Efficient Computation Discovery for Polynomials

Abstract

We present an approach based on attribute grammars for automatically discovering efficient ways to compute polynomial expressions. We show how this technique can be used to marginalize over huge sets. For instance, how to marginalize dropout over all possible masks, or how to marginalize process of data augmentation. Moreover, we show how to compute exactly partition function for RBM like expressions in polynomial time. More generally, our method can be regarded as a graph search algorithm in the space of computations. It can be used to find close form for marginalization of any symmetric enough expression, which can be expressed as polynomial.

1 Introduction

We 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) for marginalization over huge sets. Problems of marginalization are prevalent in machine learning, and exact marginalizations can be a replacement for sampling methods.

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.

Example 1 shows our framework applied to a simple matrix expression: `sum(sum(A*B))`, where `A` and `B` are matrices. Naively, this takes $O(n^3)$ to compute due to the matrix multiply operation. The example outlines how our framework can find an alternate way of computing *exactly* the same result, but in $O(n^2)$ time. We start with this simple (non-machine learning example) to explain how system works.

We demonstrate the flexibility of our technique by applying it to various problems in machine learning:

1. A Taylor-series approximation to the partition function of a binary RBM. We can produce a 6th order approximation which is computable in closed-form which takes $O(n^3)$ time (where n is the number of visible & hidden units). Naive evaluation of the Taylor-series approximation would require summing over the 2^N possible states.

2. Closed-form computation of Dropout. Dropout Hinton et al. [2012b] involves the stochastic deletion of outputs within a dense neural network layer at training time. For n output units, the 2^n possible deletion patterns are sampled independently for each training sample. We use our framework to derive a closed-form expression that integrates out over all 2^n states.

3. Data augmentation is for object recognition purposes it consists translations, rotations, contrast alternation and few others. We show how to construct update rule to weights which integrates over all translation up to few pixels, and which integrates over all contrast alternations.

The main contribution of the paper is the introduction of a general framework for discovering efficient closed-form expressions. This are of the main use for marginalization problems (or sum over large number

of expressions). Its flexibility means can be used on a wide range of problems encountered in math and AI, beyond those outlined above. We dedicate follow up papers to address properly each of aforementioned applications. This paper focuses mainly on framework description, and gives only brief description of applications.

Example 1:

We are given matrices $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{m \times k}$. We wish to compute `sum(sum(A*B))`, i.e. :

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

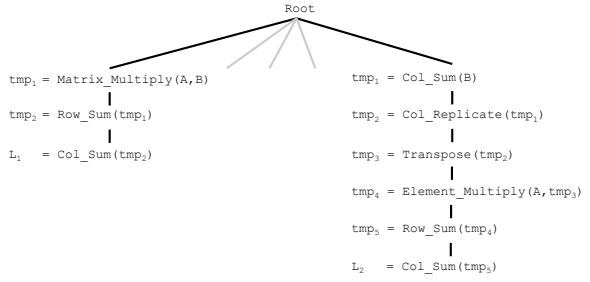
which naively takes $O(nmk)$ time. Our framework is able to discover an efficient version of the formula, that computes the same result in $O(n(k+m))$ time:

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

Our framework has four distinct stages:

1. An *attribute grammar* is defined (Section 4).
2. The *grammar tree* is built (see Algorithm 1). This contains an exhaustive set of all valid compositions of expressions from the grammar. While slow, this only needs to be performed once for a given grammar, irrespective of the number of target expressions we wish to compute. The figure below shows two branches of the grammar tree.
3. Each leaf of this tree is checked to see if the result matches the target expression. This may require solving a linear system, if the target expression has multiple terms (see Section 5.1). The two leaves $\mathtt{L}_1$ and $\mathtt{L}_2$ shown in the tree produce the same result as the target expression.
4. The computational complexity of the valid branches is then compared. The $\mathtt{L}_1$ branch is dominated by the `Matrix_Multiply` operation which is $O(nkm)$. The $\mathtt{L}_2$ branch has several operations that take $O(nk)$ or $O(nm)$ time, so is preferred.

