Efficient 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.

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

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.

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 using grammars. 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 $\mathcal{P}_{\alpha}^{n\times m}$. The upper and lower indices indicate the matrix size and polynomial degree, respectively. 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 an Expression.

Computation
A piece of code to compute a given Production rules on a particular architecture or programming language. In our rules we consider Matlab as the 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.

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 expressions in a grammar, up to degree of target expression. Next, we will look for the linear combination of obtained expressions to represent target expression exactly. If the target expression can be obtained in many ways, we choose the one with the shortest computation time.

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:

Element wise multiplication

$$\mathbb{A} \in \mathcal{P}_{\alpha}^{n\times m}, \mathbb{B} \in \mathcal{P}_{\beta}^{n\times m} \to \mathbb{C} \in \mathcal{P}_{\alpha+\beta}^{n\times m}$$
$$\mathbb{C}.time := O(\mathbb{A}.time + \mathbb{B}.time + \mathbb{A}.n * \mathbb{B}.m);$$
$$\mathbb{C}.computation := \mathbb{A}.computation. * \mathbb{B}.computation;$$
$$\forall_{\substack{i\le n\\ j\le m}} \mathbb{C}.term[i][j] := \mathbb{A}.term[i][j] * \mathbb{B}.term[i][j];$$

Matrix multiplication

$$\mathbb{A} \in \mathcal{P}_{\alpha}^{n\times k}, \mathbb{B} \in \mathcal{P}_{\beta}^{k\times m} \to \mathbb{C} \in \mathcal{P}_{\alpha+\beta}^{n\times m}$$
$$\mathbb{C}.time := O(\mathbb{A}.time + \mathbb{B}.time + \mathbb{A}.n * \mathbb{A}.m * \mathbb{B}.m);$$
$$\mathbb{C}.computation := \mathbb{A}.computation * \mathbb{B}.computation;$$
$$\forall_{\substack{i\le n\\ j\le m}} \mathbb{C}.term[i][j] := \sum_{k=1}^{m} \mathbb{A}.term[i][k] * \mathbb{B}.term[k][j];$$

Figure 1. Grammar rules operating on two expressions.

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

- Develop the grammar to obtain all possible expressions up to degree k. It gives rise to a finite number of expressions.
- If there are multiple ways of computing the same expression up to a multiplicative constant, choose one with shortest time.
- Find a linear combination of expressions which equals the target expression $\mathbb{T}$ (this relation is a symbolic relation, which is valid for any assignment of symbols). While solving this linear system, choose the solution with the smallest total computation time. See 4.1 for more details.
- Apply optimization like sub-expression elimination, and exploit the distributive property of multiplication in order to decrease the final computational time of $\mathbb{T}$. This step is optional, and won't decrease computational complexity, but can improve performance.

Pseudo code is shown in Algorithm 1.

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 $\mathbb{T}$

Input: Target expression $\mathbb{T}$, initial expression W, $Rules = \{R_1, \ldots, R_n\}$, maximum degree of polynomials k
Output: Computation $\mathcal{C}$ of expression $\mathbb{T}$.

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

4.1. Linear combination

We search for the linear combination of generated expressions which equal 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).

The multiple solutions case is essentially an integer programing problem, or knapsack problem, which is 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 it 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 expressions, starting with set of expressions of $O(1)$ complexity and adding new expressions with higher and higher complexity ($O(n)$ and then $O(n^2)$ and so on). If algorithm finds a solution at some point, then it is optimal in terms of complexity. However, discovery of the fastest performance (i.e. lowest run-time) requires exploration of all possible combinations of expressions, which is far more expensive.

4.2. Computation derivation as the algebra

The space of matrices of polynomials forms an algebra $\mathcal{A}$. A *grammar* $\mathcal{G} = \{g_1, \ldots, g_K; h_1, \ldots, h_{K'}\}$ defined on this algebra is a set of functions:

$$\begin{aligned} g_i &: \mathcal{A} \longrightarrow \mathcal{A}\ ,\ (i \le K)\ , \\ h_i &: \mathcal{A} \times \mathcal{A} \longrightarrow \mathcal{A}\ ,\ i \le K'\ , \end{aligned}$$

which might eventually only defined for certain subalgebras of $\mathcal{A}$ (e.g. matrix multiplication requires inner dimensions to agree).

