Computation Discovery for Polynomials

Abstract

We present an approach based on attributive grammars for automatically discovering efficient ways to compute polynomial expressions. Our method scales to expressions with thousands of terms, otherwise intractable for humans. We use the approach to compute a Taylor series approximation for the partition function of a restricted Boltzmann machine (RBM). We show how to compute a 6th order approximation in polynomial time, compared to exponential time of the naive approach. More generally, our method can be regarded as a brute force search over proofs in formal system, defined by a set of axioms. It has the potential to be combined with machine learning techniques that can provide superior search strategies, enabling the learning of symbolic systems.

1. Introduction

Progress in some branches of mathematics might be restricted by size of symbolic derivations that human can handle. Mathematicians can deal with expressions having dozens of variables but would struggle to discover useful relations between hundreds or thousands of variables. We will show how the task of finding equivalent mathematical expressions can be automated. Our focus is on finding equivalent mathematical formulas (i.e. which give an identical numerical result to the original expression), but are faster to compute. The definition of faster can be either (i) the number of operations or (ii) the computational time for particular hardware.

We propose a deterministic, grammar based framework, which discovers relations between multi-variable polynomial expressions. First, we construct an attribute grammar – a context-free grammar extended to contain set of attributes, introduced by Donald Knuth (Knuth, 1968). To define the search space we use a set of context-free grammar rules representing admissible operations. By representing the cost of every operation as a synthesized attribute (i.e. computed bottom-up from child node attributes) we can search for formula with low time complexity. Through a linear combination of grammar elements, we can find a solution to the desired expressions. Finally, this computation solution can be further speed up by use of standard techniques of optimization in compiler domain. We show the power of the approach by deriving a closed form, $O(n^3)$ time solution (where n is the number of units) for a Taylor series approximation to the partition function of an RBM. This violates the conventional wisdom that accurate computation of the partition function requires an exponential number of elements.

XXX : Change according to Joan's section. Finally, we can regard the set of generated rules as a small axiomatic system. The presented algorithm is deterministic, and provably runs in finite time. We are excited about using machine learning for automatic reasoning, and this subset of mathematics seems to be perfect fit for testing automatic reasoning systems. Usually, axiomatic systems like number theory, set theory, or topology have very small number of axioms, but their application to terms might be complex. Proofs for such systems are difficult to represent in software, and there is no baseline algorithms which could brute force proof by iterating over all of them in reasonable finite time. Where in the contrary, presented here axiomatic system has very simple representation in software, and this paper presents brute force algorithm to find proofs. Surprisingly, even brute force algorithm is able to find new more efficient, computational expressions for expressions that we use.

2. Related Work

The attribute grammar, originally developed in 1968 by Knuth (Knuth, 1968) in context of compiler construction, has been successfully used as a tool for design, formal specification and implementation of practical systems. It has influenced areas of Computer Science such as natural language processing (Hafiz & Frost, 2011), (Starkie, 2002), definite clause grammars (Bratko, 2001), query processing (Koch & Scherzinger, 2007), (Ramakrishnan & Sudarshan, 1991) and specification of algorithms (Bellanova, 1984). Thirunarayan (Thirunarayan, 2009) provides a good overview of applications, including static analysis of programs, program translation, specifying information extraction algorithms and optimization of datalog programs.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

In our work, we apply attribute grammars to an optimization problem. This has previously been explored in range of domains: from well-known algorithmic problems like knapsack packing (ONeill et al., 2004), through bioinformatics (Waldispühl et al., 2002) to music domain(Desainte-Catherine & Barbar, 1994). However, we are not aware of any previous work related to discovering mathematical formulas. The closest work to ours can be found in (Cheung & McCanne, 1999) which involves searching over the space of algorithms and the grammar attributes are also computational complexity.

3. Terminology

We define the following vocabulary:

Space of matrices of polynomials
A matrix of homogeneous polynomials, which we denote by $\mathcal{P}_{\alpha}^{n\times m}$. Upper index indicates size of matrix, and lower indicates degree. For instance, $\begin{pmatrix} a^3+b^3 & b^3+bc^2+c^3 \\ cd^2 & d^3 \end{pmatrix} \in \mathcal{P}_3^{2\times 2}$.

Grammar
An attributive grammar, with Production rules defined on Expressions and semantic rules on Attributes. The attributes associated to every Expression, are: Computation, Time and Term.

Production rules
Syntactic rules defined on Expressions (transformation Expression → Expression), with associated semantic rules on Attributes. They are defined in figures 1 and 2.

Expression
Matrices of polynomials having a homogeneous degree α. They belong to the spaces $\mathcal{P}_{\alpha}^{n\times m}$. Different expressions might be in different spaces, and it restricts which production rules can be applied. Every expression has associated Attributes

Attribute
A value associated with Expression.

