IEEE Computer: Special Issue
Editor: Name, xxxx@email

Benchmarking data science: Twelve ways to lie with statistics and performance on parallel computers.

Torsten Hoefler
ETH Zürich

***Abstract*—Progress in artificial intelligence and machine learning is largely driven by growing compute capabilities of specialized accelerators and large datasets. A remarkable property is that model accuracy can be traded off for compute performance: the computation only needs to be as accurate as the statistical noise in the dataset. Given this sensitive trade off, we recognize that reproducibility and interpretability of compute performance of data science workloads differs fundamentally from both core HPC and machine learning. In this article, we humorously discuss twelve fallacies when focusing on compute performance that we frequently observed in practice—fast but wrong models are worse than slow models. We follow each with a recommendation to mitigate the danger and hope to contribute to establishing a good benchmarking etiquette for data science. Our work aims to start a discussion that eventually leads to better science in the quickly emerging field of "systems for AI".**

INTRODUCTION

Data science, Artificial Intelligence (AI), and Machine Learning (ML) have gained significant importance in the recent past. The AI revolution is in full swing after machine learning, specifically Deep Learning (DL) systems, have demonstrated to beat human performance in many tasks. Market analysts expect that AI and DL workloads will soon consume the vast majority of compute cycles spent in datacenters, edge, and consumer devices. This development will be further fueled by the quest towards more general human-like intelligence. OpenAI, for example, follows a strategy that trains extremely large models (e.g., GPT-3 with 175 billion parameters) on a large language corpus gathered from human-written texts. It is speculated that training just one single model costs upwards of $10 million with growing costs for even larger models.

Thus offering cost-effective high-performance computer systems for artificial intelligence and deep learning is becoming not just a large market but a necessity. It is of utmost importance that researchers, software, and hardware engineers compare performance and cost effectiveness of such architectures in *reproducible and interpretable ways*.

When designing systems, software, or algorithms for AI *we must consider the intersection between the general fields of data science and systems engineering*. In this paper, we discuss a set of fallacies that emerge from this intersection and we propose mitigations as well as a general methodology to improve reproducibility and interpretability in the quickly growing field of "Systems for AI".

Interpretability has long been a sore topic in data science—going back to Huff's now 67-year-old book "How to Lie with Statistics". While this book falls into the general category of *data literacy* (how to interpret data), it contains many foundational lessons that remain important for modern data science. The fundamentally hard problem of reproducibility and interpretability remains a hot topic today. There is rarely a year where not one or two keynotes at top-class machine learning conferences focus on these topics. Ali Rahimi's test of time award talk at NeurIPS'17 provoked a significant debate by comparing the state of the art in machine learning with alchemy in the medieval ages.

Ensuring reproducibility in data science is in part hard because algorithms usually base on statistics where we cannot (and do not want to) expect bit-wise reproducibility. The fundamentally stochastic nature of computations requires a more complex model for reproducibility: *instead of reproducing a specific result, we need to reproduce a distribution* [1]. Thus, the measure of success is often defined as achieving a specific *accuracy* on a test-set, e.g., classifying 95% of the examples correctly. Here, it does not matter which 95% set—implying that many different models can be considered to reproduce a specific result. In fact, due to complex hyperparameters that control the often nondeterministic learning algorithms, finding the exact same model in different training runs is a rarity, if not practically impossible.

Compute performance is similarly hard to reproduce because it also follows a stochastic process in complex systems. Even system availability (e.g., can I find a computer system from 10 years ago?) makes reproducing results hard and forces scientists and engineers towards interpretability [2]. In both cases—data science and compute performance—rigorous statistics are necessary to model either accuracy or compute performance. Similarly, in both cases, the fields are largely driven by empirical results. Combining rigorous benchmarking with rigorous data science is necessary to design, build, and understand next-generation artificial intelligence systems. For example, cost forces many to employ the most efficient (low bitwidth) datatype for learning systems but one must pay attention that the resulting lossy compression does not invalidate the results.

The rush for good results

Pressure and competition in the field are fierce—numerous startups, vendors, and large cloud providers compete for the expected trillion-dollar business. Efficiency is key because many tasks, especially grand-challenges, such as more general intelligence benefit from larger models. Thus, specialized parallel high-performance architectures are most important to train such large models. At the same time, models must be usable in practice, which limits the cost of training and as well as inference.