Given an initial element $w \in \mathcal{A}$ and a target element $t \in \mathcal{A}$, the question is whether t can be written as a finite sequence of operations involving only w and $g, h \in \mathcal{G}$. If the algebra is defined over a discrete field, then the problem can be cast as a shortest path problem on a sparse directed graph. We define a "sink" node which contains all polynomials (or matrices of polynomials) with degree $\ge D_{\max}$ where ($D_{\max} > \deg(t)$), and a node for each of the other elements of $\mathcal{A}$. Two nodes x_1, x_2 have a connection $x_1 \to x_2$ if there exists $g \in \mathcal{G}$ such that $x_2 = g(x_1)$. The edge is assigned a cost which depends on the complexity of performing the operation. Similarly, we can define a 3-cycle to (x_1, x_2, x_3) if there exists $h \in \mathcal{G}$ such that $x_3 = h(x_1, x_2)$. The brute force algorithm constructs the graph by advancing sequentially from smallest to largest degree.

4.3. Representation of Expressions

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

Let us consider two matrices of polynomials:

$$\mathbb{A} = \begin{pmatrix} a^2 + 2ab + b^2 & a^2 \\ b^2 & ab \end{pmatrix} \tag{4.1}$$

$$\mathbb{B} = \begin{pmatrix} a + b & a \\ b & 2a + 2b \end{pmatrix} \tag{4.2}$$

We consider element-wise matrix multiplication of $\mathbb{A}$ and $\mathbb{B}$. We derive the result using three different ways of representating 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. It guarantees correctness and is easy to debug. However, it has computational drawback: multiplying two polynomials, each having N monomials is $O(N^2)$ compared to the same operation on instantiations of this polynomials, which is $O(1)$. The symbolic representation allowed us to find patterns for polynomials up to degree 4 in $\sim$ 8 hours of computation. The example of this representations for matrices 4.1 and 4.2 can be found in Table 1.

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

Polynomials can be encoded by their evaluation at various, random points. This encoding simplifies operations on expressions, and the time to multiply expressions is proportional to the number of evaluation points (for every evaluation point, we just have to perform floating point multiplication).

We need to evaluate our polynomials at a larger number of points than the final size of the linear system of equations (see Section 4.1). Note that we do not have to recover the coefficients of polynomials, as we are just interested in finding the proper linear combination of expressions. Unfortunately, this method is numerically unstable, and for polynomials of degree $k \ge 3$ it is unable to discover solutions, even if they exist.

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

We can avoid numerical issues by evaluating polynomials in the $\mathbb{Z}_\text{ı}$ group (for some large prime p), instead of $\mathbb{R}$. This guarantees all the results of the computation to be exact since there are no rounding errors on integers, and values are bounded by p.

Although this representation is the fastest, and has no problems with numerical errors, it is difficult to implement and debug. Using this representation, we were able to completely develop grammars up to degree 6. Table 1 shows the example of Eqns. 4.1 & 4.2 for $a = 1, b = 2$ and $p = 11$.

Table 1. Comparision of representation types described in section 4.3. The table contains instantiation of matrices 4.1 and 4.2, for $a = 1, b = 2$ (symbolic and $\mathbb{Z}_p$) or $a = 0.5, b = 1.5$ ($\mathbb{R}$) representations. For symbolic, we present only the representation of the top-left (1,1) matrix cell and the letter α denotes the vector of coefficients.

Representation	$\mathbb{A}$	$\mathbb{B}$	$\mathbb{A}$.* $\mathbb{B}$
Symbolic (cell 1,1)	$\begin{pmatrix}2 & 1 & 0\\ 0 & 1 & 2\end{pmatrix}$ $\alpha = [1, 2, 1]$	$\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$ $\alpha = [1, 1]$	$\begin{pmatrix}3 & 2 & 1 & 0\\ 0 & 1 & 2 & 3\end{pmatrix}$ $\alpha = [1, 3, 3, 1]$
$\mathbb{R}$	$\begin{pmatrix}4 & 0.25\\ 2.25 & 0.75\end{pmatrix}$	$\begin{pmatrix}2 & 0.5\\ 1.5 & 4\end{pmatrix}$	$\begin{pmatrix}8 & 0.125\\ 3.375 & 3\end{pmatrix}$
$\mathbb{Z}_p$ $(p = 11)$	$\begin{pmatrix}9 & 1\\ 4 & 2\end{pmatrix}$	$\begin{pmatrix}3 & 1\\ 2 & 6\end{pmatrix}$	$\begin{pmatrix}5 & 1\\ 8 & 1\end{pmatrix}$

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 a Taylor series approximation of the desired function and use our approach to derive a fast way of computing the polynomial terms of the expansion in closed form.