Computation
A piece of code to compute given Production rules on particular architecture or programming language. In our rules we consider Matlab as underlying computation language. For example, in Matlab the transpose operation on matrix $\mathbb{A}$ is denoted as $\mathbb{A}'$, so our computation for transpose production is $\mathbb{A}'$. If we ever decide to implement our framework in R, the Computation for transpose would be $t(\mathbb{A})$. Computation is an Attribute.

Element wise multiplication

$\mathbb{A} \in \mathcal{P}_{\alpha}^{n\times m}, \mathbb{B} \in \mathcal{P}_{\beta}^{n\times m} \rightarrow \mathbb{C} \in \mathcal{P}_{\alpha+\beta}^{n\times m}$

$\mathbb{C}.time := O(\mathbb{A}.time + \mathbb{B}.time + \mathbb{A}.n * \mathbb{B}.m);$

$\mathbb{C}.computation := \mathbb{A}.computation. * \mathbb{B}.computation;$

$\forall_{\substack{i\leq n \\ j\leq m}} \mathbb{C}.term[i][j] := \mathbb{A}.term[i][j] * \mathbb{B}.term[i][j];$

Matrix multiplication

$\mathbb{A} \in \mathcal{P}_{\alpha}^{n\times k}, \mathbb{B} \in \mathcal{P}_{\beta}^{k\times m} \rightarrow \mathbb{C} \in \mathcal{P}_{\alpha+\beta}^{n\times m}$

$\mathbb{C}.time := O(\mathbb{A}.time + \mathbb{B}.time + \mathbb{A}.n * \mathbb{A}.m * \mathbb{B}.m);$

$\mathbb{C}.computation := \mathbb{A}.computation * \mathbb{B}.computation;$

$\forall_{\substack{i\leq n \\ j\leq m}} \mathbb{C}.term[i][j] := \sum_{k=1}^{m} \mathbb{A}.term[i][k] * \mathbb{B}.term[k][j];$

Figure 1. Grammar rules operating on two expressions

Time
A computation time for a particular architecture and programming language (seconds), or computation complexity (polynomial). It is an Attribute.

Term
An instance of Expression. It is an Attribute.

4. Admissible Computation as a Grammar

Our goal is to find a fast way to compute the target expression in language and architecture of our choice. We will achieve it by developing literals in a grammar, up to degree of target expression. Next, we will look for the linear combination of obtained expressions to find how to express target expression exactly. If there are multiple literals with the same expression (up to multiplicative real constant), we choose one with the shortest computation time $t_{\mathcal{C}}$.

Let's ground our approach in an example. We assume that we are interested in finding algorithm with smallest operation complexity, and computation platform is a Matlab. We consider following set of operations on matrices:

To find any (or the cheapest) way of computing target expression $\mathbb{T}$ of degree k, we proceed as follows:

- develop grammar to obtain all possible literals up

Columns marginalization

$\mathbb{A} \in \mathcal{P}_\alpha^{n\times m} \to \mathbb{C} \in \mathcal{P}_\alpha^{n\times 1}$

$\mathbb{C}.time := O(\mathbb{A}.time + \mathbb{A}.n * \mathbb{A}.m);$

$\mathbb{C}.computation := sum(\mathbb{A}.computation, 2);$

$$\forall_{i\le n}\ \mathbb{C}.term[i][1] := \sum_{j=1}^{m} \mathbb{A}.term[i][j];$$

Rows marginalization

$\mathbb{A} \in \mathcal{P}_\alpha^{n\times m} \to \mathbb{C} \in \mathcal{P}_\alpha^{1\times m}$

$\mathbb{C}.time := O(\mathbb{A}.time + \mathbb{A}.n * \mathbb{A}.m);$

$\mathbb{C}.computation := sum(\mathbb{A}.computation, 1);$

$$\forall_{j\le m}\ \mathbb{C}.term[1][j] := \sum_{i=1}^{n} \mathbb{A}.term[i][j];$$

Columns repetition

$\mathbb{A} \in \mathcal{P}_\alpha^{n\times 1} \to \mathbb{C} \in \mathcal{P}_\alpha^{n\times m}$

$\mathbb{C}.time := O(\mathbb{A}.time + \mathbb{A}.n * m);$

$\mathbb{C}.computation := repmat(\mathbb{A}.computation, 1, m);$

$$\forall_{\substack{i\le n\\ j\le m}}\ \mathbb{C}.term[i][j] := \mathbb{A}.term[1][j];$$

Rows repetition

$\mathbb{A} \in \mathcal{P}_\alpha^{1\times m} \to \mathbb{C} \in \mathcal{P}_\alpha^{n\times m}$

$\mathbb{C}.time := O(\mathbb{A}.time + n * \mathbb{A}.m);$

$\mathbb{C}.computation := repmat(\mathbb{A}.computation, n, 1);$

$$\forall_{\substack{i\le n\\ j\le m}}\ \mathbb{C}.term[i][j] := \mathbb{A}.term[i][1];$$