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



2 Related Work

The attribute grammar, originally developed in 1968 by Knuth [1968] in context of compiler construction, has been successfully used as a tool for design, for-

mal specification and implementation of practical systems. It has influenced areas of Computer Science such as natural language processing (Hafiz and Frost, 2011, Starkie, 2002), definite clause grammars Bratko [2001], query processing (Koch and Scherzinger, 2007, Ramakrishnan and 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 a range of domains: from well-known algorithmic problems like knapsack packing ONeill et al. [2004], through bioinformatics Waldispühl et al. [2002] to music Desainte-Catherine and 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 and McCanne [1999] which involves searching over the space of algorithms and the grammar attributes also represent computational complexity.

3 Terminology

We define the following vocabulary:

Space of matrices of polynomials
A matrix of homogeneous polynomials, denoted 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 attribute 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 $\rightarrow$ 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 this restricts which production rules can be applied. Every expression has associated Attributes.

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

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

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

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

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

Figure 1: Grammar rules operating on two expressions.

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 a 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 Grammar Definition

For the purposes of this paper, we assume that we are interested in finding algorithm with smallest operation complexity, and computation platform is a Matlab. We define the grammar rules shown in Figures 1 and 2, which consist of the matrix operations: { `*`,`.*`, `'`, `sum`, `repmat` }.

Columns marginalization

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

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

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

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

Rows marginalization

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

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

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

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

Columns repetition

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

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

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

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

Rows repetition

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

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

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

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

Entry repetition

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

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

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

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

Transposition

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

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

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

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

Figure 2: Grammar rules operating on one expression.

5 Approach

Our goal is to find a fast way to compute the target expression in a language and architecture of our choice. We will achieve it by developing expressions in the grammar of Section 4, up to the degree of the target expression. Next, we will look for a linear combination of obtained expressions to represent the target expression exactly. If the target expression can be obtained in several ways, we choose the one with the shortest computation time.

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 5.1 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 the computational complexity, but can improve performance.

Pseudo code is shown in Algorithm 1.

