradually compose them to form more complex models.

```
qbeta = PointMass(params=tf.Variable(tf.zeros([K, D])))
qz = Categorical(logits=tf.Variable(tf.zeros([N, K])))

inference_e = ed.VariationalInference({z: qz}, data={x: x_train, beta: qbeta})
inference_m = ed.MAP({beta: qbeta}, data={x: x_train, z: qz})
...
```

Figure 3: An example from Edward. Illustration of composing inference.

Fig. 3 shows an example from ref [6], it is a part of the realization of Expectation maximization algorithm in Edward, and Edward is a Probabilistic Programming Language that support compositional representations of both random variable and inference. From the example, we see one can first infer on `z` based on observed data `x` and `beta`. Then, `z` can be treated as data to infer on `beta`.

2 the inference algorithms used must be adapted to **rapid and repeated** inference. Unlike the inference in traditional probabilistic models, Deep Neural Network models are usually applied on very large dataset. Further, one need to keep doing inference on the training set iteratively, until the model converges. Thus the efficiency is a key, and the variational inference algorithms become popular choices.

5 Application

5.1 Mechanism

5.1.1 Bayesian Neural Network [9, 12]

Compared with deep neural network, bayesian neural network can offers uncertainty estimates via its parameters in form of probability distributions instead of just single point-estimates. From a probability theory perspective, bayesian neural network are more robust to over-fitting, and can easily learn from small datasets. At the same time, by using a prior probability distribution to integrate out the parameters, the average is computed across many models during training, which gives a regularization effect to the network, thus preventing over-fitting and give us a measure of uncertainty in the prediction. But inferring model posterior in a Bayesian neural network is a difficult task in both theory and computational cost.

$$P(\theta|x) = \frac{P(x|\theta)P(\theta)}{P(x)}$$

The goal of applying Bayes theorem in neural network model is to find the probability θ for any given data x. This is exactly the result from the calculated posterior of Bayes theorem. We define $P(\theta)$ as our prior, $P(x|\theta)$ as the likelihood and tells us the data distribution, $P(x)$ is the observed result(also named as evidence).

$$P(x) = \int P(x, \theta) d\theta$$

To calculated this, we need to integrating all possible model values which causes the whole solution intractable. The current popular way of solving this is to use Variational inference to approximate the functional form. In their work [12], the author introduces the idea of applying two convolutional operations, one for the mean and one for the variance for Bayesian Convolutional Neural Network.

Generally, People design special loss function in deep neural network for different optimizing purpose, which is easy to do in a well-defined framework nowadays, like keras, tensorflow and pytorch. However, it is not intuitive that how to use that in a Bayesian Neural Network. The classical way to measure loss in a Bayesian Neural Network is to minimise the Kullback–Leibler (KL) divergence, which is intuitively a measure of similarity between two distributions, but it is not clear that how to change the formate of this measurement to match existing loss functions.

5.1.2 Structured Variational Auto-Encoder [8, 11, 10]

Structured Variational AutoEncoders (SVAE) is a popular example of Deep probabilistic models instead of integrating reasoning capabilities into a complex neural network architecture, it uses Neural Network to extract features from the observed data, and do/guide inference using probabilistic model.

The task is learning a vector-space representation z for each observed data point x (e.g., a handwritten digit in the the classic MNIST dataset). The computed representation can then be use as input of other models (e.g., a classifier). Each data point x depends on the latent representation z in a complex non-linear way, via a deep neural network: the decoder. The top half of Figure 4 shows the corresponding graphical model. The output of the decoder is a vector μ that parameterizes a Bernoulli distribution over each pixel in the image x. Each pixel is thus associated to a probability of being present in the image. The parameter θ of the decoder is global (i.e., shared by all data points).

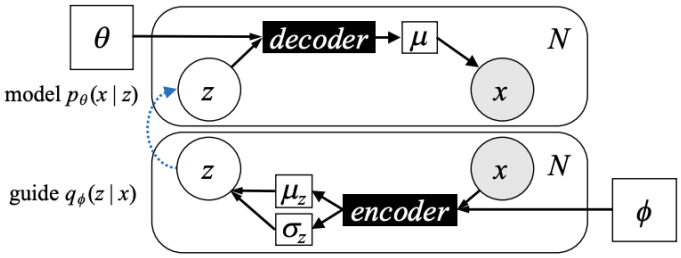


Figure 4: Graphical model of the SVAE (model and guide).