Entry repetition

$\mathbb{A} \in \mathcal{P}_\alpha^{1\times 1}, t_{\mathcal{A}} \to \mathbb{C} \in \mathcal{P}_\alpha^{n\times m}$

$\mathbb{C}.time := O(\mathbb{A}.time + n * m);$

$\mathbb{C}.computation := repmat(\mathbb{A}.computation, n, m);$

$$\forall_{\substack{i\le n\\ j\le m}}\ \mathbb{C}.term[i][j] := \mathbb{A}.term[1][1];$$

Transposition

$\mathbb{A} \in \mathcal{P}_\alpha^{n\times m} \to \mathbb{C} \in \mathcal{P}_\alpha^{m\times n}$

$\mathbb{C}.time := O(\mathbb{A}.time + \mathbb{A}n * \mathbb{A}.m);$

$\mathbb{C}.computation := \mathbb{A}.computation';$

$$\forall_{\substack{i\le n\\ j\le m}}\ \mathbb{C}.term[i][j] := \mathbb{A}.term[j][i];$$

Figure 2. Grammar rules operating on one expression

Algorithm 1 Find computation for expression

Input: Target expression $\mathbb{T}$, initial expression W, $Rules = \{R_1, \dots, R_n\}$, maximum degree of polynomials k

Output: Computation $\mathcal{C}$ of expression $\mathbb{T}$.

```
Initialize set 𝕊 of all admissible literals with (W, 𝒲, t_W)
while 𝕊 grows do
  for all rule R ∈ Rules do
    𝕊² = {(x, y) : x ∈ 𝕊 ∧ y ∈ 𝕊}
    for all literal L ∈ 𝕊 ∪ 𝕊² do
      if can't apply R to L then
        continue
      end if
      L' ← Apply rule R to L
      if degree(L') > k then
        continue
      end if
      // If expression not yet in 𝕊 or it can be
      // computed faster, then add it.
      if L'.expr ∉ 𝕊 or L'.time < S[L'].time then
        Add L' to 𝕊
      end if
    end for
  end for
end while
Find linear combination of expressions from literals stored in 𝕊 to express 𝕋
Run optimizer (optional)
```

to degree k. It gives rise to finite number of literals.

- if there are multiple way of computing the same expression up to multiplicative constant, choose one with shortest time.

- find linear combination of expressions which is equal the target expression $\mathbb{T}$ (this relation is a symbolic relation, which is valid for any assignment of symbols). While solving this linear set of systems, choose solution with smallest sum of computation times. See 4.1 for more details.

- apply optimization like sub-expression elimination, and exploit distributive property of multiplication in order to decrees final computational time of $\mathbb{T}$. This step is optional, and it won't decrease computational complexity, but it can improve performance.

Pseudo code for the algorithm is shown in Algorithm 1.

4.1. Linear combination

We look for a linear combination of generated expressions in our literals, which is equal to the target expression. There are several possible scenarios:

- *single solution* - linear solver will have a unique solution (in practice this is uncommon)
- *no solution* - target expression is out of scope for the computation defined by our grammar.
- *multiple solutions* - we look for the linear combination with the smallest cost (computational time).

Multiple solutions case is essentially an integer programing problem, or knapsack problem, which are NP-complete. However, we can relax this to a linear program, which will give a solution almost as efficient as the optimal one. Experiments suggest that is good to also minimize number of coefficients, so avoiding unnecessary non-integer coefficient values, which can contribute numerical errors.

It is worth noting that while it is easy to find best solution in terms of *complexity*, it is NP-complete to find the best solution in terms of *performance*. In order to find best solution in terms of complexity, one has to run linear solver multiple times. Every time, it should include broader number of literals. Starting with set of literals having the smallest complexity (e.g. those which are $O(1)$), and adding new literals with higher and higher complexity ($O(n)$ and then $O(n^2)$ and so on). If algorithm, find solution at some point, than this solution is optimal in terms of complexity. However, discovery of the fastest performance (i.e. lowest runtime) requires exploration of all possible combinations of literals, which is far more expensive.

4.2. Computation derivation as the formal system

For Joan.

4.3. Representation of Expressions

Expressions belong to $\mathcal{P}_{\alpha}^{n\times m}$. There are various ways how they can be represented in the software. Such representation is crucial, and has consequences in correctness of solution (due to numerical errors), and computation speed.

Let us consider two matrices of polynomials:

$$\mathbb{A} = \begin{pmatrix} a^2+2ab+b^2 & a^2 \\ b^2 & ab \end{pmatrix}, \mathbb{B} = \begin{pmatrix} a+b & a \\ b & 2a+2b \end{pmatrix}$$

We consider element-wise matrix multiplication of $\mathbb{A}$ and $\mathbb{B}$. We derive result for different representation of expressions.

