NAP: Noise-Based Sensitivity Analysis for Programs

Jesse Michel* MIT
Sahil Verma* IIT Kanpur
Benjamin Sherman MIT
Michael Carbin MIT

1 Introduction

Low-precision approximation of programs allows for faster computation, especially in fields such as machine learning, data analytics, and vision. Such approximations automatically transform a program into one that approximates the original output but executes much faster. At the heart of this approximation is sensitivity analysis – understanding the program's robustness to various perturbations. Sensitivity analysis provides a metric for importance that may be used to select how much each parameter may be modified in the search for fast and accurate approximate programs.

We propose *NAP* (Noise-based Analyzer of Programs) which provides a novel sensitivity analysis to model perturbations of each operator and variable in an expression. NAP performs sensitivity analysis by introducing independent Gaussian noise to each of the parameters to be approximated and then optimizing the parametrized variances over the stochastically sampled model (see section 2 for an example). These variances describe a distribution over approximations, which we term a noise envelope. If a parameter p is less sensitive to perturbations, then p forms a larger noise envelope, i.e. the optimal variance of the Gaussian for p will be large. Likewise, a parameter that is more sensitive to perturbations will form a smaller noise envelope.

NAP defines a novel constrained optimization problem to maximize the volume of the noise envelope while still meeting a given quality-of-service in the form of a constraint on the expected error. Using SGD for the optimization yields sensitivities for each parameter. We explore the advantages and disadvantages of optimizing for expected error rather than maximum error. We validated NAP's sensitivities by using them to generate mixed-precision approximate programs for a set of scientific computing benchmarks and for the weights of a neural network and found both to perform well.

2 NAP (by Example)

We will first give an overview of our approach and then show how this applies to a simple example. NAP takes as input a program f, a distribution $\mathcal{D}$ over inputs to f, and a loss function $\mathcal{L}$ that describes how good approximate outputs are, and it produces a sensitivity analysis. The following is the pipeline for sensitivity analysis (using NAP) and approximation:

1. Add 0-mean Gaussian noise parametrized by variance to every variable and operator.
2. Solve the optimization problem to obtain variances.
3. Use the variances to find a mixed-precision allocation for variables and operators.

For example, consider the quadratic function $x^2 + 2x + 1$ or, parametrizing the operators,

$$f(x, \{\times_0, +_1, \times_2, +_3\}) = x \times_0 x +_1 2 \times_2 x +_3 1.$$

NAP provides sensitivity analysis by explicitly inserting variables $\epsilon_k \sim \mathcal{N}(0, \sigma_k^2)$ for each operator $\odot_k$ and $\epsilon_y \sim \mathcal{N}(0, \sigma_y^2)$ for each variable y. Inserting these variables gives

$$g(x; \vec{\epsilon}) = [(x + \epsilon_x)(x + \epsilon_x) + \epsilon_0] + \epsilon_1 + [2(x + \epsilon_x) + \epsilon_2] + \epsilon_3 + 1.$$

Letting $\vec{\sigma}$ be the vector of standard deviations, NAP uses SGD to solve the optimization problem

$$\max_{\vec{\sigma}} \Big(\underbrace{\sum_i \log(\sigma_i)}_{\text{noise envelope}} - \lambda \underbrace{\mathop{\mathbb{E}}_{x \sim \mathcal{D}, \vec{\epsilon} \sim \mathcal{N}(\vec{0}, \vec{\sigma}^2)} \mathcal{L}(x, \vec{\epsilon})}_{\text{minimize error}} \Big) \tag{1}$$

where $\lambda > 0$ is a regularization parameter and in this example the loss $\mathcal{L}$ is the squared error

$$\mathcal{L}(x, \vec{\epsilon}) = \left(g(x; \vec{0}) - g(x; \vec{\epsilon}) \right)^2.$$

NAP uses the optimal variances to determine the approximate program by assigning each operator $\odot_k$ and variable y an allocation of mantissa bits. Imagine $x \sim \mathcal{U}[-1-\delta, -1+\delta]$ in our running example. If δ is large, NAP will make σ_x small since there will be a large noise envelope of roughly $\sigma_x x$ to the final answer. For small δ, $\sigma_x x$ is small allowing for σ_x to increase. Verifying this experimentally, NAP assigns a standard deviation σ_x of four times the magnitude when $\delta = 0.1$ versus $\delta = 1$. Thus, different data distributions result in different sensitivities and thus different variable and operator precisions. We estimate the expected error by averaging over samples from the data distribution.

