Causes and Effects of Unanticipated Numerical Deviations in Neural Network Inference Frameworks

Alexander Schlögl **Nora Hofer** **Rainer Böhme**

Department of Computer Science
Universität Innsbruck, 6020 Innsbruck, Austria

{alexander.schloegl | nora.hofer | rainer.boehme} @ uibk.ac.at

Abstract

Hardware-specific optimizations in machine learning (ML) frameworks can cause numerical deviations of inference results. Quite surprisingly, despite using a fixed trained model and fixed input data, inference results are not consistent across platforms, and sometimes not even deterministic on the same platform. We study the causes of these numerical deviations for convolutional neural networks (CNN) on realistic end-to-end inference pipelines and in isolated experiments. Results from 75 distinct platforms suggest that the main causes of deviations on CPUs are differences in SIMD use, and the selection of convolution algorithms at runtime on GPUs. We link the causes and propagation effects to properties of the ML model and evaluate potential mitigations. We make our research code publicly available.

1 Introduction

The "reproducibility crisis" machine learning is facing, and potentially fueling [18], has drawn attention to efforts that improve reproducibility. They include checklists [27] and guidelines [5; 33], benchmarks designed with reproducibility in mind [10; 4; 22], reproducibility contests [32], and repositories like PapersWithCode [30]. However, we find that even with a fully defined environment and without stochastic processes, runtime optimizations of ML frameworks can cause deviations in inference results. These are outside of researchers' control, and cannot be fully avoided at present.

Uncontrolled numerical deviations are detrimental to many aspects of ML. If deviations occur systematically, key assumptions in federated learning [7; 25], heterogeneous ML [36], and proof-of-learning [11] may not hold. The fact that platforms leave fingerprints in the inference results opens new possibilities for forensics [34]. Finally, numerical deviations have implications on ML security: specifically crafted "boundary samples" may trigger label flips depending on the hardware [35].

To study the causes and effects of these deviations, we instrument the popular TensorFlow framework (version 2.5.0) on various layers. All our experiments are containerized and automatically deployed and executed on 75 distinct hardware configurations hosted on the Google Cloud Platform (GCP), Amazon Web Services (AWS), and on our premises. (Details of our setup are in Section B of the supplementary material.) For a fixed trained model and input, the 75 platforms produced up to 26 different softmax vectors in the last layer. As label flips remain rare, the existence of deviations is not apparent in typical performance metrics, such as test set accuracy.

We make three contributions:

1. We offer the so-far most comprehensive evaluation of causes and effects of (known) numerical deviations in CNN inference, spanning a wide range of heterogeneous platforms.
2. We are the first to associate causes of deviations with properties under control of the ML engineer, such as floating-point precision, layer type, or activation function.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

3. We make the code of our infrastructure[1] and experiments[2] publicly available, allowing follow-up researchers to measure deviations between runs and platforms and inspect them layer by layer. The set of supported platforms can be adjusted with limited effort.

The body of this paper is structured as follows. Section 2 provides background, establishes terminology, and reviews related work. Section 3 presents the experimental results. Starting from end-to-end inference pipelines, we walk through individual causes for CPUs (Section 3.1), GPUs (Section 3.2), link them to properties of the ML model (Section 3.3), and study potential mitigations (Section 3.4), always supported with experiments. Section 4 discusses the findings and Section 5 concludes.

2 Background

The main causes for deviations in inference results are different aggregation orders at finite precision, and the use of different approximate convolution algorithms.

Aggregation order The order in which arithmetic operations are executed can affect the result for limited precision. Consider an example in a toy decimal floating point representation with *only one* significant digit. In this representation, integers $0 \leq |x| \leq 10$ can be represented exactly. For values $10 < |y| \leq 100$, y must be rounded to the next multiple of 10, and the least significant digit is lost. We will denote rounding to the nearest representable value with $[x]$.

$$(a + b) + c = [[7 + 4] - 5] = [1\text{E}1 - 5] = 5 \tag{1}$$
$$a + (b + c) = [7 + [4 - 5]] = [7 - 1] = 6 \tag{2}$$

The above example shows how the aggregation order can change the result. In Equation (1), $[7{+}4] = [11]$ is rounded to $1\text{E}1 = 10$. This effect is known as *swamping* [13]. Hence, optimizations that change the aggregation order depending on the hardware can cause deviations between platforms.

Convolution algorithms Recall the formula for 2D convolution,

$$R_{i,j} = \sum_{n=0}^{h} \sum_{m=0}^{w} I_{i-m,j-n} F_{m,n}, \tag{3}$$

for a 2-dimensional filter of size $h \times w$ with input I and filter F. To reduce the size of the convolution result, sometimes a stride is applied to the convolution, which is multiplied with input indices i, j.

The naive implementation of convolution is a nested loop expressing the nested sums. This approach is often inefficient as spatial proximity in the input does not imply locality in memory, in particular for higher dimensions. Modern hardware uses multiple layers of caching, which depend on locality and coalesced access for maximum performance. Due to the prevalence of convolution, especially in ML, many optimized implementations are available [2; 3].