5.1 Solving for the Target Expression

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

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)
```

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.

5.2 Representation of Expressions

Expression In the description above, we did not detail how the polynomial expressions $\mathcal{P}_{\alpha}^{n\times m}$ are represented in software. There are several possibilties 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:

$$\mathbb{A} = \begin{pmatrix} a^2 + 2ab + b^2 & a^2 \\ b^2 & ab \end{pmatrix} \tag{5.1}$$

$$\mathbb{B} = \begin{pmatrix} a + b & a \\ b & 2a + 2b \end{pmatrix} \tag{5.2}$$

We consider element-wise matrix multiplication of $\mathbb{A}$ and $\mathbb{B}$. We derive the result using three different ways of representating expressions.

5.2.1 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 ~ 8 hours of computation on a standard laptop. The example of this representations for matrices 5.1 and 5.2 can be found in Table 1.

5.2.2 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 5.1). 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 $k \geq 3$ it is unable to discover solutions, even if they exist.

5.2.3 Evaluation of polynomials in $\mathbb{Z}_p$

We can avoid numerical issues by evaluating polynomials in the $\mathbb{Z}_{\text{l}}$ group (for some large prime p), instead

Table 1: Comparision of representation types described in section 5.2. The table contains instantiation of matrices 5.1 and 5.2, 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)	$\begin{pmatrix} 2 & 1 & 0 \\ 0 & 1 & 2 \end{pmatrix}$ $\alpha = [1, 2, 1]$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ $\alpha = [1, 1]$	$\begin{pmatrix} 3 & 2 & 1 & 0 \\ 0 & 1 & 2 & 3 \end{pmatrix}$ $\alpha = [1, 3, 3, 1]$
$\mathbb{R}$	$\begin{pmatrix} 4 & 0.25 \\ 2.25 & 0.75 \end{pmatrix}$	$\begin{pmatrix} 2 & 0.5 \\ 1.5 & 4 \end{pmatrix}$	$\begin{pmatrix} 8 & 0.125 \\ 3.375 & 3 \end{pmatrix}$
$\mathbb{Z}_p$ $(p = 11)$	$\begin{pmatrix} 9 & 1 \\ 4 & 2 \end{pmatrix}$	$\begin{pmatrix} 3 & 1 \\ 2 & 6 \end{pmatrix}$	$\begin{pmatrix} 5 & 1 \\ 8 & 1 \end{pmatrix}$

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. 5.1 & 5.2 for $a = 1, b = 2$ and $p = 11$.

6 Results

We now apply our framework to a number of different problems.

6.1 Matrix Multiplication-Sum Identities

In addition to the identity shown in Example 1, our approach was also able to discover the following similar identities:

1. `sum(sum(A*B*C,2),1)` $\iff$
   `sum(A, 1) * (B * sum(C, 2))`

2. `sum(sum(A*A',2),1)` $\iff$
   `sum(sum(A .* repmat(sum(A, 2), [1, m])'))`

3. `sum(sum(A*A'*A,2),1)` $\iff$
   `sum(sum(((repmat(sum(A, 2), [1, m]) .* repmat(sum(A, 1), [n, 1])) .* A), 2), 1)`

As far we we are aware, these identities are novel. To make practical use of them, 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 RBM Partition Function Approximation

The algorithm presented in Section 4 allows us to find concise formulas for polynomial expressions. However, many interesting functions are outside of this family. One way around this is to use a Taylor series approximation of the desired function. The polynomial terms in the series may then be computed efficiently by formulae discovered by our algorithm.

One of the most important problems in machine learning is accurately estimating the partition function of a probabilistic model. We now show how we can use our approach to derive several terms in a Taylor series that approximates the partition function of one particular model, namely a binary RBM.

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: $e^x = 1 + x + x^2/2! + x^3/3! + \cdots$.

To illustrate how our automatic algorithm works, we first derive a fast computation procedure for low order terms in the series, i.e. $g(x \to x^k, W)$, for $k = 1, 2$.

6.2.1 $\mathbf{g(x \to x, W)}$

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.

6.2.2 $\mathbf{g(x \to x^2, W)}$

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:

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} x = 2^{n+m-4} \begin{pmatrix} 2^2 \\ 2^1 \\ 2^1 \\ 2^0 \end{pmatrix} \tag{6.1}$$

This has the unique solution $x = 2^{n+m-4} * [1, 1, 1, 1]^T$, 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. In the supplementary material we provide the expressions derived for $k = 3, 4, 5, 6$.

6.2.3 RBM Experiments

As we showed above, we can manually find polynomial time computation of $g(x \to x^k, W)$ for $k = 1, 2$ instead of native exponential time computation. Then, using our framework, we found rules for $k = 3, 4, 5, 6$.

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 be performed once. However, due to limited computational power we were able to analyze powers $k \leq 6$. As we note in Section 9.2, 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. These