Startups are fighting for their place and often take extreme positions such as systems without large main memories or wafer-scale computing to claim their 2-30x benefit over all other solutions. Those approaches push the system balance into extreme regimes, which may be applicable to the workload but needs to be carefully analyzed. One big danger is to only focus on a specific aspect or metric of the system. For example, in many High-Performance Computing (HPC) environments, the floating point rate (flop/s) is seen as *the* measure of performance. This leads to many studies focusing on *«floptimization»*[1] and not sustained performance. Many researchers are in a similar position because competition to disseminate new ideas is fierce. With dozens of papers appearing each week on arXiv alone, fast publication of new results is imperative. In such

[1] we use «text» to indicate irony in the quoted text

high-pressure environments, it is most important to avoid pitfalls in performance optimization and analysis of data science workloads.

Life at the intersection

Performance optimization, measurement, and reproducibility has been a topic in the HPC community for decades. Sets of rules and best/worst practices exist for performance measurements [3], [2] and major HPC conferences launched serious reproducibility initiatives.

Similarly, the data science community has established a set of rules for ensuring result reproducibility [4] that have been endorsed by top ML conferences. Yet, the two fields are fundamentally different—the HPC field focuses on both results and performance, while today's machine learning field predominantly targets at results. Arguably, the result of HPC simulations can often be defined in a bit-reproducible manner [5], [6] while ML results are usually of stochastic nature requiring to share exact experimental setups [4].

Both communities focused on their specific fields in isolation—however, the fields are starting to mix in productive ways. This gives rise to machine learning tracks at traditional HPC conferences and systems tracks at traditional ML conferences or even new conferences aimed at the intersection, such as MLSys.

Our paper aims to bridge the silos between ML and HPC by identifying a set of fallacies in performance analysis and system design for data science workloads. These fallacies and resulting guidelines are useful for practitioners, system vendors, scientists, and end-users alike. We hope to establish some form of *benchmarking etiquette* for data science workloads. Vendors and scientists can check their own messages for violations of the etiquette while users can quickly identify the right questions to ask.

Twelve ways to fool the masses

We now identify twelve different fallacies that we continue to observe regularly in the wild such as in vendor white papers, demos, and product specifications but also many scientific papers, talks, and even prestigious award presentations. The methodology and style is inspired by Bailey's classical "twelve ways to fool the masses" [3]; we summarize each humorously, explain it in detail, and then follow with a recommendation. We believe that each of our recommendations contributes to establishing a good performance benchmarking etiquette in data science.

#1 Scale computations at all cost

«You should adjust your system to yield the highest possible performance. Forget about incidentals such as convergence or accuracy!»

The biggest peculiarity of data science workloads is that they enable a trade-off between accuracy and performance (implementation) aspects of the execution. While it is necessary to consider this trade-off, it can be very tempting to navigate it to one extreme and focus on performance at all cost. We have observed that accuracy was sacrificed for performance in HPC settings—what value does a fast but incorrect calculation have? While this trade-off and its misuse manifests itself in many data science workloads, let us consider Stochastic Gradient Descent (SGD) as an example in the following.

SGD and its variants form the basis of much of deep learning training today. Supervised SGD training works on examples that are sampled from the input and output sets of the true function to be learned. The training process adapts weight terms in a fixed computational structure (e.g., a neural network) to approximate the true function represented by those examples. SGD proceeds in an iterative process, where a subset of examples is used to calculate an averaged update for the weights in each iteration. This set is usually called "minibatch", and its size determines the quality of the overall algorithm—intuitively, if it is too small, the updates can be very noisy for complex functions or inexact examples and if the set is to large, nuances of the function to be learned could be lost in the averaging. Thus, the size of the minibatch, a simple algorithmic hyperparameter, is crucial for the accuracy of the resulting model.

The simplest way to parallelize training in deep learning is to replicate the full model and its weights and train on different parallel processors. Then, each processor computes the weight updates for a set of examples in parallel. This is possible because weights are only updated after processing a minibatch, so if there are E examples in a minibatch, one can employ up to E parallel processors with this technique. However, for efficiency reasons, one would need more than a single example per processor to re-use the weights, or in HPC language, turn a set of Matrix Vector (BLAS Level 2) operations into Matrix Matrix (BLAS Level 3) operations. The per-processor set of examples is often called “microbatch”. If we now have P processors, and a microbatch size of M, we would need $E > MP$ for our minibatch size.

As mentioned before, E is limited by statistical properties of the data and choosing E too large may negatively affect convergence [7]. Yet, if you want to set a speed record on a large-scale parallel computer, you would scale E to tens of thousands or more! This has been a typical issue in the early days and can still be seen regularly in practice.