All optimizations reduce convolution to generalized matrix multiplication (GEMM) [6]. GEMM convolution, the simplest optimization, extracts the relevant parts of the array into a Toeplitz matrix, replicates the filter as needed, and multiplies the two matrices. GEMM convolution exists in precomputed, implicit, and explicit variants [3]. By contrast, *Winograd* short convolution [9] transforms both inputs and filters to achieve a lower number of multiplications. There exist fused and non-fused variants [40]. As the transformations used for Winograd convolution are inherently lossy, and rounding occurs at multiple stages, results may deviate. Convolutions using fast Fourier transformation (FFT) exploit the fact that spatial convolution is equivalent to point-wise multiplication in the frequency domain. The transformation to and from the frequency domain is inherently lossy for finite numerical precision [24] and introduces deviations from other methods. In addition, modern GPUs include special TensorCores to compute convolution directly [28].

A more detailed description of convolution approaches can be found in Section C of the supplementary material. Performance characteristics of the different approaches are excellently explained in Blahut et al. [6]. Specific performance characteristics for NVidia GPUs are discussed in [29].

[1] `https://github.com/uibk-iNNference/iNNfrastructure`

[2] `https://github.com/uibk-iNNference/unaNNticipated`

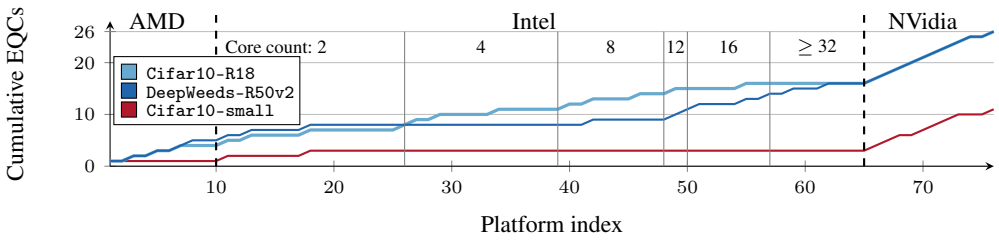


Figure 1: Main result: cumulative number of different softmax vectors in the last layer over all platforms. Platforms are sorted by vendor, architecture, and core count. Core count for Intel CPUs.

Conventions We use *architecture* as shorthand for microarchitecture. Architecture, core count, and memory size together form a *platform*. We call inference *deterministic* if the same input, the same trained model, and the same platform always produce the same output. We say it is *consistent* if the same input and model always produce the same output across all platforms. We denote our models as `Dataset-Architecture`, e.g, `Cifar10-R18` is a ResNet18 [15] trained on Cifar-10 [21].

To measure the effect of numerical deviations, we use the concept of *equivalence classes* (EQCs): platforms that produce identical outputs form an EQC. The number of EQCs is a measure of diversity. Ideally, all platforms fall into a single EQC, meaning that inference is deterministic and consistent (as expected in theory). To quantify the magnitude of deviations, we use the *remaining precision*. This metric counts the number of identical mantissa bits before the first deviation between two or more (intermediate) results. For single precision floating-point numbers, it is in the range $[0, 23]$. The metric generalizes to tensors by taking the minimum of the remaining precision of all elements.

Related work Much work has been done to improve reproducibility in ML [33; 5; 10; 4; 22]. These works make important contributions to the organizational aspects of reproducibility, but do not address the problem of numerical deviations. The literature has examined the influence of variance in algorithm and implementation on model *training*. Pham et al. [31] find that variance can lead to significant differences in model performance and training time. Zhuang et al. [41] extend the experiments and find differences in training performance across different GPUs and TPUs. This end-to-end approach is beneficial for the community and informs practitioners of the potential impact of variances in algorithm and implementation on the final model. Our work differentiates itself by focusing on *inference* and by drilling down to the root causes of the observed deviations.

Deviations in ML inference have been reported in the signal processing community by Schlögl et al. [34], however with a focus on forensics. Our earlier work offers existential evidence from a few CPU platforms, but does not investigate causes or mitigation strategies. GPUs are also not considered.

The computer arithmetic community is well aware of the non-associativity of floating point computations [26; 19], but aims to increase precision and efficiency rather than enforcing associativity.

3 Influences on model outputs

In total, 64 of our 75 platforms were CPU-based. Depending on the model, the softmax vector of the last layer produced between 3 and 16 different EQCs. This means all tested models failed to produce consistent outputs across the CPU platforms. We did not observe indeterminism on CPUs. The remaining 11 of our 75 platforms supported inference on GPUs. The softmax outputs produced between 8 and 10 different EQCs. Every EQC had cardinality one, meaning the GPU could be uniquely identified by the numerical deviations in the softmax vector. 39 out of 99 inference outputs on GPUs were indeterministic. Figure 1 shows the cumulative number of EQCs over all platforms in this scenario (see Tables SUP-1 and SUP-2 for all hardware, model, and input details).