4.3.1. Symbolic representation

The most direct representation of expressions is to store its coefficients and powers of the every monomial. This representation is exact, and corresponds one-to-one with our description of operations defined by grammar. However, it has computational drawback. Let's assume that we would like to multiply two polynomials, each having 1000 monomials. Such multiplication would take roughly 10^6 operations. While the same operation on instantiations of this polynomials would cost just a one multiplication.

We have implemented expressions in such representation, and it allowed us to find patterns for polynomials up to degree 4 in ~ 8 hours of computation. We haven't run computation for higher degrees due to resource limitations. This representation guarantees correctness, and is easy to debug, however is quite slow (as the consequence of verboseness).

Let's ground this representation in an example of element-wise matrix multiplication of $\mathbb{A}$, and $\mathbb{B}$. Result of such multiplication is a matrix. We present results for the entry $(0,0)$. Entry $(0,0)$ of the matrix $\mathbb{A}$ is the list of coefficients $[1,2,1]$, and the list of powers $\begin{pmatrix} 2 & 1 & 0 \\ 0 & 1 & 2 \end{pmatrix}$. Entry $(0,0)$ of the matrix $\mathbb{B}$ is the list of coefficients $[1,1]$, and the list of powers $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. Entry $(0,0)$ of the element-wise multiplication of $\mathbb{A}$, and $\mathbb{B}$ is the list of coefficients $[1,3,3,1]$, and the list of powers $\begin{pmatrix} 3 & 2 & 1 & 0 \\ 0 & 1 & 2 & 3 \end{pmatrix}$.

4.3.2. Evaluation of polynomials in $\mathbb{R}$

Polynomials can be encoded by its evaluation at various, random points. This encoding simplifies operations on expressions, and multiplication of expressions in such representation is proportional to the number of evaluation points (for every evaluation point, we have to perform just a common multiplication).

We need evaluate our polynomials in number of points which is larger than final size of linear equations 4.1. We do not have to recover coefficients of polynomials, as we are just interest in finding proper linear combination of expressions.

This methods is numerically unstable, and for polynomials of degree $k \geq 3$, it is not able to produce solutions, even if there are some.

Let us ground this examples. We consider 2 points were we evaluate our polynomials: $a = 0.5, b = 1.5$, and $a = 2.5, b = 3.5$.

Then $\mathbb{A}, \mathbb{B}$ have the following representation:

$$\mathbb{A} = \begin{pmatrix} 4 & 0.25 \\ 2.25 & 0.75 \end{pmatrix}, \begin{pmatrix} 36 & 6.25 \\ 12.25 & 8.75 \end{pmatrix}$$
$$\mathbb{B} = \begin{pmatrix} 2 & 0.5 \\ 1.5 & 4 \end{pmatrix}, \begin{pmatrix} 6 & 2.5 \\ 3.5 & 12 \end{pmatrix}$$

Element-wise multiplication of expressions in this representation $\mathbb{A}$, and $\mathbb{B}$ is just element-wise multiplication of evaluations in the same points:

$$\begin{pmatrix} 8 & 0.125 \\ 3.375 & 4 \end{pmatrix}, \begin{pmatrix} 216 & 15.625 \\ 42.875 & 105 \end{pmatrix}$$

This representation is numerically unstable (we are getting a lot of round offs, and large values).

4.3.3. Evaluation of polynomials in $\mathbb{Z}_p$

We use similar strategy as in case of evaluations of polynomials in $\mathbb{R}$. We avoid numerical errors by replacing computation on real number with computation in $\mathbb{Z}_p$ group (for a large prime number p). This guarantees that all the results of the computation are exact (there is no round off errors on integers, and values are bounded by p).

This representation is fastest, and has no problem with numerical errors. However, it is difficult to debug. With such representation, we were able to develop completely grammars up to the degree 5.

Let's assume that we evaluate polynomials in points $a = 1, b = 2$, and $a = 3, b = 4$, and we work in Z_{11}. Then $\mathbb{A}, \mathbb{B}$ have the following representation:

$$\mathbb{A} = \begin{pmatrix} 9 & 1 \\ 4 & 2 \end{pmatrix}, \begin{pmatrix} 5 & 9 \\ 5 & 1 \end{pmatrix}$$
$$\mathbb{B} = \begin{pmatrix} 3 & 1 \\ 2 & 6 \end{pmatrix}, \begin{pmatrix} 7 & 3 \\ 4 & 3 \end{pmatrix}$$

Element-wise multiplication of expressions $\mathbb{A}$, and $\mathbb{B}$ is just element-wise multiplication of evaluations in the same points modulo 11:

$$\begin{pmatrix} 5 & 1 \\ 8 & 1 \end{pmatrix}, \begin{pmatrix} 2 & 5 \\ 9 & 3 \end{pmatrix}$$

5. Partition Function of RBM

The algorithm presented in Section 4 allows us to find concise formulas for polynomial expressions. However, many interesting functions are outside of this family. Therefore we consider Taylor expansion of desired function and use our approach to derive a fast way of computing the expansion in closed form.