Similarly, “weak scaling”, i.e., keeping the microbatch size per processor constant while increasing the number of processors, will change the statistics. If one keeps the number of epochs constant, it will reduce the update steps, which may be most relevant for training [8]. At the end, most learning workloads are strong scaling as the model size is constant and the set of examples is typically constant as well. Another very related technique for floptimization that is even more common is running hundreds of ensembles (that may later be averaged) to show computational performance and (in this case trivial) scaling. However, those additional ensembles may not improve the quality to rationalize the investment.

In general, one needs to carefully study admissible batch sizes [9] or use specialized methods to tune the optimization algorithm to support larger batch sizes for data parallelism [10]. One could also achieve higher parallelism with synchronous methods that do not change the statistics [11], [12].

When applying a technique that changes the calculation statistics, carefully consider its impact on the quality of the result.

#2 Trade convergence for performance

«Do not worry if your performance optimization slows convergence! Simply report the time per iteration!»

Machine learning workloads can be surprisingly robust and can work with substantial inaccuracies during the calculation as long as the overall statistics are maintained. This means that one can discard much of the calculation as long as, in expectation (eventually), the statistical properties of the calculation are maintained.

Examples include stochastic rounding for low-bit datatypes and top-k gradient methods where we send only the largest gradient values while accumulating the discarded gradients locally in data-parallel training [13], [14]. However, such methods usually slow down convergence of the model optimization and thus often require many more iterations to maintain the same accuracy as exact calculations. If you save 50% of the compute time per iteration but need 4x more iterations, then you are 2x slower at the end. Thus, one needs to be very careful when influencing convergence rates through approximation techniques!

Many works consider per-iteration times and rarely analyze convergence. Even if they analyze it, results are often presented separately, such as “we sped up iterations by 2x” in the case above.

For any optimization that may slow convergence, analyze and measure the resulting convergence rate for representative examples. Or simply always report the total runtime to find the final model.

#3 Do not consider generalization accuracy

«Train the biggest model to minimize training loss for highest performance.»

Large models often allow more efficient use of accelerators and parallelism. But model size is not always a guarantee for quality as large models can simply store all presented examples but still not approximate the true function they draw from well. This phenomenon is called “overfitting” and is a well-understood danger in data sciences. Yet, when presenting performance numbers, it is regularly overlooked.

For example, larger batch sizes required for highly-parallel training can reduce the generalization accuracy while achieving low training error, leading to a “generalization gap” [7]. Generalization can be improved by careful tuning of the training dynamics through hyperparameters such as adapting the learning rate during training or increasing the number of iterations [8].

Thus, even when mainly comparing performance, we need to test and report generalization (sometimes called test-) accuracy. In practice, this is done by evaluating the model on unseen examples from the same distribution. This may require careful tuning to achieve good generalization and shows again that data science and performance engineering must be combined. Since hyperparameters are important - we must make sure to document and share those for reproducibility - ideally, share the whole code and training setup.

Always measure and report generalization accuracy after performance optimizations.

#4 Do not report hyperparameter tuning cost

«Hyperparameter tuning can be expensive, so do not talk about it when reporting costs or runtimes!»

It may not be practical if one has to train a model 20 times in order to find the parameters that make it 10% faster on some hardware. Also, why would we need to train the same model 20 times? However, this is quite often done in practice when reporting performance in both, science and industry. Specifically when tuning for errors and error patterns coming from the computer itself such as inherent inaccuracies in analog devices, shortcuts in rounding modes, or quantization errors in low-bitwidth datatypes. Some of these errors may even be characteristic for each individual device.

Document all hyperparameter tuning required for achieving the results. Analyze whether the discovered parameters generalize to different examples, models, or even tasks.

#5 Report highest (exa)op/s rates

«Floptimization is about reporting the highest flop/s rates! Thus use the smallest datatypes—after all your laptop can do 10^{18} *bit flip (exa-)ops per second! Maybe you can even get away with reporting flop/s that you never do?»*

Many machine learning workloads allow aggressive optimizations for low bitwidth data representation. Reducing the number of bits in the number representation can lead to substantial speedups because energy and silicon area for integer multiplication shrink quadratically with the bitwidth. Furthermore, the required memory bandwidth and storage shrinks linearly with bitwidth. Low-width 4-8-bit integers are commonplace in inference and 4-16 bit floating points numbers in training. While low-precision datatypes are very effective in deep learning, they often form a trade-off between accuracy and performance: very low precision quantization losses accuracy, which should always be analyzed and reported. Some company brochures even name datatypes as 32 bits but in fact some bits are simply discarded during the calculation.