Let $g(f, W)$ be the generalization of the partition function for a binary RBM (Hinton, 2002), for $f : \mathbb{R} \to \mathbb{R}$ and $W \in \mathbb{R}^{n\times m}$. We define a functional g as follows:

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

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 $k = 1, \ldots, K$, then the partition function for finite energy $v^T W h < C$ could be approximated arbitrarily well. This is a 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 $g(x \to x^k, W)$, practical only for $k = 1, 2$.

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 $O(n^2)$, compared to the exponential complexity in n of the original expression.

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 the 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 $(2^{n+m-2}, 2^{n+m-3}, 2^{n+m-3}, 2^{n+m-4})$.

Now let us consider the following expressions:

- $\sum_{i=1,\ldots,n, j=1,\ldots m} W_{i,j}^2$. This expression contains only monomials $w_{i,j}^2$, but not $w_{i,j} w_{i,k}$, or $w_{i,j} w_{k,j}$, or $w_{i,j} w_{k,l}$. Hence it can be represented as $(1, 0, 0, 0)$.
- Similarly, $(\sum_{i=1,\ldots,n, j=1,\ldots m} W_{i,j})^2$ can be encoded as $(1, 1, 1, 1)$.

- $\sum_{i=1,\ldots,n}(\sum_{j=1,\ldots,m} W)^2$ encodes to $(1,1,0,0)$.
- $\sum_{j=1,\ldots,m}(\sum_{i=1,\ldots,n} W)^2$ encodes to $(1,0,1,0)$.

Using our encodings, we form the following linear system of equations:

$$\begin{pmatrix}1&1&1&1\\0&1&1&0\\0&1&0&1\\0&1&0&0\end{pmatrix} x = 2^{n+m-4}\begin{pmatrix}2^2\\2^1\\2^1\\2^0\end{pmatrix} \tag{5.1}$$

This has the unique solution $x = [1,1,1,1]^T$, meaning that the original expression can be rewritten as:

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

On this example, our algorithm derived the following equivalent Matlab computation which only requires $O(n^2)$ time (unlike the original which is exponential in n):

```
(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)
```

Although this derivation is still within the scope of human abilities, manual derivation of $g(x \to x^k, W)$ for $k > 2$ quickly becomes intractable. However, our algorithm is able to find such complex computational patterns automatically, thus can be used for larger k.