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 $g(x \to x^k, W)$. The naive computation is exponential in n (see Figure 3(d)).

Degree	Num. terms
2	4
3	5
4	21
5	30
6	106

results show our method can be used on problems of real interets and, for certain weight matrices, provide a novel way of computing accurate solutions.

7 Dropout marginalization

In this section we will perform analogous analisys to the one from Subsection 6.2.1 for a single layer network with L_2 loss. First, we derive update rules for weights for a single layer neural networks without any regularization. Then we will confront it with dropout network. We will compute marginalization over all posible dropout masks (i.e. marginalization over all masks). We will describe how it can be discovered for more complex networks with our framework.

Single layer neural network with L_2 loss minimizes following objective:

$$\min_W ||WX - Y||^2, X \in \mathbb{R}^{f \times b}, Y \in \mathbb{R}^{g \times b}, W \in \mathbb{R}^{g \times f}$$

By differentiating above equation one gets gradient descent update rule on W (sign $\sim$ indicates that we drop multiplicative constants)

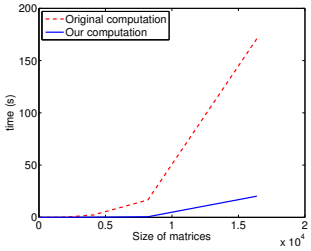
$$\partial W = 2(WX - Y)X^T \sim WXX^T - YX^T$$

We note for further derivation that $C = XX^T$ is the covariance matrix of X, and that $C_{i,j} = \sum_k X_{i,k}X_{j,k}$.

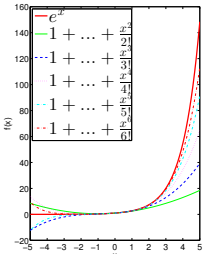
Dropout minimizes following objective:

$$\arg\min_W \mathbb{E}_M ||W(X. * M) - Y||^2, M \in \{0,1\}^{f \times b}$$

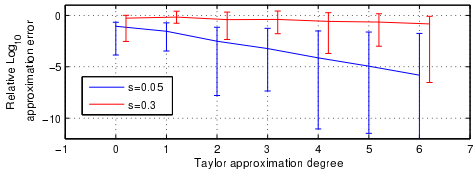
Expectation is over all possible mask assignments, and $.*$ is element-wise multiplication. For simplification let's assume that we drop neurons with probability 0.5. Expectation can be replaced with sum over all



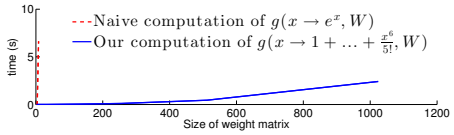
(a) Computation time for $\sum AB$ using standard algorithm vs our inferred optimal algorithm.



(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.

possible masks. Let's compute derivative with respect to weights W.

$$\partial W \sim \sum_M (W(X. * M) - Y)(X. * M)^T =$$
$$\sum_M W(X. * M)(X. * M)^T - Y(X. * M)^T =$$
$$W \sum_M [(X. * M)(X. * M)^T] - 2^{fb-1} YX^T =$$

Let's denote $D = \sum_M (X. * M)(X. * M)^T$.

$$D_{i,j} = \sum_M \sum_k X_{i,k}X_{j,k}M_{i,k}M_{j,k}$$
$$D_{i,j} \text{ for } i \neq j = 2^{fg-2} \sum_k X_{i,k}X_{j,k} = 2^{fg-2}C_{i,j}$$
$$D_{i,j} \text{ for } i = j = 2^{fg-1} \sum_k X_{i,k}X_{j,k} = 2^{fg-1}C_{i,j}$$
$$D = 2^{fg-2}(XX^T + XX^T. * I) \; I \text{ is identity matrix}$$
$$\partial W \sim \frac{1}{2} W(XX^T + XX^T. * I) - YX^T$$

Above derivation of dropout marginalization can be

discovered by our framework, and can be generalized to multiple layer network with rational (quotient of two polynomials) activation functions. More formally, it means that framework is able to discover how to update weights W for the following objective:

$$\arg\min_{W_1,W_2} \mathbb{E}_{M_1,M_2} ||W_2(f(W_1(X. * M_1)). * M_2) - Y||^2$$

f in this equation is a rational function.

8 Data augmentation marginalization

9 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 9.1.

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 $g(x \to x^k, W)$ for $k > 6$ which is needed for accurate approximation of the partition function for large energies. This is discussed further in Section 9.2.

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.

9.1 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.

9.2 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

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

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

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

G. Cheung and S. McCanne. 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, pages 797–801. IEEE, 1999.

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

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

G. Hinton, N. Srivastave, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. arXiv:1207.0580, 2012a.

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

G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012b.

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

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

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

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

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

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

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

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

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

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