Sparse computations omit zero values during the computation or data loading—gaining performance benefits for compute or bandwidth. However, some vendors count operations that are never performed. Sometimes, but not always, such practice can be identified by the term “effective operations”. While managing sparsity requires often significant additional resources (storage and compute), those can hardly be counted as arithmetic operations.

Clearly specify the bitwidth of the operations performed and only count the actual operations that are computed. Most often, a mix of different precisions (bitwidths) is used for machine learning. In such cases, we can specify the relative proportions, e.g., “we achieved 1 exaflop/s with 85% 8-bit, 10% 16-bit, and 5% 32-bit floating point operations”.

#6 Show only kernels/subsets when scaling

«Always present the fastest kernel such as matrix multiplication as this will result in highest flop rates!»

A general danger when accelerating computations is to focus too much on one specific part of the problem. In early deep learning accelerators, computations were clearly limited by matrix multiplication performance. However, reducing the datatype bitwidth made the basic multiplications quadratically cheaper [15] while the memory bandwidth cost was only reduced linearly. Specialized acceleration units such as Tensor Cores lowered the multiplication overhead further. After accelerating those workloads by more than 10x, the bottleneck shifts to other aspects such as data movement. For transformer networks on modern hardware, 99.8% of the floating point operations only take 61% of the time, while the remaining 0.2% are data-movement bound, which takes 39% of the time [16]. This demonstrates that it can be very misleading to only show the best matrix multiply unit or to report only operation counts. The same idea is of course true for any subset kernel-selection!

Always consider a complete problem when benchmarking and showing performance results. When analyzing specific kernels, put them into proportion to the overall workload.

#7 Optimize only for one network

«If you carefully tuned your experiment, code, and architecture to a specific problem, then make sure to only talk about this!»

This is a standard issue similar to #6 but one abstraction level higher. Any compute system is developed with a (set of) specific use-case(s) in mind. However, the workloads usually have some variability and the compute system will be used for a variety of tasks. Thus, it needs to perform reasonably well for many scenarios and an honest analysis should test a variety of such scenarios.

The closest comparison in the HPC space are systems that were solely designed for a good top-500 score (solving a single large system of linear equations with Gaussian elimination). This is not necessarily representative of modern HPC workloads but high top-500 rankings still make a good selling argument. In machine learning, specific networks, such as ResNet or BERT begin to play a similar role - how fast can I train a ResNet-50 on ImageNet or BERT on a specific language corpus to state of the art accuracy? Such benchmarks are extremely useful and foster reproducibility and interpretability but we must be careful to not overfit the design to them.

Many of those examples are instances of Goodhart's law that roughly states that *when a benchmark becomes an optimization target, it looses its value as benchmark!*

Always analyze and present a carefully selected set of workloads covering the full workload space of interest.

#8 Compare outdated/general purpose HW to specialized HW

«Modern accelerators show highest speedups against old hardware - so make sure to compare to the oldest machine you can find!»

This is another classic problem - many works compare aged General Purpose Graphics Processing Units (GPGPUs) with their shiny new ML accelerator or even a Central Processing Unit (CPU) that was never meant for specialized computation. The resulting huge speedups have very little meaning. Similar problems arise when comparing completely different architectures, for example Field Programmable Gate Arrays (FPGAs) and GPGPUs.

For comparing different accelerator types, one should ensure that those are manufactured in a similar silicon process with a similar die size, design power, and cost. If accelerators are made in fundamentally different processes and/or die sizes, then one can scale the performance numbers by the difference in silicon efficiency. In any case, the exact comparison points need to be documented carefully.

Ensure a fair comparison for different hardware by selecting devices of equal cost and age or scaling accordingly.

#9 Don't worry about inference costs

«If you want to show quickest time to some accuracy, use a large model!»

OpenAI showed that increasing model size and stopping training before convergence achieves the same accuracy as more expensive training of smaller models [17]. While this sounds great from a training performance perspective, the main goal of learning is to later use the model in an inference setting. It may make sense to train large models and then distill them [18] but those additional costs must be included in the overall analysis.

For practical ML workloads, the center of attention should lie on inference efficiency because inference computations will dominate during the lifetime of most models.

#10 Do not consider reading the data

«When measuring performance numbers, make sure all needed data is already loaded into main memory!»