5.2 Existing Languages

This section uses examples to illustrate several popular deep probabilistic languages which enable explicit variational inference and probabilistic models that involve deep neural networks.

5.2.1 STAN and DeepSTAN [2, 3]

```
data {
  int N;
  int<lower=0,upper=1> x[N];
}
parameters {
  real<lower=0,upper=1> z;
}
model {
  z ~ Beta(1, 1);
  x ~ Bernoulli(z);
}
```

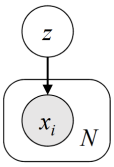


Figure 5: Learning the bias of a coin in Stan.

Stan is a popular high-level probabilistic programming language but lacks of concise, high-level, and clean ways to express deep probabilistic models. Figure 5 shows a simple Stan program from Section 2 of the Stan manual [13]. The program estimates the bias of a coin from a set of coin tosses.

The "data" block introduces observations, which are placeholders for concrete data to be provided as input to the inference procedure. The "parameters" block introduces latent random variables, which will not be provided and must be inferred. The "model" block describes the model. It places a Beta(1, 1) (i.e., uniform) prior on z, that is, initially z can take all values between 0 and 1 with equal probability.

To overcome these drawbacks and since training deep probabilistic models works best with variational inference, previous work [3] implements DeepStan which add these extensions by translating Stan programs to Pyro. Pyro embraces deep neural nets and currently focuses on variational inference. Pyro doesn't do MCMC yet. Whereas Stan models are written in the Stan language, Pyro models are just python programs with pyro.sample() statements. In DeepStan, they considered three goals to achieve : 1, Extending Stan for variational inference with high-level but explicit guides. 2,Extending Stan with a clean interface to neural networks written in Python.3,A compiler from Stan to Pyro that carefully chooses the appropriate sampling semantics. Considering the scope of topics, we will only discuss the first and second part briefly here.

To allow the user to design their own guide, DeepStan extends Stan with two new blocks: "guide parameters" and "guide". The "guide" block defines a distribution parameterized by the "guide parameters". Variational inference then optimizes the values of these parameters to approximate the true posterior.

```
networks {
  Decoder decoder;
  Encoder encoder;
}
data {
  int<lower=0, upper=1> x[28, 28];
}
parameters {
  real z[_];
}
model {
  real mu[_, _];
  z ~ Normal(0, 1);
  mu = decoder(z);
  x ~ Bernoulli(mu);
}
guide {
  real encoded[2, _] = encoder(x);
  real mu_z[_] = encoded[1];
  real sigma_z[_] = encoded[2];
  z ~ Normal(mu_z, sigma_z);
}
```

Figure 6: Variational Auto-Encoder (VAE) in DeepSTAN.

The main idea of the VAE is to use variational inference to learn the latent representation. In Figure 6, Block "networks" declares neural networks at the beginning of the pipeline. Blocks "guide" parameters and guide specify an explicit guide for variational inference after the model.

DeepStan also extends the grammar of Stan. As in Stan, a program is a succession of blocks. Objects defined in one block can be used in subsequent blocks.

```
program ::= nets? funs? data? tdata? params? tparams?
            model gparams? guide? generated?
nets    ::= networks { ndecl* }
gparams ::= guide parameters { vdecl* }
guide   ::= guide { vdecl* stmts* }
ndecl   ::= className var ;
```

Figure 7: Syntax of the language extensions.

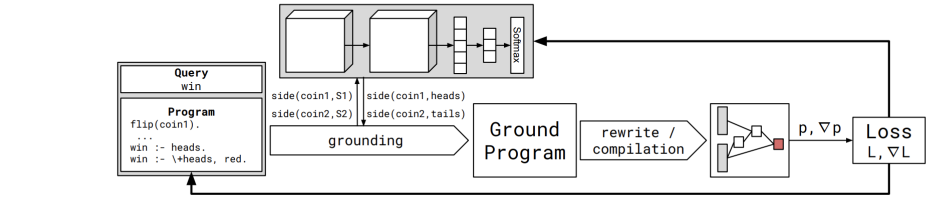
The non-terminal funs, data, tdata, params, tparams, andgquant corresponds to the original Stan blocks (respectively functions, data, transformed data, parameters, transformed parameters, and generated quantities). The only mandatory block is model. The non-terminals vdecl and stmt corresponds to variable declarations (type, name, dimensions) and statements.

