Implementing Inference with Sum-Product Networks

Jhonatan S. Oliveira oliveira@cs.uregina.ca
Department of Computer Science
University of Regina
Regina, Canada

Editor: CS807 - Advanced Architecture - Dr. Gerhard and Trevor Tomesh

Abstract

Implementing Sum-product networks (SPNs) shows challenges for a resource constrain computing point of view. This is due to the size and processing requirements of these models. SPNs are a layer-wise mathematical models for learning and inference with probabilistic graphical model. Given a SPN, inference is done by propagating up and down in the network. Here, we propose one possible implementation for SPNs by using GPUs when parallelizing computation on the up and down in the networks.

Keywords: sum-product networks, resource constrain, gpu

1. Introduction

The problem of parallelizing computation has been often focused by machine learning scientist in the last few years. This is due to the increasing interest of the community and industry in the applications of deep architectures. One challenge of using these type of models is the resource constrains due to high processing power required and storage. Thus, common solutions for these problems were gathered together in open source frameworks offered by institutions of the machine learning area. Next, it is shown some of the most well known frameworks when dealing with deep models in common computers.

The first one is Theano (Theano). This is a Python library that allows a user to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. Some of the core features of this library is the integration with the famous Python library called NumPy, use of GPU for parallel computation, efficient symbolic differentiation, speed and stability optimizations, dynamic C code generation on the fly, and extensive unit-testing and self-verification. Since 2007, Theano has been powering large-scale computationally intensive scientific investigations. Although open source, Theano is primarily developed by academics by the University of Montreal.

The second one is TensorFlow (TensorFlow). TensorFlow is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.

The third one is called Torch Torch. This library is a scientific computing framework with wide support for machine learning algorithms that puts GPUs first. It is made to be easy to use and efficient, due to its easy and fast scripting language, LuaJIT, and an underlying C/CUDA implementation. In the core, Torch has a powerful N-dimensional array, lots of routines for indexing, slicing, transposing, among others; linear algebra routines, and GPU support.

2. Background

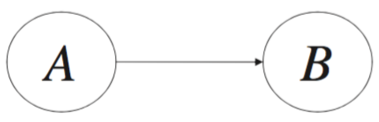
This background is drawn from the research task C, where the problem of using and implementing Sum Product networks are discussed in details.

A *join probability distribution* (JPD) can be written as a polynomial. Indeed, Darwiche (2009) proposed a novel way of representing the JPD represented by a *Bayesian network* (BN) as a simple polynomial, called a *network polynomial*. Given a JPD, it is added to each configuration (row) of it variables called indicators. Each variable in a JPD has a correspondent indicator. This type of variables can only assume values zeros or ones. They are used to mark the correspondent JPD variable as being used or not. Next, the polynomial network can be obtained by summing each row of the JPD.

Example 1 *Consider the BN in Figure 1. By definition (Koller and Friedman, 2009), the JPD represented by the BN can be obtained by multiplying all conditional probability tables (CPTs). Figure 2 shows the JPD represented by the BN in Figure 1. Now, the indicator variables can be added, being one per variables in the JPD. Figure 3 illustrated the JPD with indicator variables λ_x for each JPD variable x. One can now represent the JPD by summing all terms for each row of the table. The polynomial network representing the JPD in Figure 3 is:*

$$f \quad = \quad \lambda_a \lambda_b \theta_a \theta_{b|a} + \lambda_a \lambda_{\overline{b}} \theta_a \theta_{\overline{b}|a} + \lambda_{\overline{a}} \lambda_b \theta_{\overline{a}} \theta_{b|\overline{a}} + \lambda_{\overline{a}} \lambda_{\overline{b}} \theta_{\overline{a}} \theta_{\overline{b}|\overline{a}}, \tag{1}$$

where each variable A and B can assume values in $\{a, \overline{a}\}$ and $\{b, \overline{b}\}$, respectively.



A	Θ_A
true	$\theta_a = .3$
false	$\theta_{\bar{a}} = .7$

A	B	$\Theta_{B\|A}$
true	true	$\theta_{b\|a} = .1$
true	false	$\theta_{\bar{b}\|a} = .9$
false	true	$\theta_{b\|\bar{a}} = .8$
false	false	$\theta_{\bar{b}\|\bar{a}} = .2$

Figure 1: A BN defined by a DAG and a set of CPTs.

In 2011, Poon and Domingos (2011) extended the idea of the polynomial network for any unnormalized distribution. In practice, that means that any non-negative function

A	B	$\Pr(A, B)$
a	b	$\theta_a\theta_{b\|a}$
a	$\bar{b}$	$\theta_a\theta_{\bar{b}\|a}$
$\bar{a}$	b	$\theta_{\bar{a}}\theta_{b\|\bar{a}}$
$\bar{a}$	$\bar{b}$	$\theta_{\bar{a}}\theta_{\bar{b}\|\bar{a}}$

Figure 2: A JPD represented by the BN in Figure 1.