Let $g(f, W)$ be the generalization of partition function for a binary RBM (Hinton, 2002). g is a functional, and it takes the following arguments $f : \mathbb{R} \to \mathbb{R}$, and weights W. It is defined as follows:

$$g(f, W) = \sum_{v \in \{0,1\}^n, h \in \{0,1\}^m} f(v^T W h)$$
$$f : \mathbb{R} \to \mathbb{R}$$
$$W \in \mathbb{R}^{n \times m}$$

We consider the computation of $g(x \to x^k, W)$ for a given power k, and for any $W \in \mathbb{R}^{n \times m}$ (and any size n, m). Potentially, if we would be able to compute $g(x \to x^k, W)$ for $k = 1, \ldots, K$, than partition function for finite energy $v^T W h < C$ could be approximated arbitrarily well. This is consequence of expressing as a finite sum approximation through Taylor expansion: $e^x = 1 + x + x^2/2! + x^3/3! + \cdots$.

5.1. Low degree examples

In order to present how our algorithm works, we will manually derive a fast computation procedure for $g(x \to x^k, W)$. However, this can be done manually only for very small $k = 1, 2$.

5.1.1. $\mathbf{g(x \to x, W)}$

Let's consider function $f(x) = x$. We will show that function $g(x \mapsto x, W)$ is computable in $O(nm)$ time (i.e. linear with respect to number of entries in W matrix).

$$g(x \to x, W) = \sum_{v \in \{0,1\}^n, h \in \{0,1\}^m} v^T W h$$

An entry $w_{i,j}$ in the sum is counted only if $v_i = 1$ and $h_j = 1$. Other variables $v_1, \ldots v_{i-1}, v_{i+1}, \ldots v_n$ and $h_1, \ldots h_{i-1}, h_{j+1}, \ldots h_m$ can be assigned arbitrarily, with the number of arbitrary assignments being 2^{n+m-2}. Hence:

$$\sum_{v \in \{0,1\}^n, h \in \{0,1\}^m} v^T W h = 2^{n+m-2} \sum_{i=1,\ldots,n, j=1,\ldots,m} W_{i,j}$$

The above mathematical formula (or description of computation) is a closed form solution for the sum over exponentially many elements. Note that its complexity is linear in size of W, which is $O(n^2)$.

5.1.2. $\mathbf{g(x \to x^2, W)}$

Now we wish to compute the following expression:

$$g(x \to x^2, W) = \sum_{v \in \{0,1\}^n, h \in \{0,1\}^m} (v^T W h)^2$$

There are multiple second order monomials that emerge:

- $w_{i,j}^2$ – present iff $v_i = 1, h_j = 1$. Appears 2^{n+m-2} times. We encode sum of all monomials like this as $(1, 0, 0, 0)$.
- $w_{i,j}w_{i,k}, j \neq k$ – present iff $v_i = 1, h_j = 1, h_k = 1$. Appears 2^{n+m-3} times. We encode sum of all monomials like this as $(0, 1, 0, 0)$.
- $w_{i,j}w_{k,j}, i \neq k$ – present iff $v_i = 1, v_k = 1, h_j = 1$. Appears 2^{n+m-3} times. We encode sum of all monomials like this as $(0, 0, 1, 0)$.
- $w_{i,j}w_{k,l}, i \neq k, j \neq l$ – present iff $v_i = 1, v_k = 1, h_j = 1, h_l = 1$. Appears 2^{n+m-4} times. We encode sum of all monomials like this as $(0, 0, 0, 1)$.

We encode above quantities in a vector, which indicate how many times each of the monomials appears. The vector expressing this relation for $g(x \mapsto x^2, W)$ is $(2^{n+m-2}, 2^{n+m-3}, 2^{n+m-3}, 2^{n+m-4})$

Rob: too big a jump. Need transition sentence here. Let us consider the following expressions:

- $\sum_{i=1,\ldots,n,j=1,\ldots m} W_{i,j}^2$ encodes $(1, 0, 0, 0)$. This expression contains only type of second degree monomials. It contains only monomials $w_{i,j}^2$, but not $w_{i,j}w_{i,k}$, or $w_{i,j}w_{k,j}$, or $w_{i,j}w_{k,l}$.
- $(\sum_{i=1,\ldots,n,j=1,\ldots m} W_{i,j})^2$ encodes $(1, 1, 1, 1)$.
- $\sum_{i=1,\ldots,n}(\sum_{j=1,\ldots,m} W)^2$ encodes $(1, 1, 0, 0)$.
- $\sum_{j=1,\ldots,m}(\sum_{i=1,\ldots,n} W)^2$ encodes $(1, 0, 1, 0)$.