While the main results above are end-to-end measurements for realistic inference pipelines, it is instructive to study individual causes with specifically designed experiment on a reduced set of platforms. All following results are supported with isolated experiments. We discuss the influences affecting model outputs on CPUs and GPUs in turn.

Table 1: Different CPUs produce deviations based on their SIMD capabilities.

		Flag cluster													
	EQC	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Intel Sandy Bridge	1					×										×
Intel Ivy Bridge	1					×					×	×				×
Intel Haswell	2	×∗				×					×	×		×		×
Intel Broadwell	2	×				×			×		×	×	×	×		×
Intel Skylake	3	×	×†			×	×		×		×	×	×	×	×	×
Intel Cascade Lake	3	×	×			×	×		×	×•	×	×	×	×		
Intel Ice Lake	4	×	×	×‡		×	×	×	×	×	×	×	×	×		
AMD Rome	5	×			×§		×	×	×			×				
AMD Milan	5	×			×		×	×	×		×	×		×		

∗ contains 256-bit SIMD flags; † contains some 512-bit SIMD flags; ‡ contains more 512-bit SIMD flags; § contains `sse4a` (128-bit SIMD) and `misalignsse` flags; • contains the `avx512vnni` flag

3.1 Influences when inferring on CPUs

Model outputs computed on CPUs may deviate because of differences in data and task parallelism. Both affect the aggregation order of convolutions.

Data parallelism Modern architectures feature a variety of SIMD instructions, which affect both the floating-point accuracy and the aggregation order. While CPUs traditionally compute floating-point operations at very high precision in the FPU, the introduction of SIMD instructions, such as `SSE` and `AVX` on x86, enables data parallelism at the cost of reduced precision. Newer CPUs have larger SIMD registers, which allow more flexibility when adjusting the SIMD width (i. e., data parallelism) and desired precision. Some CPUs support fused multiply-and-add in SIMD [16; 1].

To measure the effect of data parallelism on CPU, we perform inference with the `Cifar10-R18` model on all dual-core x86 systems available on GCP, and collect all CPUID flags indicating hardware support of SIMD instructions. Since the number of flags (169) exceeds the number of platfoms, we cluster flags that always co-occur. Table 1 shows the relation between EQCs and flag clusters.

Observe that a subset of the flag clusters perfectly aligns with the EQCs. Flag clusters 0–3 are all related to the support for different SIMD capabilities. Flag cluster 0 contains the `avx2` flag, denoting 256-bit SIMD support. Flag cluster 1 contains the `avx512f` flag, indicating an SIMD width of 512 bit. Flag cluster 2 contains four SIMD-related flags: `avx512vbmi` (vector bit manipulation instructions), `avx512ifma` (fused multiply-add for integers), `vpclmulqdq` (carry-less multiplication), and `avx512vpopcntdq` (population count). While the co-occurrence prevents us from attributing the deviation to a single flag, we consider this evidence to show that some SIMD-related feature is responsible for the EQC split. Flag cluster 3 contains the `sse4a` flag not present in the other CPUs, as well as the `misalignsse` flag to indicate support for misaligned memory access when using legacy SIMD instructions. For this EQC, however, the effect of different SIMD support may be superseded by general architectural differences of AMD processors. Interestingly, no new EQC emerges based on the support for `avx512vnni` vector neural-network instructions, indicating that the framework does not use this hardware feature yet.

Task parallelism Divide-and-conquer algorithms can increase performance and reduce memory overhead by distributing work to multiple cores [38]. The use of task parallelism involves additional aggregation steps, the order of which can vary depending on the implementation, e. g., atomic versus sum–reduce. As work gets distributed on more cores, the individual workload decreases. When the workload per core becomes too small, data parallelism cannot be used anymore. Figure 2 visualizes this effect. When increasing core count from 2 to 4, the additional 512-bit SIMD capabilities of the Intel Ice Lake platform become unusable, producing the same outputs as Intel Sky-/Cascade Lake. Going from 8 to 16 cores makes the per-core workload too small for 256-bit SIMD instructions, resulting in a single EQC for all Intel platforms newer than Ivy/Sandy Bridge. We did not observe additional EQC collapses for core counts larger than 16. Both the Intel Sandy/Ivy Bridge

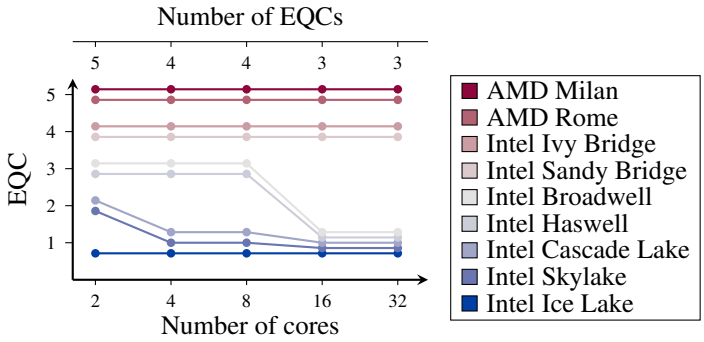


Figure 2: Deviations disappear as the number of CPU cores increases.

and the AMD Rome/Milan architectures have sufficiently different SIMD capabilities (cf. Table 1) to produce deviations even for larger core counts.