3 Case Study: Accuracy

We used NAP to generate mixed-precision approximate programs satisfying the given expected error constraint. NAP is well suited to mixed precision approximation due to it's flexibility (noise envelopes can be trained over arbitrary parameters), control over the accuracy-noise trade-off, and interpretability.

We compare expected error produced by NAP to maximum error bounds produced by FPTuner on a suite of numerical programs for scientific computing from the FPBench benchmark set [1]. One would expect that for a fixed number of bits

*These authors contributed equally to the paper.

WAX'19, June 22, 2019, Phoenix, AZ, USA

the expected error (from NAP) will be lower than maximum error (from FPTuner) and for an expected error equalling the maximum error, fewer bits should be needed. Indeed, we found this was the case (Table 1).

Program inputs for FPBench programs have a specified range. We generated distributions over inputs by assuming each input is uniformly distributed over its range and that inputs are independent. We then ran NAP to create mixed-precision approximate programs (using MPFR) and evaluated their root-mean-square error (RMSE) [4]. FPTuner [1] provides upper bounds on the error for double-precision versions of these FPBench programs for *any* possible inputs within the range, so we set λ (using binary search) to yield RMSEs just lower than that bound. Table 1 shows these RMSEs in comparison to the FPTuner error bounds, together with the mean number of mantissa bits used in our approximate programs to achieve those RMSEs. This gives some measure of how weakening from worst-case bounds to average-case bounds allows further approximation.

Benchmarks	FPTuner	RMSE	Mean Bits
verlhulst	3.79e-16	3.72e-16	50
sineOrder3	1.17e-15	7.90e-16	50
predPrey	1.99e-16	1.73e-16	50
sine	8.73e-16	8.34e-17	51
doppler1	1.82e-13	7.75e-14	51
doppler2	3.20e-13	1.07e-13	51
doppler3	1.02e-13	5.10e-14	51
rigidbody1	3.86e-13	1.37e-13	51
sqroot	7.45e-16	4.00e-16	50
rigidbody2	5.23e-11	6.08e-12	51
turbine2	4.13e-14	2.35e-14	50
carbon gas	1.51e-08	3.01e-09	49
turbine1	3.16e-14	1.13e-14	51
turbine3	1.73e-14	1.40e-14	50
jet	2.68e-11	1.07e-11	50

Table 1. We compare FPTuner's maximum error bound against NAP's empirical root-mean-squared error. Mean bits is the average number of bits in the mantissa of the approximate program (vs. FPTuner's 52-bit mantissa).

This relaxed requirement gives NAP the flexibility to generate tighter expected error bounds using fewer bits than FPTuner for all of the benchmarks.

4 Case Study: Scalability

An additional challenge for FPTuner (and other solver-based techniques) is that they do not scale to larger programs. To evaluate FPTuner's scalability on larger numerical programs, we evaluated FPTuner on two programs. The first computes the sum of elements of the matrix obtained by multiplication of two 5 by 5 input matrices. The second computes the result of applying a sigmoid function to the dot product of an input vector with a weight vector each of size 50. These programs model the computation of a neuron in a neural network. We applied FPTuner to both programs with an error bound of 10^{-15} and FPTuner timed out given a threshold of 15 hours.

To validate NAP's scalability, we used it to perform mixed-precision quantization and pruning of deep neural networks. In the case of neural network quantization, we only add (and train) noises on model weights and not on the operators. We update variances for 50 epochs on the MNIST dataset.

Figure 1 presents our results on a LeNet-style architecture for MNIST [8]. Our LeNet-style architecture involves arithmetic over 27K parameters [7].