By solving the linear equations:

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} x = 2^{n+m-4} \begin{pmatrix} 2^2 \\ 2^1 \\ 2^1 \\ 2^0 \end{pmatrix} \quad (1)$$

The solution to this equation is $x = [1, 1, 1, 1]^T$, thus

$$g(x \to x^2, W) = 2^{n+m-4} \Big(\sum_{i=1,\ldots,n,j=1,\ldots m} W_{i,j}^2 + (\sum_{i=1,\ldots,n,j=1,\ldots m} W_{i,j})^2 + \sum_{i=1,\ldots,n} (\sum_{j=1,\ldots,m} W_{i,j})^2 + \sum_{j=1,\ldots,m} (\sum_{i=1,\ldots,n} W_{i,j})^2 \Big)$$

Our algorithm derived the following equivalent Matlab computation for it:

```
(sum(sum(W)) .^ 2 + ...
sum(sum(W, 2) .* sum(W, 2)) + ...
sum(sum(W, 1) .* sum(W, 1)) + ...
sum(sum(W .* W))) * 2 ^ (n + m - 4)
```

The above derivation is still within the scope of human skill. However, manual derivation of $g(x \to x^k, W)$ for $k > 2$ quickly becomes intractable. Our algorithm is able to find such complex computational patterns automatically.

5.1.3. $\mathbf{g(x \to x^3, W)}$

The manual derivation for $k = 3$ is challenging. We present here all derived computations in for expressions in $\mathcal{P}_3^{1\times1}$. We will use following symbols to keep our notation concise:

```
A = sum(W, 2);
B = sum(W, 1);
C = sum(sum(W));
D = repmat(sum(W, 1), [n, 1]);
E = repmat(sum(W, 2), [1, m]);
```

All possible expressions up to degree 3 generated with our grammar are the following :

```
C
C .^ 2
sum(B .^ 2)
sum(sum(B .*  E .* W)))
sum(B .^ 3)
sum(A .^ 2)
sum(A .^ 3)
sum(sum((W .* W)))
C .^ 3
sum(sum(D .^ 2 .* E))
sum(B .* sum(W .* W, 1)))
C .* sum(sum((W .* W), 2))
sum(sum((E .^ 2 .* D))))
sum(A .* sum((W .* W), 2))
sum(sum(W .* W .* W))
```

Forming a linear system of these expressions and solving, we obtain the following expression for $g(x \to x^3, W)$, which is exactly equivalent to the original:

```
(C .^ 3 +
C .* sum(A .^ 2) * 3 +
C .* sum(sum(W .* W, 2)) * 3 +
C .* sum(B .^ 2) * 3 +
sum(A .* sum(W .* D, 2)) * 6)) / 64;
```

6. Experiments

We present various mathematical expressions, which can be computed more efficiently than initial formulation would suggest.

6.1. Matrix Multiplication-Sum Identities

Given matrices $A \in \mathbb{R}^{n\times m}$, $B \in \mathbb{R}^{m\times k}$, we wish to compute:

$$\sum AB = \sum_{i=1}^{n}\sum_{k=1}^{m}\sum_{j=1}^{k} A_{i,k}B_{k,j}$$

A naive algorithm would take $O(nmk)$ time. However, we have found with our framework computation giving the same expression in time $O(n(k+m))$:

```
sum(sum(A .* repmat(sum(B, 2), [1, m])'))
```

Empirical tests indicate that our expression is indeed faster to compute than the naive one (Figure 3).

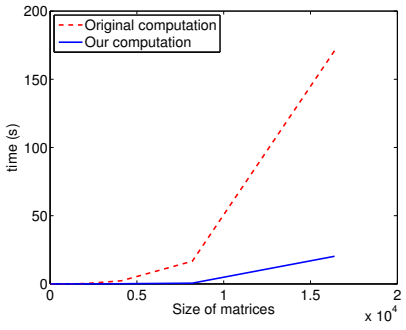

Figure 3. Computation time for $\sum AB$ using standard algorithm vs using inferred optimal algorithm.

Similarly, we found quadratic time computation, instead of cubic, for similar expressions such as:

$$\sum ABC,\ \sum AA^T,\ \sum AA^TA$$

As far we we are aware, these identities appear to be novel. Our system could automatically analyze large code repositories to find these and other expressions, which are currently computed inefficiently. Alternatively, our optimization rules could be placed into compilers to generate efficient code.

6.2. Partition Function Approximation

As we showed in Section 5, we can manually find $O(n^2)$ computation of $g(x \to x^k, W)$ for $k = 1, 2$ instead of native exponential time computation. By use of our framework, we found rules for $k = 3, 4, 5, 6$. However for $k = 4, 5, 6$ these rules used computation with $O(n^3)$ time (i.e. matrix multiplication). Finding computational rules for higher degree polynomials is expensive: Table 1 shows the time necessary to generate all the rules. It is worth noting that grammar can be evaluated just once, and the resulting coefficients can be stored. Furthermore, the process of discovering the computational rule for a given power only need to be performed once. However, due to limited computational power we analyzed only powers $k \leq 6$. As we note in Section 8.2 a future direction would be to learn patterns for $k = 2, 3, 4, 5, 6$ that allow us to generalize to $k > 6$ without exhaustively searching all possible rules.