5.2.2 ProbLog and DeepProbLog [11, 10]

From an existing probabilistic logic programming language, ProbLog, which can be regarded as a very expressive directed graphical modeling language. DeepProbLog extend it with the capability to process neural predicates. The idea is simple: in a probabilistic logic, atomic expressions of the form q(t1, ..., tn) (aka tuples in a relational database) have a probability p. Consequently, the output of neural network components can be encapsulated in the form of "neural" predicates as long as the output of the neural network on an atomic expression can be interpreted as a probability. The new language is called as DeepProbLog.

One important extension for DeepProbLog is called as a set of ground neural annotated disjunctions (ADs) of the form :

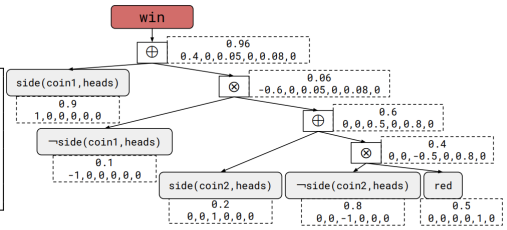
$$nn\left(m_q, \vec{t}, \vec{u}\right) :: q\left(\vec{t}, u_1\right); \ldots; q\left(\vec{t}, u_n\right) : -b_1, \ldots, b_m$$



(a) The learning pipeline.

```
flip(coin1). flip(coin2).
nn(m_side,C,[heads,tails])::side(C,heads);side(C,tails).
t(0.5)::red;t(0.5)::blue.
heads :- flip(X), side(X,heads).
win :- heads.
win :- \+heads, red.
query(win).
```

(b) The DeepProbLog program.



(c) SDD for query `win`.

Figure 8: Parameter learning in DeepProbLog.

For instance, in the MNIST addition example, we would specify the nAD :

$$nn\left(m_{\text{ digit }}, \mathbf{pic}(\mathbf{5}), [0, \ldots, 9]\right) :: \text{digit}(\mathbf{pic}(\mathbf{5}), 0); \ldots; \text{digit}(\mathbf{pic}(\mathbf{5}), 9)$$

where $m_{\text{ digit }}$ is a network that probabilistically classifies MNIST digits. $\text{digit}(\mathbf{pic}(\mathbf{5}), 0)$ is the corresponding neural predicate. DeepProbLog is a probabilistic logic programming language that incorporates deep learning by means of neural predicates. As a framework where general-purpose neural networks and expressive probabilistic-logical modeling and reasoning are integrated in a way that exploits the full expressiveness and strengths of both worlds and can be trained end-to-end based on examples. Compared with the mechanism of Bayesian Neural Network, the inference of DeepProbLog closely follows ProbLog and also use gradient descent for allowing the seamless integration with neural network training.

5.2.3 Other Languages [4, 5, 6]

Pyro is a probabilistic programming language built on Python as a platform for developing advanced probabilistic models in deep learning framework. To scale to large data sets and high-dimensional models, Pyro uses stochastic variational inference algorithms and probability distributions built on top of PyTorch. To accommodate complex or model-specific algorithmic behavior, Pyro leverages Poutine, a library of composable building blocks for modifying the behavior of probabilistic programs.

Edward is a probabilistic modeling library firstly been introduced in 2017 which builds on top of TensorFlow to support distributed training and hardware such as GPUs. Compared with Stan and other previous works, it enables the development of complex probabilistic models and their algorithms at a massive scale.

Edward2 is a distillation of Edward. It is a low-level language for specifying probabilistic models as programs and manipulating their computation. Probabilistic inference, criticism, and any other part of the scientific process use arbitrary TensorFlow ops. Their associated abstractions live in the TensorFlow ecosystem such as in TensorFlow Probability, and do not strictly require Edward2.