3.2 Influences when inferring on GPUs

The choice of convolution algorithm is a cause of deviations on GPUs. Due to different performance characteristics, there is no universally best convolution algorithm [6]. Modern GPUs implement a large number of convolution algorithms, and select the fastest algorithm for the convolution parameters (number of filters, stride, etc.) at runtime based on microbenchmarks.[3] Therefore, the final algorithm choice may not only vary with the GPU and model architecture, but also with the remaining hardware and even uncontrollable conditions, like bus contention and parallel load. As each convolution is benchmarked separately, different layers of a model can use different algorithms.

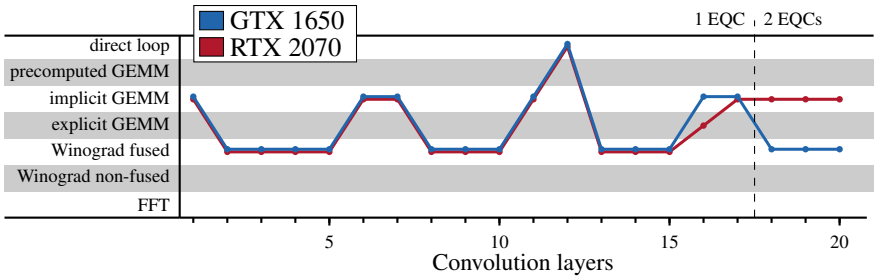
Deviations between GPUs We perform inference with the `Cifar10-R18` model on all GPUs available on GCP and locally. Using the TensorFlow profiler, we log all function calls and extract the convolution algorithms. Internally, the TensorFlow profiler uses the NVidia profiler to record calls to the CUDA API and some CUDA-internal function calls. A list of all observed convolution algorithms can be found in Table SUP-4 in the supplementary material.

For the GPUs with deterministic outputs, we can explain the deviations by the use of different approximate convolution algorithms. Figure 3a shows exemplary traces from the GTX 1650 and RTX 2070 GPUs on our local machines. While 16 different convolution functions were called in our experiments, we group them by their approach for ease of reading. The lines in Figure 3a plot the selected algorithm for each convolutional layer of the model. In this case, both GPUs choose the same algorithm for each of the first 15 convolutions. Convolution 16 is computed with different algorithms, but they happen to produce identical results. This is because the only difference between explicit and implicit GEMM is the fact that explicit stores the Toeplitz matrix in memory, whereas implicit GEMM computes it on the fly. The deviations are caused by the final three convolutions, where the RTX 2070 continues to use implicit GEMM while the GTX 1650 switches back to Winograd. The separation of EQCs is indicated by the dashed vertical line, which we verify by comparing the intermediate results. We confirm from all other traces that this pattern is typical for deterministic deviations between GPUs.

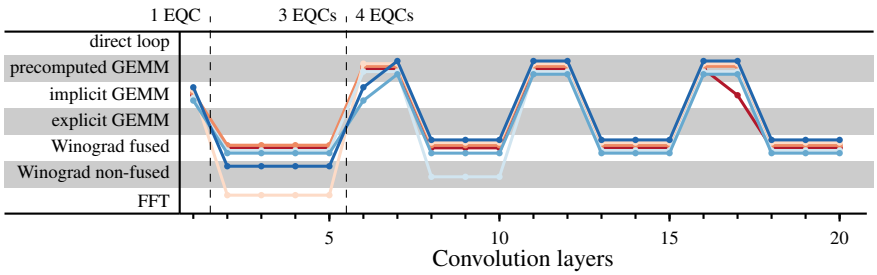
Deviations between inferences on the same GPU Microbenchmarks are run just before the first inference. The framework uses random data to fill a buffer of the problem dimensions and measures the execution time of all supported convolution algorithms. Since these measurements take up valuable execution time, each candidate algorithm is timed only once. This makes the choice of the "winner" susceptible to runtime conditions, e. g., bus contention and memory latency.

To measure variations between inferences, we perform repeated inference with the `Cifar10-R18` model. Each inference happens in a new session. We plot the traces obtained with the same profiler instrumentation. Figure 3b shows the behavior of an NVidia P100 GPU. For 33 inferences on the same model and data, we observe 6 different paths resulting in 4 EQCs. Observe that up to three

[3] This approach is also called "auto-tuning" in the literature, e. g., [14; 31].



(a) Different GPUs choose different algorithms in certain layers. All GEMM variants produce identical outputs.



(b) The same GPU chooses different algorithms depending on variance in the microbenchmarks.

Figure 3: Choice of convolution algorithm per layer. Function calls are grouped by approach.

different approaches are used to compute the same convolution (layers 2–5). The leftmost dashed line provides evidence that the intermediate results differ at that layer and propagate further. Another separation happens at layer 6, whereas the choice of algorithm at layer 17 does not fork out another EQC. We confirm from all other traces that this pattern is typical for indeterministic deviations between GPUs.