After training the variances, for each weight w we have a tuned standard deviation σ and we define

$$f(\sigma) = \lfloor c - \log(\sigma) \rfloor$$

for a constant c that is varied in order to tune the average bitwidth as shown in Figure 1. If $f(\sigma) \leq 0$ we prune w, and otherwise we quantize w to $f(\sigma)$ mantissa bits. We set the loss $\mathcal{L}$ used in equation 1 to be the categorical cross-entropy loss, which is consistent with the loss used to train the original network.

Figure 1 shows that even for low bitwidths, much of the accuracy of the model remains. For example, with an average of about 2 bits we achieve 95.3% test accuracy. With 8-bits, all of the approaches lose no more than 1% of the test accuracy.

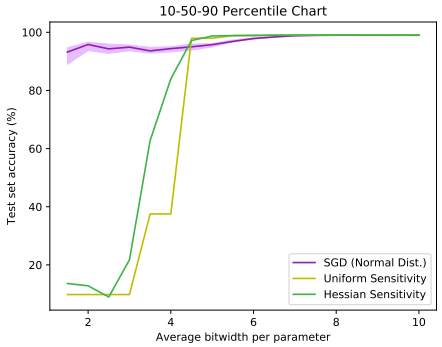

Figure 1. We show the space-accuracy tradeoff after quantizing weights (fixed-point quantization). At low bitwidths, NAP outperforms uniform and Hessian-based quantization. Because results generated from NAP (purple) are randomized, we ran it 30 times to show the distribution of results.

The Hessian sensitivity approach in Figure 1 computes the sensitivity of each parameter p as $\log(\frac{\partial^2 \mathcal{L}}{\partial p^2})$, which is a standard analytical technique [2].

Approximating tens of thousands of weights demonstrates that NAP's sensitivity analysis scales well to large numerical programs in a way unlike other approaches.

5 Related Work

Researchers have approached program approximation by performing loop perforations, function substitutions, and

quantization [3, 5, 9]. Other numerical tuning approaches provide maximum instead of expected error bounds, which scales poorly and is less well-suited for modern applications such as machine learning [1, 3]. Still others produce annotations that identify operations requiring high precision. [10, 11].

We compare a Hessian-based quantization approach to ours and show that at low bitwidths, it produces worse results than our approach [2]. Our sensitivity analysis is similar to the noise model used to compute generalization bounds on neural networks [6]. We hope to extend our work to provide generalization bounds on families of approximations.

References

[1] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir RakamariÄĞ. 2017. Rigorous floating-point mixed-precision tuning. *ACM SIGPLAN Notices* 52, 1 (2017), 300âĂŞ315. https://doi.org/10.1145/3093333.3009846

[2] Yann Le Cun, John S. Denker, and Sara A. Solla. 1990. Optimal Brain Damage. In *Advances in Neural Information Processing Systems*. Morgan Kaufmann, 598–605.

[3] Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 14* (2014). https://doi.org/10.1145/2535838.2535874

[4] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.* 33, 2, Article 13 (June 2007). https://doi.org/10.1145/1236463.1236468

[5] S. Han, H. Mao, and W. J. Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *ArXiv e-prints* (Oct. 2015). arXiv:cs.CV/1510.00149

[6] G. Karolina Dziugaite and D. M. Roy. 2017. Computing Nonvacuous Generalization Bounds for Deep (Stochastic) Neural Networks with Many More Parameters than Training Data. *ArXiv e-prints* (March 2017). arXiv:1703.11008

[7] Yann Lecun, LÃĺon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*. 2278–2324.

[8] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. (2010). http://yann.lecun.com/exdb/mnist/

[9] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and Accuracy-Aware Optimization of Approximate Computational Kernels. *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications - OOPSLA 14* (2014). https://doi.org/10.1145/2660193.2660231

[10] B. Nongpoh, R. Ray, S. Dutta, and A. Banerjee. 2017. AutoSense: A Framework for Automated Sensitivity Analysis of Program Data. *IEEE Transactions on Software Engineering* 43, 12 (Dec 2017), 1110–1124. https://doi.org/10.1109/TSE.2017.2654251

[11] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. 2014. ASAC: Automatic Sensitivity Analysis for Approximate Computing. *Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems - LCTES 14* (2014). https://doi.org/10.1145/2597809.2597812