Degree	Grammar size	Time (s)
2	5	17
3	15	188
4	48	2535
5	139	31320
6	437	434681

Table 1. Table summarizes size and computational time for grammars of specific degree. All computation here are performed on expressions, and have nothing to do with computation time on instantiations. This procedure has to be executed only once.

Degree	Num. terms	Complexity
2	4	$O(n^2)$
3	5	$O(n^2)$
4	21	$O(n^3)$
5	30	$O(n^3)$
6	106	$O(n^3)$

Table 2. Table summarizes complexity of computation of $g(x \to x^k, W)$. Number of terms, and complexity of every term is important for the computation of the target expression. Potentially, we need to evaluate partition function multiple times, so the "Complexity" is the complexity of our final system were we use such expressions.

Table 2 shows number of terms necessary to derive $g(x \to x^k, W)$ for various k. Figure 4 shows how well partition function is approximated with finite Taylor expansion. Finally, Figure 5 compares computation time of derived rules to the computation time of naive exponential time algorithm.

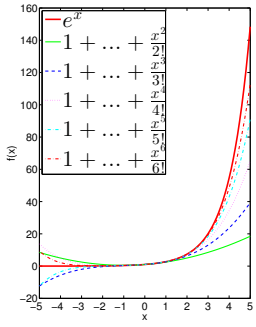

Figure 4. Comparison of approximations with various number of Taylor terms.

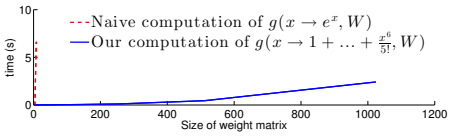

Figure 5. Comparison of computation time for naive exponential time algorithm vs our optimized derivation.

7. Conjecture on Hardness of Partition Function Approximation

Approximation of partition function is one of crucial problems in machine learning. Above framework gave us some potential future solutions to this issue, as well as the intuition about how hard this problem is. This section assumes what we have observed for $g(x \to x^k, W)$, and describes consequences on hardness of approximation. We now present conjectures relating to the hardness of approximation of the partition function. Unfortunately, thus far we are not able to prove, or disprove these conjectures.

Conjecture 7.1. *$g(x \to x^k, W)$ consists of a linear combination of terms for a matrix W generated by use of rules defined in Figure 1 and 2. Furthermore, the size of the generated grammar up to degree k grows like $O(\lambda^k)$ for some constant λ.*

The following corollary and lemma assumes that Conjecture 7.1 is valid.

Corollary 7.2. *$g(x \to x^k, W)$ can be computed in exponential time in k, but in $O(n^3)$ time with respect to size of W (W is of size $n \times n$).*

Lemma 7.3. *For $x < C$, e^x can be approximated up to ϵ accuracy in log scale with $\frac{Ce}{\epsilon}$ terms.*

Proof. For every $x < C$ there exists $u < x$, such that:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^{k-1}}{(k-1)!} + \frac{x^k}{k!}e^u$$

This implies

$$|e^x - (1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^{k-1}}{(k-1)!})| < \frac{C^k}{k!}e^C$$
$$\sim \exp(k \log C + C - k \log k + k) < \epsilon$$

We require that:

$$k(1 + \log C - \log k) < \log \epsilon - C$$
$$k(\log Ce - \log k) < \log \epsilon - C$$

Let us assume that $\log \epsilon < -1$, and $1 - Ce < -C$. These assumptions are valid if ϵ is small enough, and C is large enough.

For such ϵ and C, and $k > \frac{Ce}{\epsilon}$, the following inequalities are satisfied:

$$\frac{Ce}{\epsilon}(\log Ce - \log \frac{Ce}{\epsilon}) < \log \epsilon - C$$
$$\frac{Ce}{\epsilon} \log \epsilon < \log \epsilon - C$$
$$C < \log \epsilon (1 - \frac{Ce}{\epsilon})$$

This implies that is the energy of the RBM is bounded by C, we can approximate it up to ϵ approximation of partition function (in log scale) by using $\frac{Ce}{\epsilon}$ terms of the Taylor approximation. This algorithm assumes Conjecture 7.1 has complexity $O(\lambda^{\frac{Ce}{\epsilon}} n^3)$.

□

8. Discussion and Future Work

We have presented an automatic method for discovery of computations for polynomial expressions. The method itself is novel, as well as the derived approximations to the RBM partition function. There are four major directions of future research.

The first is focus on extension of our method to (i) richer class of functions than polynomials, (ii) larger set of production rules, (iii) more complex objects than matrices (e.g. tensors).

The second would be to use learning to generalize computation (Subsection , i.e. replace current brute force process with something more intelligent. E.g. We could generalize computation of $g(x \to x^k, W)$ for $k = 1, \ldots, 6$ to $k > 6$.