ML models are usually trained on large amounts of data and reading this data can be a substantial bottleneck. After all, the deep learning revolution is fueled by algorithms, *data*, and compute. Thus, for each iteration, data needs to be loaded from storage, converted, augmented, and computed by the model. Many toolchains exist for the *data input pipeline*, but Input/Output (I/O) is often ignored in performance experiments.

The data is not always coming from storage—examples for reinforcement learning environments often come from a simulation process executed on CPUs. Running those simulations and transmitting the data to the training accelerators can quickly become a bottleneck in highly-optimized learning processes. One general systems design issue is to have enough external bandwidth for I/O into the training system.

Consider the whole system pipeline when analyzing performance of machine learning workloads including storage access and other data sources.

#11 Train on unreasonably large examples

«Larger computations are simpler to parallelize and achieve higher flop rates, so make sure to choose the largest inputs for highest performance!»

Sensors often return high-resolution data. For example, modern commodity camera sensors record tens of megapixels but those are rarely needed for typical object detection or classification tasks. Following this observation, the ImageNet benchmark scales input images down by orders of magnitude for computational efficiency. This input feature compression and selection is important, otherwise, the network would need to learn the compression wasting valuable compute cycles and weight storage. Thus, one needs to carefully analyze how to format the input data to the machine learning model. This can be as important as the design of the model and the optimization algorithm itself.

Always define and consider input sizes, format, and transformations carefully in performance analysis of learning systems.

#12 Choose your comparison points well

«Make sure to only compare performance ***or*** *accuracy if your model is unlikely to win in both categories!»*

As machine learning is often about an accuracy-performance trade-off, it is necessary to look at both in tandem. Pareto optimality is a useful measure for comparing methods. Any method that is not dominated by the Pareto front may be useful in practice.

Another issue is how to present relative accuracy. Since approximation methods usually reduce accuracy, it is natural to report closeness to the state of the art result. Often, this is reported similar to "we achieve 99% of the baseline performance". However, what is meant by this phrase depends on the exact interpretation. Let us assume a 95% state of the art accuracy for some task. This means that the model function outputs the same as the true function in 9,500 out of 10,000 examples.

Some interpret 99% of baseline as 9,405 correct examples while others (falsely?) assume they can loose an absolute 1% going down to 9,400 correct examples. A second, more stringent, interpretation could consider the incorrect results. To be within 99% with respect to the incorrect results means to increase them by no more than 1%. In this case, we only allow 5 more incorrect examples, i.e., the model would need to classify 9,495 examples correctly. It is thus quite important to be precise when defining relative accuracy to state of the art results.

Always present both accuracy and performance and be precise when defining relative accuracies.

Discussion

We present twelve ways to sugercoat performance results of data science workloads. Many of those adjust the trade-off between accuracy and performance in order to shed the best possible light on the performance of specific machine learning systems. Each of these ways points at a potential problem with setups for analyzing performance of machine learning workloads. While we propose simple mitigations for each, we find that general principles emerge that can help to make performance analysis in this field transparent and reproducible. The most important principle here is documentation and transparency to enable interpretability of the results. This can ideally be achieved by sharing the whole experimental setup. We hope to spark a discussion and present a blueprint for sanity-checking results, reports, and presentations.

While we outline the interaction between data science and performance, we note that both fields have their own standards for scientific reproducibility (e.g., [4], [2]) that need to be considered! In addition, we recommend following the general good scientific practice [19].

All-in-all we aim this work to contribute to the definition of a good benchmarking etiquette in the quickly growing field of "Systems for AI".

Acknowledgments

The core of this article was sparked at the IPAM workshop "HPC for Computationally and Data-Intensive Problems" organized by Joachim Buhmann, Jennifer Chayes, Vipin Kumar, Yann LeCun, and Tandy Warnow. There, I presented a preliminary version of the twelve rules during a spontaneous evening talk and thank all attendees, especially Yann, Vipin, and Joachim for great comments and suggestions. The later refinement of the list was influenced by all my data science collaborators at SPCL, ETH Zurich, and Microsoft.