6 Summary

In this project we have surveyed on the Probabilistic Programming in the context of Deep Neural Networks, focusing on the cooperation of this two fields. During the survey, we have learned a lot about the key ideas, methods and sampling/inference algorithms used in Probabilistic Programming. Further, we realized that the Probabilistic Programming (the probabilistic modeling) and Deep Neural Network are related and complementary. Combining the two could a path that may leads to more powerful models in the future. Through the study of concrete examples, we also get more familiar with the existing popular Probabilistic Programming Language frameworks, such as Stan, Edward and DeepProLog. We do realize that our report is not strong enough in the "language design" aspect. This is partly due to the fact that most works concerns develop a comprehensive framework for the general Probabilistic Programming rather than focus on the specific direction of combining the Probabilistic Programming with the Deep Neural Network models. However, this direction is getting more and more popular, we believe that after this project we have prepared a good background for getting deeper into the research works in this direction.

A code description

We successfully test the Bayesian Neural Network (BNN) model and Structured Variational AutoEncoder (SVAE) model in Pyro framework. Here is the link to the code: `https://github.com/kriszhao/Pyro_BNN_SVAE/tree/6b775891dccc829ae91b7cb569d3d7483345fe84`.

We have tried to implement both above models in DeepStan and DeepProbLog. It turns out there are issues with the dependent lib (PySDD; in the Deep ProbLog case) and we find that the present version of TensorFlow conflicts with the Edward (cannot import name 'set_shapes_for_outputs'). We have tried very hard to fix the environment, however, due to the limit of time, fail to resolve the problems. In the end, we decided to read and running existing codes. We list their links in the footnote[1][2].

We believe that even though both implemented models only use the fully-connected layers (or equivalently, the Dense layers), they are sufficient to illustrate how to write down a model that combines both Probabilistic model and Deep Neural Network model using the Probabilistic Programming language.

References

[1] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An Introduction to Probabilistic Programming. sep 2018.

[2] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan : A Probabilistic Programming Language. *Journal of Statistical Software*, 76(1), 2017.

[3] Javier Burroni, Guillaume Baudart, Louis Mandel, Martin Hirzel, and Avraham Shinnar. Extending Stan for Deep Probabilistic Programming. pages 1–11, 2018.

[4] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming. *arXiv preprint arXiv:1810.09538*, 2018.

[5] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. Edward: A library for probabilistic modeling, inference, and criticism. 2016.

[6] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. Deep Probabilistic Programming. pages 1–18, 2017.

[1] `https://github.com/pyro-ppl/pyro/tree/d128c1ac51bba400918844b20cef807df1cd345e`

[2] `https://github.com/paraschopra/bayesian-neural-network-mnist/tree/249624681abce81d9a4ce52a9b293e02f7abafd5`

[7] Shengjia Zhao, Jiaming Song, and Stefano Ermon. Towards Deeper Understanding of Variational Autoencoding Models. 2017.

[8] Matthew J. Johnson, David Duvenaud, Alexander B. Wiltschko, Sandeep R. Datta, and Ryan P. Adams. Composing graphical models with neural networks for structured representations and fast inference. (Nips), 2016.

[9] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight Uncertainty in Neural Networks. 37, 2015.

[10] Robin Manhaeve, Sebastijan Dumančić, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. DeepProbLog: Neural Probabilistic Logic Programming. 2018.

[11] Hai Wang and Hoifung Poon. Deep Probabilistic Logic: A Unifying Framework for Indirect Supervision. 2018.

[12] Kumar Shridhar, Felix Laumann, and Marcus Liwicki. A Comprehensive guide to Bayesian Convolutional Neural Network with Variational Inference. pages 1–38, 2019.

[13] Stan Development Team. Stan Modeling Language Users Guide and Reference Manual, Version 2.18.0, 2018.

[14] Radford M. Neal. *Bayesian Learning for Neural Networks*, volume 118 of *Lecture Notes in Statistics*. Springer New York, New York, NY, 1996.

[15] Cam Davidson-Pilon. *Bayesian Methods for Hackers*. 2014.