The host system of the GPU can also influence the race and thus the number of unique algorithm sequences. The red trace in Figure 3a shows one of two unique algorithm sequences for an RTX 2070, which both lead to one deterministic output. For 33 inferences on the same model and data, we deterministically observe two EQCs, one for each GPU. We find that the same GPU can produce different algorithm sequences, resulting in a different number of EQCs on other host systems.

3.3 Model-specific influences

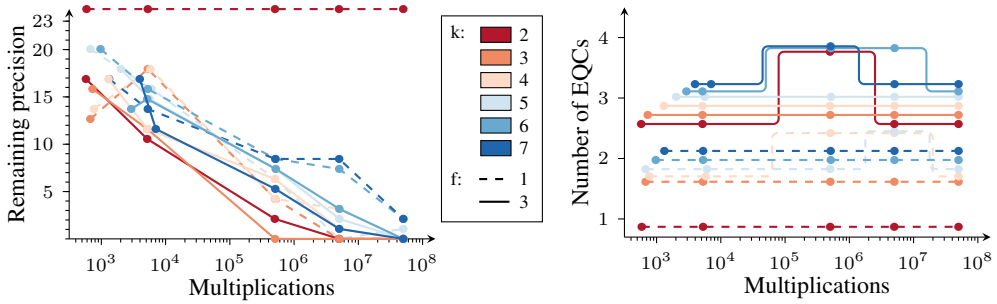
The architecture, parameters, and input dimensions of an ML model can influence the number and nature of deviations by amplifying or suppressing influences.

Number of multiplications/parameters Numerical deviations are caused by arithmetic operations, therefore more operations mean potentially more deviations. Both larger inputs and a higher number of parameters increase the number of computations during inference. Moreover, the problem dimensions impact the performance of convolution algorithms and affect the algorithm choice.

As deviations arise from rounding errors and different aggregation orders, we hypothesize that more multiplications per convolution layer lead to more deviations. We investigate this with an ablation study for the parameters of standard 2-dimensional convolution: kernel size k, number of filters f, and input dimensions $i \times j \times c$. Our experiment covers several orders of magnitude in the number of multiplications, ranging from 10^3 to 10^8, which is more than the number of multiplications in our `Cifar10-R18` model. For each order of magnitude, we fix k, f and c, and adjust i and j to reach the desired number of multiplications while aiming for square inputs, i. e., $i \approx j$. Figure 4 shows $c = 3$, but other values give the same results. Weights are drawn from a uniform distribution over $[0, 1)$.[4]

Our remaining precision metric (cf. Section 2) measures the magnitude of the deviations. Figure 4a shows that the remaining precision decreases with the number of multiplications and does not vary

[4]This does not affect results, as discussed in the "Parameter and input distribution" paragraph of Section 3.5.



(a) Remaining precision over # of multiplications.

(b) Number of EQCs for the same platforms.

Figure 4: Effect of different convolution dimensions (mapped to the number of multiplications).

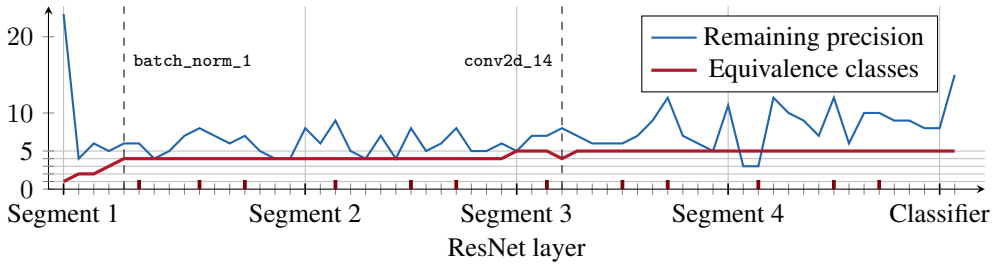


Figure 5: Causes and effects by layer: number of EQCs and remaining precision for `Cifar10-R18`.

significantly between the platforms (excluding the smallest case $k = 2, f = 1$, where no deviations emerge). This is plausible, as approximately Gaussian deviations lead to a lower minimum remaining precision if more independent realizations are aggregated. The remaining precision drops to 0, which means that at least one deviation is at least as large as factor 2.

Figure 4b shows the number of equivalence classes as a function of the number of multiplications. Observe that the relation is not monotonic. Both $f = 1$ and $f = 3$ produce more EQCs for 10^6 to 10^7 multiplications than for 10^8. The number of filters seems to have a larger influence than the filter size. Note that the maximum number of EQCs in this experiment is four, whereas we observe up to five EQCs for the same set of platforms in our main experiment with complete models. We interpret this as evidence that deviations propagate between layers and may produce forks into different EQCs in later layers.

