fuzzy-eqsat: Equality saturation with approximations

Jeff Zhang
McGill University
Montreal, Canada

***Abstract*—Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.**

I. Introduction

This paper explores the feasibility of introducing rewrite rules representing **approximations** for linear algebra operations, such as matrix multiplication, using equality saturation. The concept of compilers automatically inserting approximations for operations, trading precision for performance, is not new. A prominent example of this is the `-ffast-math` flag in GCC [1], which speeds up floating point operations by allowing the compiler to make many assumptions that do not respect the IEEE standard.

The approach explored in this paper requires that values in the expression being optimized are known ahead of time, which is highly restrictive. Nevertheless, this remains applicable for speeding up neural network inference, as the model parameters are all known after training is complete.

II. Background

A. Equality saturation

The fundamental data structure used in equality saturation is the **e-graph**. E-graphs are a graph data structure that contain a set of **e-classes** uniquely identified by an ID. E-classes are equivalence classes each containing a set of equivalent **e-nodes**. E-nodes represent a term in the language, with child terms being e-class IDs.

An e-graph compactly represents a very large number of equivalent expressions for a given expression subject to a set of **rewrite rules**. For example, we might have a rewrite rule that replaces division by 2 by a bitshift:

$$\frac{x}{2} \to x \ll 1$$

[Add diagram]

Equality saturation works by repeatedly applying these rewrite rules until:

1) The graph is saturated (applying further rewrites would not result in any changes)
2) A timeout is reached

Note that a saturated e-graph represents all possible equivalent expressions reachable from the initial expression using the given rewrite rules.

Then, the best possible expression can be extracted from the e-graph according a specific **cost function** which outputs a cost value given an e-node. However, the extraction problem is known to be NP-Hard [2], as such computing an exact solution is intractable for large e-graphs.

Common solutions are greedy approximation algorithm (taking the minimum cost e-node from each e-class) and integer linear programming (computes an exact solution, but can be slow).

Due to the constraints of the cost model chosen in this paper, a genetic algorithm was used for extraction.

B. Genetic algorithms

A genetic algorithm is a heuristic search method for optimization problems inspired by the biological process of evolution.

The set of parameters that encode a solution are called a **chromosome**, and a single parameter is called a **gene**. It works by randomly generating a **population** of initial solutions. These solutions are evaluated according to some **fitness function**. Then, the best solutions are selected to "reproduce" by combining their chromosomes via some **crossover** method to produce the next population, with the hope being that combining two good solutions will produce an even better solution. Finally, random **mutation** is applied gene by gene in the new offspring to introduce more diversity in the solutions. [3]

This process is repeated for some number of iterations, and the best solution is returned.

III. Methods

This approach to equality saturation comes with the restriction that all variable values in the expression are known ahead of time, since otherwise it is not possible to compute a relative error for approximations. The method was applied to the problem of neural network optimization for inference on two classification models.

1) A simple multilayer perception with 1 hidden layer
2) LeNet 5 [4], one of the earliest convolutional neural networks

Both of these models are trained on the task of hand-written digit recognition using the MNIST dataset [5].

A. Representation

The two neural network architectures used are both feedforward neural networks, as such they can easily be converted into a single expression for optimization.

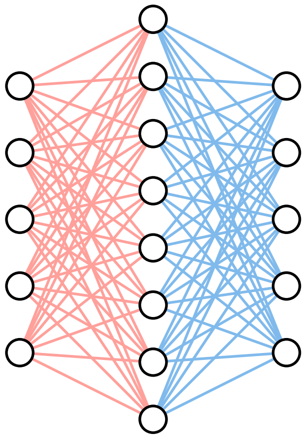

Fig. 1. A multilayer perception with 1 hidden layer.

`fuzzy-eqsat` uses the `egg` equality saturation library written in Rust, with the following grammar for expression rewrites.

Num	n		*integer literals*
Expr	e	$::=$ x	*variable*
		$\mid$ $e + e$	*matrix addition*
		$\mid$ $e * e$	*matrix multiplication*
		$\mid$ $\text{svd\_u}(x, n)$	*U matrix of truncated SVD*
		$\mid$ $\text{svd\_d}(x, n)$	*Σ matrix of truncated SVD*
		$\mid$ $\text{svd\_vt}(x, n)$	*V^T matrix of truncated SVD*
		$\mid$ $\text{ReLU}(e)$	*ReLU function*
		$\mid$ $\tanh(e)$	*hyperbolic tangent function*
		$\mid$ $\text{softmax}(e)$	*softmax function*

The network shown in Fig. 1 matches the architecture of the first classification model tested, albeit with fewer neurons. In `egg`, which uses an s-expression syntax, this network corresponds to the expression:

$$(\text{softmax}\ (+\ (*\ w_1\ (\text{ReLU}\ \ (+\ (*\ w_0\ x)\ b_0))\ b_1))$$

where w_i and b_i correspond to the weight matrix and bias vector for the i-th layer respectively, and x corresponds to the model input.

The model parameters are constants, but the model input by definition can't be known ahead of time. This can be resolved by treating x as a random variable which is defined by some distribution. The value assigned to x is the test set matrix from the classification problem's data set, since this approximates the distribution of possible model inputs.

B. Rewrite rules

The two approximation rewrite rules used were truncated singular value decomposition (SVD) and pruning. In addition to these, a basic rewrite rule for matrix multiplication associativity is also applied since this can give small cost reductions for free:

$$A(BC) \to (AB)C$$

a) ***Truncated SVD****:* The singular value decomposition of an $m \times n$ matrix A is a decomposition of the form

$$A = U\Sigma V^T$$

where U is an $m \times r$ matrix, Σ is an $r \times r$ diagonal matrix, and V is an $n \times r$ matrix, and $r = \min(m, n)$.