A	B	$\Pr(A, B)$
a	b	$\lambda_a\lambda_b\theta_a\theta_{b\|a}$
a	$\bar{b}$	$\lambda_a\lambda_{\bar{b}}\theta_a\theta_{\bar{b}\|a}$
$\bar{a}$	b	$\lambda_{\bar{a}}\lambda_b\theta_{\bar{a}}\theta_{b\|\bar{a}}$
$\bar{a}$	$\bar{b}$	$\lambda_{\bar{a}}\lambda_{\bar{b}}\theta_{\bar{a}}\theta_{\bar{b}\|\bar{a}}$

Figure 3: The JPD from Figure 2 with indicator variables for each JPD variable.

can be represented as a polynomial network. This extension is now known as *sum-product networks* (SPNs).

One can draw the network polynomial as a directed acyclic graph (DAG). Here, each multiplication is represented by a multiplication node, where the children are the factors involved in that particular multiplication. Sum nodes represent summing terms of the polynomial. The polynomial is written in a way that allows a pattern to form in the correspondent DAG. This pattern is a layer of sum and a layer of product. Thus, the name of the model SPNs.

Example 2 *The JPD in Figure 3 has a network polynomial described in Equation (1). This network polynomial can be draw as a DAG, as illustrated in Figure 4. Notice that multiplications in (1) are represented as a multiplication node in Figure 4 where the children are the involved factors. Similarly, sum in (1) are shown as sum nodes in Figure 4.t*

Inference in these models can be done by propagating up and down in the SPN DAG, as proposed by the *differential method* Darwiche (2003). It is out of the scope of this paper to explain the differential approach when computing marginals for all variables of the JPD. But it is of our interest the process of propagation itself. Consider, for instance, propagating up in the DAG. Values for each node are computed by following the operators. Notice that nodes in the same layer can be computed simultaneously, since one does not require the value from the other.

Example 3 *Consider the SPN in Figure 4. When propagating up in this DAG, the first product layer is computed. Here, $\theta_{b|a}$ is multiplied with λ_b, for instance. But notice that λ_b can also be multiplied with $\theta_{b|\overline{a}}$, simultaneously.*

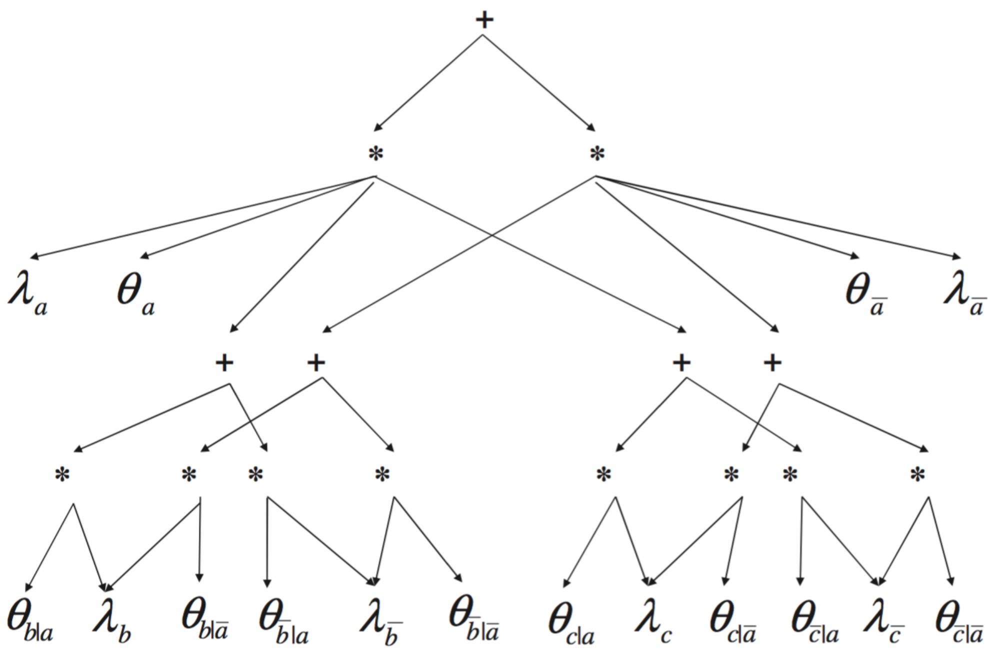


Figure 4: A SPN representing the JPD in Figure 3.

This paper proposes a method for parallelizing the computation at each layer of the SPN, as described in the next section.