Architecture and layer types The model architecture defines the operations and hence determines if a cause of deviation is present or not. In particular, convolutional and other parallel data processing layers tend to introduce deviations. Layers that aggregate results and reduce information, such as pooling layers, may reduce or eliminate deviations from preceding layers. This extends to activation functions. Continuous functions that preserve information (e. g., sigmoid and softmax) can maintain or potentially amplify deviations. Functions that reduce information, like the rectified linear unit (ReLU), can have a diminishing effect on deviations. This means that a sufficiently large model can introduce deviations in earlier layers, and remove them in pooling and activation layers. Skip layers can preserve deviations by bypassing information reduction.

To investigate the influence of different layer types in a full inference computation, we instrument a model to output the intermediate results of all layers in addition to the final class label. We verify that this does not alter the EQCs. Figure 5 plots the number of EQCs as well as the remaining precision over the outputs of the 60 layers of `Cifar10-R18`. Note that activations count as distinct layers, marked with a thick dark-red tick, and not every convolution is immediately followed by an activation. The annotated segments on the x-axis refer to the ResNet convention of bundling convolutions of the same size into segments [15].

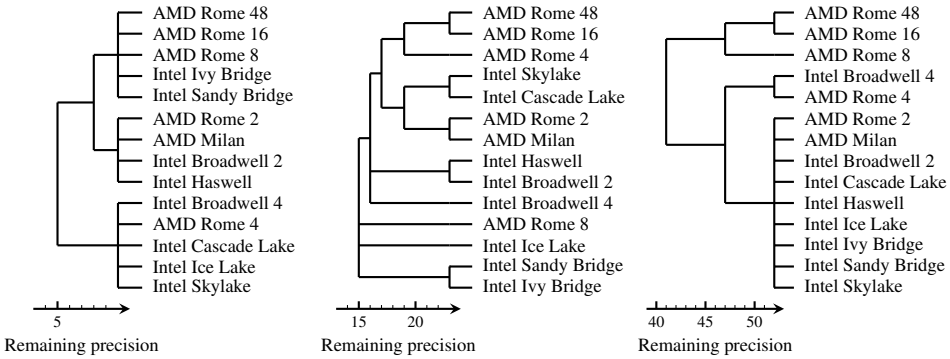
The remaining precision changes after every layer. This can be explained by the fact that the very first convolution following the input layer already produces deviations that result in multiple EQCs.

As long as more than one EQC exists (i. e., for all remaining layers), the remaining precision indicates how far these EQCs fall apart in the most extreme case. In terms of magnitude, the remaining precision never reaches zero. This corroborates the results of our ablation study (cf. Figure 4a) as the maximum number of multiplications per layer is in the order of 10^6. Although the number of total multiplications increases monotonically, the ReLU activations increase the remaining precision, visible as peaks of the remaining precision. One instance of aggregation is very pronounced at the rightmost layer, where the output is projected to a small label space.

Turning to the number of EQCs, new ones only emerge as outputs of convolutional and batch normalization layers. The number of EQCs tends to increase over the execution of the model. However, we also observe a reduction in the number of EQCs, which occurs once in a convolutional layer (`conv2d_14`) in the third segment of the model. We did not expect to sees this, in particular given that this layer does not perform any quantization or aggregation. This shows that past deviations may be cancelled out. Another unexpected finding is the emergence of the fourth EQC, which occurs in a batch normalization layer (`batch_norm_1`), and deviations arise in layers other than convolution.

3.4 Influence of floating point precision

Figure 6 shows how casting a trained model to a different floating-point precision affects deviations. The figure shows similarities between the EQCs as dendrograms, with remaining precision used as distance metric. Branch lengths within the dendrograms are proportional to the remaining precision between the EQCs, and connections further to the right indicate higher remaining precision. Branches extend past the maximum remaining precision of the format to indicate platforms that produce identical outputs and thus are in the same EQC.



(a) Half precision: 3 EQCs. (b) Single precision: 9 EQCs. (c) Double precision: 4 EQCs.

Figure 6: Influence of casting the `Cifar10-R18` model to different floating-point precisions. Both half (float16) and double precision (float64) floating-point numbers generate fewer deviations than single precision. The same pattern holds for all models, cf. Section E in the supplementary material.

Rounding to IEEE-754 16-bit half-precision (`float16`) suppresses deviations and reduces the number of EQCs. However, which EQCs fall together seems rather chaotic in our experiments. For example, AMD Rome processors fall into all three EQCs, depending on their core count. Reducing precision thus mitigates deviations, but in potentially unpredictable ways. Alternatively, the model can be converted to fixed point integer arithmetic, which reportedly eliminates all deviations [34].

Interestingly, increasing precision also reduces the number of EQCs. While we observe a total of four different EQCs when inferring with our model in 64-bit double precision, this is still fewer than for 32-bit single precision. As we simply cast the model to the higher precision without modifying the weights, the lower part of the 64-bit mantissas are zero. This provides more room for shifting and prevents swamping to some extent. Platforms from different EQCs for 32 bit match for 64 bit according to their core count, indicating that deviations due to task parallelism persist. We conjecture that zeroing the last n mantissa bits has the same effect, reducing the performance cost of the mitigation measure at the cost of reduced fidelity.