The truncated SVD is obtained by taking only the first k columns of each matrix, for some $1 \leq k < r$, giving us an approximation for A

$$A_k = U_k \Sigma_k V_k^T$$

This gives the **best rank-k** approximation for A by the Eckart-Young theorem.

Given a matrix multiplication AB, we rewrite A into its truncated SVD.

$$AB \to U_k \Sigma_k V_k^T B$$

If A is $m \times n$, and B is $n \times p$, then the total cost is mnp multiplications. When multiplying by the truncated SVD instead, the total cost is

- $V_k^T B$: knp multiplications
- $\Sigma_k \big(V_k^T B\big)$: kp multiplications since Σ_k is diagonal
- $U_k \big(\Sigma_k \big(V_k^T B\big)\big)$: kmp multiplications

$knp + kp + kmp$ can be much smaller than mnp if k is sufficiently small. In the rewrite implementation, the truncated SVD for several k are inserted in a linearly decreasing manner. The step size is equal to $\lfloor \log_2(\min(m, n)) \rfloor$. For example, for a 784×100 matrix, $\lfloor \log_2(\min(784, 100)) \rfloor = 6$, thus the truncated SVDs for $k = 94, 88, 82, \ldots$ are merged into the node's eclass. This rewrite is only applied once per e-class because since applying an SVD multiple times is redundant. This rewrite is also only applied to terms whose values can be known ahead of time, such as plain variables. Note that this means the cost of the svd_u, svd_d, svd_vt terms in the language is 0, because the decomposition is computed ahead of time during equality saturation.

b) ***Pruning***: Pruning is the process of truncating small values in a matrix to 0.

If Z is the amount of zeroes in matrix A, then to compute AB where A is $m \times n$, and B is $n \times p$, the number of multiplications required is $(mn - Z)p$ since we don't have to consider the zero entries in A.

The pruning rewrite rule is applied to terms that can be known ahead of time such as plain variables or terms of their SVD. It simply introduces a new variable for the pruned value. This is done instead of adding a prune term to the language because the rewrite

$$A \to \text{prune}(A, k)$$

introduces a cycle into the e-graph, which would complicate extraction significantly. The actual rewrite is

$$A \to P_{i,k}$$

where $P_{i,k}$ is a fresh variable name that has the value of matrix A where all x in A such that $x < 10^k$ are truncated to 0. Similar to the truncated SVD, multiple pruning thresholds are inserted into the eclass. The values of k chosen were −3, −2, and −1.

C. *Cost model*

The cost model used differs from typical equality saturation in many ways.

`egg` provides functionality to attach arbitrary data to eclasses during e-graph saturation via the `Analysis` trait.

First, during the analysis phase, a *canonical* expression value is computed for each eclass, which is derived from the variable values. The root e-class's canonical value corresponds to the actual model output given the test set.

The cost function implemented not only outputs an integer cost, but also the actual value of the e-node and its relative error compared to the e-class's canonical value. The relative error η between the canonical value A and its approximation A' is computed in terms of the Frobenius norm:

$$\eta = \frac{\|A - A'\|_F}{\|A\|_F}$$

The cost function itself can be configured by passing a maximum allowed relative error. The ordering between costs is determined using this value.

```
impl PartialOrd for CostWithErrorBound {
    fn partial_cmp(&self, other: &Self) ->
Option<Ordering> {
        if self.error > self.max_rel_error &&
other.error > self.max_rel_error {
            self.error.partial_cmp(&other.error)
        } else if a_err > self.max_rel_error {
            Some(Ordering::Greater)
        } else if b_err > self.max_rel_error {
            Some(Ordering::Less)
        } else {
            self.cost.partial_cmp(&other.cost)
        }
    }
}
```

D. *Genetic algorithm implementation*

The e-graph extraction problem can be formulated as follows: Given an e-graph, pick an enode from each e-class so that the resulting expression's cost is minimized.

Chromosome: A solution to the extraction problem is simply a mapping from e-class IDs to indices, with the indices corresponding to a specific e-node in that e-class. This is represented using a hashmap.

Population size: The population size used was 50, which is somewhat small however the cost function is quite expensive to compute due to the size of the matrices. 50 was chosen to strike balance between extraction time and quality of solutions produced.

Crossover: Crossover is done using simple single point crossover. The 2 parent solutions are flattened into an array, and a random index is chosen to be the crossover point. The two slices past the crossover point are then swapped, producing 2 children.

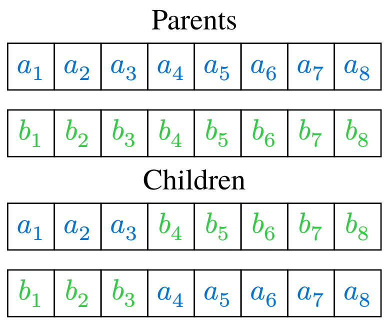
Parents
a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8
Children
a_1 a_2 a_3 b_4 b_5 b_6 b_7 b_8
b_1 b_2 b_3 a_4 a_5 a_6 a_7 a_8

Fig. 2. Single point crossover.

Mutation: The mutation rate chosen was 5%, mutation simply randomly selects a different index for an e-class.

IV. Experiments

V. Results & Discussion

VI. Related Work

VII. Conclusion

References

[1] “Options That Control Optimization ,” 2025, [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[2] M. B. Stepp, “Equality Saturation: Engineering Challenges and Applications,” pp. 79–83, 2011.

[3] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

[4] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, doi: 10.1109/5.726791.

[5] L. Deng, “The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web],” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012, doi: 10.1109/MSP.2012.2211477.