3. The Comparison

As described in the research task C, a parallel implementation is proposed to make inference in a SPN faster. This parallel solution makes use of a GPU to parallelize computation within each layer. Since this solution is common to machine learning frameworks, in this implementation TensorFlow is used. The code is shown below.

```
import numpy as np
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Layer, Activation
import time
import tensorflow as tf

f = open("results.csv", "w")

for i in range(1,50):

    INPUT_SIZE = 10 * i
    OUTPUT_SIZE = INPUT_SIZE
    nb_class = 3

    batch_size = 128
    nb_epoch = 40

    np.random.seed(123)

    X_train = np.random.rand(INPUT_SIZE, nb_class)
    Y_train = np.random.rand(OUTPUT_SIZE, nb_class)

    X_test = np.random.rand(INPUT_SIZE)
    Y_test = np.random.rand(OUTPUT_SIZE)

    start_time = time.time()

    model = Sequential()
    model.add(Dense(INPUT_SIZE, input_shape=(nb_class,)))
    model.add(Activation('linear'))
    model.add(Dense(OUTPUT_SIZE))
    model.add(Activation('linear'))
    model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

    final_time = time.time()
    diff_time = final_time - start_time

    f.write(str(i)+","+str(diff_time)+","+"\n")

f.close()
```

In the code, we construct a network with 4 layers using random data. That is, no specific task is being learned by the network. Then, we perform inference on it during the learning phase. Also, at each iteration, we increase the size of the input layer, so we can compare the effect of using GPUs as the network gets slightly bigger. The code is run twice, one with the GPU turned on and another one with it off. The result is then illustrated in the graph of Figure 5.

The graph shows that using GPU makes inference, and therefore learning, faster. Even when the network gets bigger, using GPU is faster than CPU in a linear fashion. The peaks in both graph lines can be explained by the non deterministic way that the library creates and manages the networks. It is worth mentioning that the user only described how the network should be, but the library is the one which creates the network, after using optimization methods. Therefore, for the randomness of this experiment may cause the performance of these networks to fluctuate as shown in both graphs.

4. Conclusions

SPNs are layer-wise models used to represent JPDs in a compact and graphical way. Inference in a SPN can be done by propagating up and down on the SPN DAG. When propagating in the tree, operations in the same layer can be computed simultaneously, since they do not require input from within the same layer. In this way, one can perform inference in SPN hopefully faster.

The comparison of inference in SPNs using CPUs and GPUs shows that GPUs are faster even for bigger networks. Future works can study the effect of GPUs in massively large networks, when compared to small ones, like the one used in this paper.

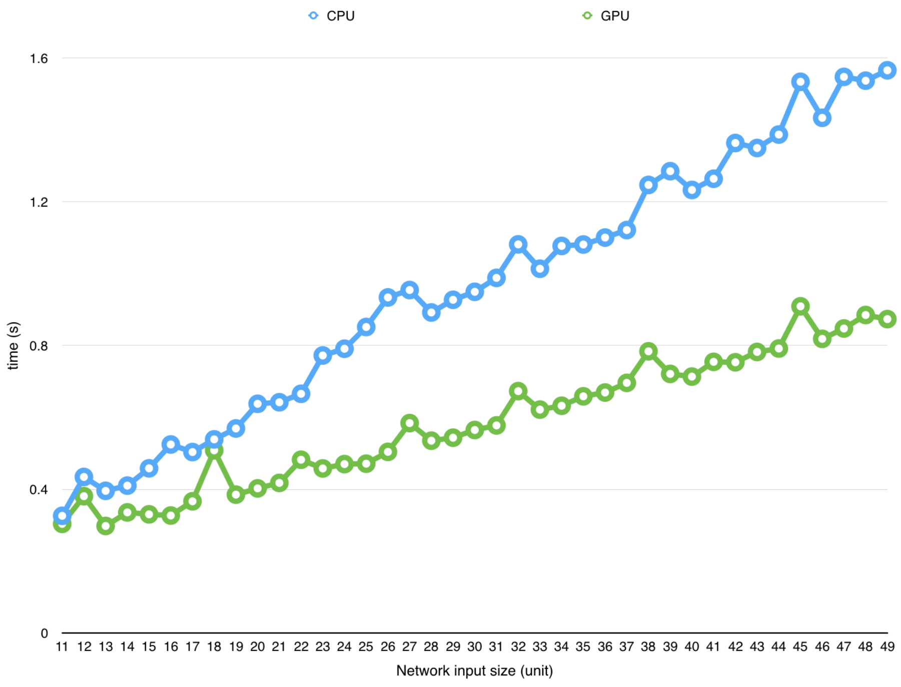


Figure 5: A comparison of inference using a deep network in a GPU and CPU.

References

Adnan Darwiche. A differential approach to inference in Bayesian networks. *Journal of the ACM*, 50(3):280–305, 2003.

Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 1 edition, 2009.

Daphne Koller and Nir Friedman. Probabilistic Graphical Models - Principles and Techniques. *MIT Press 2009*, 2009.

Hoifung Poon and Pedro M Domingos. Sum-Product Networks: A New Deep Architecture. *UAI*, pages 337–346, 2011.

TensorFlow. URL `https://www.tensorflow.org/`.

Theano. URL `http://deeplearning.net/software/theano/`.

Torch. URL `http://torch.ch/`.