3.5 Potential influences not observed in experiments

Our research identified a number of potential influences that did not surface in isolated experiments.

Compute graph optimization Recent approaches go beyond optimized algorithms and instead transform the operations and their ordering (the compute graph) [23]. For example, the Accelerated Linear Algebra (XLA) project is a JIT compiler for ML operations [39]. The resulting compute graph is specific to the hardware [34]. It determines if and how deviations caused in the hardware come to effect. XLA is not yet used by default [39]. As we did not activate it, we can rule out special compute-graph optimizations as cause for deviations.

Device placement Most ML frameworks allow running operations on both CPU and GPU. In theory this lets them choose the target device at runtime. Under certain conditions, the framework may decide to execute operations on the CPU even though a GPU is available [37]. Since CPU and GPU implementations very likely differ in algorithm, floating-point precision, and aggregation order, the device placement can cause deviations. We log the device placement in our experiments and verify that EQCs do not change with our instrumentation.

Race conditions Task-parallel execution can lead to different aggregation orders depending on the implementation. If all intermediate results are stored in an array and then summed up (e. g., with sum–reduce), the execution time of individual tasks will not change the aggregation order. However, if the results are summed up immediately, with some sort of locking mechanism, the execution time of individual tasks determines the aggregation order. In our experiments, all model outputs on CPUs and GPUs are consistent and deterministic when controlling for the influences we discovered, ruling out race conditions as causes of deviations.

Parameter and input distributions The distributions of model parameters and input values can both affect the shape and magnitude of deviations. To measure their influence, we first extract the `conv2d_11` layer from our `Cifar10-R18` model and capture its intermediate input during a forward pass. Then, we perform inference on twelve combinations of input and parameter distributions on all dual-core GCP instances. We use the original weights, weights drawn from the Glorot uniform distribution [12], and weighs drawn from a Gaussian with $\mu = -0.019, \sigma = 0.226$, fitted to the trained weights. For the inputs we use the captured values, two random permutations of these values (preserving the marginal distribution), and random inputs drawn uniformly from $[0, 1)$. The EQCs for all cases are identical. This suggest that the emergence of EQCs is independent of the parameter and input distribution. Incidentally, the microbenchmarks for algorithm choice on GPUs make a similar assumption. Their inputs are filled with random data.

4 Discussion

Impact The deviations we observe might impair ML reproducibility and security. Concerning reproducibility, as different platforms can produce different outputs from the exact same model and input data, it is unlikely that high-precision numerical results can be reproduced exactly. Assertions in automated software tests are prone to fail on specific hardware. Such limitations to reproducibility are not always visible in headline indicators because classifier performance is typically measured on the level of labels. The deviations we study rarely cause label flips for natural inputs. However, we have demonstrated in [35] that label flips can be provoked by searching for synthetic inputs that map to the space between decision boundaries of different platforms. Moreover, Casacuberta et al. [8] point out that numerical deviations can undermine security guarantees of ML implementations with differential privacy. Differences between symbolic math and actual floating-point computations at limited precision have been used to fool ML verifiers [42; 17]. Varying algorithm choices break these assumptions even further, as they fundamentally change the type and order of arithmetic operations. These examples highlight the security impact of our observation. Our study of causes breaks ground for a principled approach to assess and mitigate these risks.

Federated learning distributes ML training tasks over many machines under the assumption of compatible gradients. The systematic deviations observed here call for robustness checks when combining gradients from heterogeneous hardware [7; 25]. The fact that deviations leave traces of the executing hardware enables forensicability and attribution of ML outputs [34]. This could be used,

e. g., in the combat against harmful generated content; but can also be misused, e. g., for unauthorized surveillance. This adds a new aspect to the debate on societal and ethical aspects of ML.

Mitigation strategies As shown in Figure 6, quantization can suppress deviations and reduce the number of EQCs. However, these measures do not mitigate deviations due to algorithm choice, and thus are not applicable to GPUs. While researchers can instruct the ML framework to use deterministic implementations of operators,[5] the fact that different GPUs support different algorithms can lead to deviations even when such options are set. Similarly, TensorCores [28] *may* be used if available, which can also lead to deviations. While they can be disabled, their use cannot be enforced. Reaching full reproducibility requires giving researchers access to low-level functionalities and providing them with fine-grained control over optimizations. Another approach is to sidestep platform-specific optimizations at runtime by transforming the model into a flat compute graph. TFLite, for example, does this by default; yet for the purpose of a smaller memory footprint rather than consistency.

While all these strategies are heuristic, developers of future ML frameworks and accelerator libraries should consider supporting an option for fully deterministic and consistent computation. This comes almost unavoidably at the cost of lower performance. Cryptographic libraries, for example, went a similar path when constant-time options were added in response to the discovery of side-channel attacks [20]. In both domains, successful mitigation depends on exact knowledge of the entire stack down to the hardware.