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

The manual derivation for $k = 3$ is challenging. We present all generated expressions up to degree 3 (for notation clearity, we first define variables A-E).

```
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]);

C, C .^ 2, C .^ 3, sum(B .^ 2),
sum(B .^ 3), sum(A .^ 2), sum(A .^ 3)
sum(sum(B .*  E .* W))), sum(sum((W .* W)))
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 the 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 $O(n(k+m))$ time:

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

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

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 2 shows the time necessary to generate all the rules. Note that the grammar need only be evaluated once, with the resulting coefficients being stored. Furthermore, the process of discovering the computational rule for a given power also need only

Table 2. A summary of size and computational time for grammars of a specific degree. All computation here is performed on *expressions*, and has nothing to do with computation time on their instantiations (this is shown in Table 3). Note that this procedure has to be executed only once.

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

Table 3. A summary of the complexity of computation for $g(x \to x^k, W)$. Potentially, we need to evaluate partition function multiple times, so the "Complexity" is the complexity of our final system which exctly reproduces the Taylor-series approximation, the naive computation of which would be exponential n (see Figure 3(d)).

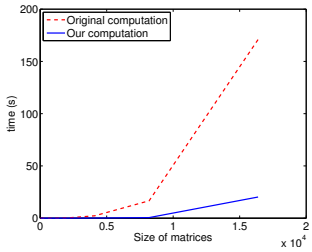
Degree	Num. terms	Complexity
2	4	$O(n^2)$
3	5	$O(n^2)$
4	21	$O(n^3)$
5	30	$O(n^3)$
6	106	$O(n^3)$

be performed once. However, due to limited computational power we were able to analyze 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. Table 3 shows number of terms necessary to derive $g(x \to x^k, W)$ for various k. Figure 3(b) shows how well partition function is approximated with finite Taylor expansion. Figure 3(c) shows the approximation error of the Taylor series for W. When the weights are small so is the energy and our approximation is accurate. With larger weights (and hence energy) the accuracy is diminished and more Taylor terms are needed. Finally, Figure 3(d) compares computation time of derived rules to the computation time of naive exponential time algorithm.

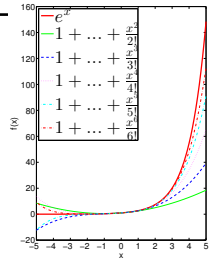
7. Conjecture on Hardness of Partition Function Approximation

Approximation of the partition function is one of the most important problems in machine learning. Our framework gives a potential solution to this issue, as well an intuition about how hard this problem is. This section assumes what we have observed for $g(x \to x^k, W)$ holds true for powers beyond $k = 6$, and presents conjectures relating to the hardness of approximation of the partition function. Unfortunately, thus far we have not been able to prove, or disprove these conjectures.

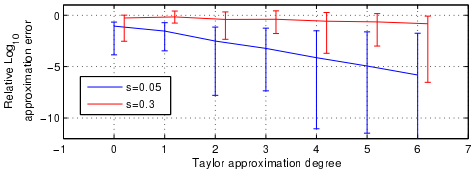
Conjecture 7.1. *$g(x \to x^k, W)$ consists of a linear combination of terms for a matrix W, generated by use of rules defined in Figure 1 and 2. Furthermore, the size of the generated grammar up to degree k grows like $O(\lambda^k)$ for some constant λ.*



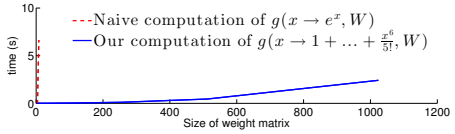
(a) Computation time for $\sum AB$ using standard algorithm vs our inferred optimal algorithm.



(b) Approximations of e^x using Taylor series.



(c) Relative approximation error of the partition function for different degrees of Taylor approximation. Max, mean and min errors are shown for 100 different trials for W of size 7×7, with $W_{ij} \sim$ Laplacian(s) and $s = 0.05$ (blue) and $s = 0.3$ (red).



(d) Comparison of computation time for naive exponential time algorithm vs our optimized derivation.

Figure 3. Experimental results.

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:

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

Using Moiver's formula ($k! \sim [const] * \frac{k^{k+1/2}}{e^k}$) we get:

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

To find the number of terms required to approximate with accuracy ϵ, we look for the smallest k satisfying:

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

Let us assume that:

$$\log \epsilon < -1 \tag{7.1}$$

$$1 - Ce < -C \tag{7.2}$$

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:

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

Using assumption 7.1 we get:

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

Which is true, because of assumption 7.2 ($\epsilon < 1 \Rightarrow \frac{Ce}{\epsilon} > Ce$).

This implies that if $max_{v,h}\, v^T W h < C$, we can approximate the partition function up to ϵ (in log scale) by using $k = \frac{Ce}{\epsilon}$ terms of the Taylor series. 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 discovering efficient way to computate polynomial expressions. The method itself is novel, as well as the derived approximations to the RBM partition function. There are three major directions of future research:

1. Extend our method to (i) a richer class of functions than polynomials, (ii) a larger set of production rules, (iii) more complex objects than matrices (e.g. tensors). We elaborate on this in Section 8.1.

2. Use machine learning to generalize computation, i.e. replace current brute force grammar search process with something more sophisticated. This would allow, for example, computation of $g(x \rightarrow x^k, W)$ for $k > 6$ which is needed for accurate approximation of the partition function for large energies. This is discussed further in Section 8.2.

3. Explore real-world applications of this framework on large code databases. It could be used to automatically detect parts of code with expressions that have suboptimal time complexity and replace them with more efficient expressions.

8.1. Extension of Grammar

The presented framework has some limitations, but also can be easily extended. First, we operate on polynomials instead of generic functions. It is quite easy to verify equality of polynomials, but for generic functions it becomes more complex. 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 become prohibitive. 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 solutions to more intelligent approaches which learn rules that are likely to be helpful, based on expressions for smaller powers. This would allow us to apply our method for polynomials of higher degrees, 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.