REFERENCES

1. V. Stodden, "Reproducing statistical results," *Annual Review of Statistics and Its Application*, vol. 2, no. 1, pp. 1–19, 2015.
2. T. Hoefler and R. Belli, "Scientific Benchmarking of Parallel Computing Systems," pp. 73:1–73:12, ACM, Nov. 2015. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15).
3. D. H. Bailey, "Twelve ways to fool the masses when giving performance results on parallel computers," *Supercomputing Review*, pp. 54–55, 08/1991 1991.
4. J. Pineau, P. Vincent-Lamarre, K. Sinha, V. Larivière, A. Beygelzimer, F. d'Alché Buc, E. Fox, and H. Larochelle, "Improving reproducibility in machine learning research (a report from the neurips 2019 reproducibility program)," 2020.
5. A. Arteaga, O. Fuhrer, and T. Hoefler, "Designing Bit-Reproducible Portable High-Performance Applications," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE Computer Society, Apr. 2014.
6. J. Demmel and H. D. Nguyen, "Parallel reproducible summation," *IEEE Transactions on Computers*, vol. 64, no. 7, pp. 2060–2070, 2015.
7. M. Li, T. Zhang, Y. Chen, and A. J. Smola, "Efficient mini-batch training for stochastic optimization," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, (New York, NY, USA), p. 661670, Association for Computing Machinery, 2014.
8. E. Hoffer, I. Hubara, and D. Soudry, "Train longer, generalize better: closing the generalization gap in large batch training of neural networks," 2018.
9. S. McCandlish, J. Kaplan, D. Amodei, and O. D. Team, "An empirical model of large-batch training," 2018.
10. Y. You, I. Gitman, and B. Ginsburg, "Large batch training of convolutional networks," 2017.
11. T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis," *ACM Comput. Surv.*, vol. 52, pp. 65:1–65:43, Aug. 2019.
12. S. Li and T. Hoefler, "Chimera: Efficiently Training Large-Scale Neural Networks with Bidirectional Pipelines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC21)*, ACM, 11 2021.

13. D. Alistarh, T. Hoefler, M. Johansson, S. Khirirat, N. Konstantinov, and C. Renggli, “The Convergence of Sparsified Gradient Methods,” in *Advances in Neural Information Processing Systems 31*, Curran Associates, Inc., Dec. 2018.
14. C. Renggli, D. Alistarh, M. Aghagolzadeh, and T. Hoefler, “SparCML: High-Performance Sparse Communication for Machine Learning,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*, Nov. 2019.
15. A. Karatsuba, “The complexity of computations,” *Proceedings of the Steklov Institute of Mathematics*, vol. 211, pp. 169–, 01 1995.
16. A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, “Data Movement Is All You Need: A Case Study on Optimizing Transformers,” in *Proceedings of Machine Learning and Systems 3 (MLSys 2021)*, Apr. 2021.
17. J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, “Scaling laws for neural language models,” 2020.
18. Z. Li, E. Wallace, S. Shen, K. Lin, K. Keutzer, D. Klein, and J. E. Gonzalez, “Train large, then compress: Rethinking model size for efficient training and inference of transformers,” 2020.
19. N. A. of Sciences Engineering, Medicine, *et al.*, *Reproducibility and Replicability in Science*. National Academies Press, 2019.

Torsten Hoefler is a full Professor of Computer Science at ETH Zurich, Switzerland. Before joining ETH, he led the performance modeling and simulation efforts of parallel petascale applications for the NSF-funded Blue Waters project at NCSA/UIUC. He is also a key member of the Message Passing Interface (MPI) Forum where he chairs the "Collective Operations and Topologies" working group. Torsten won best paper awards at the ACM/IEEE Supercomputing Conference 2010 (SC10), EuroMPI 2013, SC13, SC14, SC19, IPDPS'15, ACM HPDC'15 and HPDC'16, ACM OOPSLA'16, and other conferences. He published numerous peer-reviewed scientific conference and journal articles and authored chapters of the MPI-2.2 and MPI-3.0 standards. For his work, Torsten received the ACM Gordon Bell Prize in 2019, the IEEE TCSC Award of Excellence (MCR) in 2019, ETH Zurich's Latsis Prize in 2015, the SIAM SIAG/Supercomputing Junior Scientist Prize in 2012, and the IEEE TCSC Young Achievers in Scalable Computing Award in 2013. He was also awarded the BenchCouncil Rising Star Award in 2020. Following his Ph.D., he received the Young Alumni Award 2014 from Indiana University. Torsten was elected into the first steering committee of ACM's SIGHPC in 2013 and he was re-elected in 2016 and 2019. He was the first European to receive many of those honors he also received both an ERC Starting and an ERC Consolidator grant. His research interests revolve around the central topic of "Performance-centric System Design" and include scalable networks, parallel programming techniques, and performance modeling. Additional information about Torsten can be found on his homepage at http://htor.inf.ethz.ch.