The third is concerned with computing the partition function and its derivatives, using approximations for learning, approximations for deep Boltzmann machines, optimization of the derived formulas, and understanding the mathematical principles behind formulas derived by our programs.

Finally, the forth direction are real-world applications of this framework on large code databases. We could automatically detect parts of code with expressions that have suboptimal time complexity and replace them.

We discuss in following subsections two first major directions of future research.

8.1. Extension of Grammar

The presented framework has some limitations, but also can be easily extended. First, we operate on polynomials instead of general functions. It is quite easy

to verify equality of polynomials, because it is enough to check equality of coefficients or evaluations on sufficiently many points (more than degree of polynomial). However, for generic functions it is not as simple. For example, if our basis would to include trigonometric functions, then the framework would have to be aware of many additional identities, e.g. $\sin^2 x + \cos^2 x = 1$, or $2 \sin x \cos x = \sin 2x$. Although challenging, we believe it is possible to extend our framework to handle such cases.

It is straight forward to extend our framework to matrices of fractional polynomials instead of matrices of polynomials. We could then include productions like matrix inverse and element-wise inverse of a matrix. Another direction would be to generalize matrices to arbitrary tensors for which it is even harder for humans to spot useful relations.

Finally, we are interested in exploiting productions which could discover recurrences. This would allow the discovery the fast Fourier transform, or Strassen's algorithm for fast matrix multiplication. This family of problems is quite broad, and crucial in computation.

8.2. Computation Generalization

This paper presents a brute force search over all possible computations, to yield terms that could lead to the target result. Although it works well for polynomials of small degrees $k \leq 6$, for larger powers brute force methods seems to fail. One of the contributions of this paper is to demonstrate how a small subset of mathematics may be explored with an automatic proving system. However, to broaden the range of math that we can address, we must move beyond brute force methods to more intelligent approaches which learn which rules are likely to be helpful, based on expressions for smaller powers. This would allow us move to polynomials of $k > 6$, making accurate estimation of the partition function for many deep networks possible. More broadly, it would bring machine learning techniques to the area of automatic theorem proving, which is another domain of human intelligence distinct from perceptual tasks (where machine learning is already effective).

References

Alon, Noga and Naor, Assaf. Approximating the cut-norm via grothendieck's inequality. *SIAM Journal on Computing*, 35(4):787–803, 2006.

Bellanova, A. Examples of algorithms specification by means of attribute grammars. *Calcolo*, 21(1):15–32, 1984.

Bratko, Ivan. *Prolog: programming for artificial intelligence*. Pearson education, 2001.

Cheung, Gene and McCanne, Steven. An attribute grammar based framework for machine-dependent computational optimization of media processing algorithms. In *Image Processing, 1999. ICIP 99. Proceedings. 1999 International Conference on*, volume 2, pp. 797–801. IEEE, 1999.

Desainte-Catherine, Myriam and Barbar, Kablan. Using attribute grammars to find solutions for musical equational programs. *ACM SIGPLAN Notices*, 29 (9):56–63, 1994.

Hafiz, Rahmatullah and Frost, Richard A. Modular natural language processing using declarative attribute grammars. In *Advances in Artificial Intelligence*, pp. 291–304. Springer, 2011.

Hinton, Geoffrey E. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.

Knuth, Donald E. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

Koch, Christoph and Scherzinger, Stefanie. Attribute grammars for scalable query processing on xml streams. *The VLDB journal*, 16(3):317–342, 2007.

Long, Philip M and Servedio, Rocco. Restricted boltzmann machines are hard to approximately evaluate or simulate. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 703–710, 2010.

ONeill, Michael, Cleary, Robert, and Nikolov, Nikola. Solving knapsack problems with attribute grammars. In *Proceedings of the Third Grammatical Evolution Workshop (GEWS04)*. Citeseer, 2004.

Ramakrishnan, Raghu and Sudarshan, S. Top-down vs. bottom-up revisited. In *Proceedings of the International Logic Programming Symposium*, pp. 321–336, 1991.

Salakhutdinov, Ruslan. Learning deep boltzmann machines using adaptive mcmc. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 943–950, 2010.

Starkie, Bradford. Inferring attribute grammars with structured data for natural language processing. In *Grammatical Inference: Algorithms and Applications*, pp. 237–248. Springer, 2002.

Thirunarayan, Krishnaprasad. Attribute grammars and their applications. *Encyclopedia of Information Science and Technology*, pp. 268–273, 2009.

Tieleman, Tijmen. Training restricted boltzmann machines using approximations to the likelihood gradient. In *Proceedings of the 25th international conference on Machine learning*, pp. 1064–1071. ACM, 2008.

Waldispühl, Jérôme, Behzadi, Behshad, and Steyaert, J-M. An approximate matching algorithm for finding (sub-) optimal sequences in s-attributed grammars. *Bioinformatics*, 18(suppl 2):S250–S259, 2002.