Limitations While our work covers a lot of ground concerning numerical deviations, there are still some areas for further exploration. Our experiments have intentionally kept some factors constant: the versions of TensorFlow (2.5.0), all dependent libraries and drivers, the compiler version and options used to build TensorFlow, as well as the concurrent load on the system besides our experiments. Additional layer types and other frameworks can also be subject to follow-up work.

5 Conclusion

Performing inference on the same trained model and input data is not always consistent across platforms, and sometimes not even deterministic on the same platform. This paper explores the causes for the numerical deviations, in particular in convolution operations. They include the choice of the specific convolution algorithm, the floating-point precision, and the order of aggregation. The order of aggregation is defined by SIMD capabilities and the number of cores of the executing CPU. As the number of CPU cores grows, task parallelism can supersede the deviations caused by SIMD capabilities. GPUs use microbenchmarks to select the fastet supported convolution algorithm at runtime. We find that this can produce seemingly indeterministic behavior when different algorithms are chosen in different sessions. We validate our findings on 75 platforms, including CPUs and GPUs, hosted locally and on two large commercial cloud platforms. Our measurement infrastructure, which is made available on GitHub,[6] can facilitate future comparative studies. Our analysis offers a number of mitigation strategies, but certain deviations appear unavoidable at present. These findings imply that authors interested in improving reproducibility should meticulously document specifics of the computation hardware, in addition to the amount of compute [27], and provide high-precision intermediate results as reference points for replication attempts.

Acknowledgments and Disclosure of Funding

We thank Matthias Kramm for insightful comments on earlier versions of this paper. We also thank Michael Fröwis, Jakob Hollenstein, Martin Beneš, and Patrik Keller for fruitful discussions during this research. Parts of this work were funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101021687, and a PhD Fellowship from NEC Laboratories Europe (2021–2022).

[5] For example, by setting the `enable_op_determinism` flag, TensorFlow chooses the first available convolution algorithm rather than the fastest one.

[6] `https://github.com/uibk-iNNference/`

References

[1] AMD. 128 bit and 256 bit VPX instructions. `http://support.amd.com/TechDocs/43479.pdf`, 2009.

[2] Andrew Anderson and David Gregg. Optimal DNN primitive selection with partitioned boolean quadratic programming. In *International Symposium on Code Generation and Optimization (GCO)*, pages 340–351. ACM, 2018.

[3] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. High-performance low-memory lowering: GEMM-based algorithms for DNN convolution. In *Interantional Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 99–106. IEEE, 2020.

[4] Sebastian Pineda Arango, Hadi Samer Jomaa, Martin Wistuba, and Josif Grabocka. HPO-B: A large-scale reproducible benchmark for black-box HPO based on OpenML. In *Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2021.

[5] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and don'ts of machine learning in computer security. In *USENIX Security Symposium (USENIX Security)*, pages 3971–3988. USENIX Association, 2022.

[6] Richard E. Blahut. *Fast Algorithms for Signal Processing*. Cambridge University Press, 2010.

[7] Keith Bonawitz et al. Towards federated learning at scale: System design. In *Machine Learning and Systems (MLSys)*. mlsys.org, 2019.

[8] Sílvia Casacuberta, Michael Shoemate, Salil P. Vadhan, and Connor Wagaman. Widespread underestimation of sensitivity in differentially private libraries and how to fix it. In *Conference on Computer and Communications Security (CCS)*, pages 471–484. ACM, 2022.

[9] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[10] Katharina Eggensperger, Philipp Müller, Neeratyoy Mallik, Matthias Feurer, Rene Sass, Aaron Klein, Noor Awad, Marius Lindauer, and Frank Hutter. HPOBench: A collection of reproducible multi-fidelity benchmark problems for hpo. In *Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2021.

[11] Congyu Fang, Hengrui Jia, Anvith Thudi, Mohammad Yaghini, Christopher Choquette-Choo, Natalie Dullerud, Varun Chandrasekaran, and Nicolas Papernot. Proof-of-learning is currently more broken than you think. In *European Symposium on Security and Privacy (EuroS&P)*, pages 797–816. IEEE, 2023.

[12] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 249–256. JMLR, 2010.

[13] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[14] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar)*, 2012.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778. IEEE, 2016.

[16] Intel. Intel® C++ compiler classic developer guide and reference. `https://www.intel.com/content/dam/develop/external/us/en/documents/cpp_compiler_classic.pdf`, 2021.

[17] Kai Jia and Martin Rinard. Exploiting verified neural networks via floating point numerical error. In *International Symposium on Static Analysis (SAS)*, pages 191–205. Springer, 2021.

[18] Sayash Kapoor and Arvind Narayanan. Leakage and the reproducibility crisis in ML-based science. *arXiv*, 2022. arXiv Computing Research Repository (CoRR), abs/2207.07048.

[19] Nachiket Kapre and Andre DeHon. Optimistic parallelization of floating-point accumulation. In *Symposium on Computer Arithmetic (ARITH)*, pages 205–216. IEEE, 2007.

[20] Paul C. Kocher. Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems. In *Advances in Cryptology (CRYPTO)*, pages 104–113. Springer, 1996.

[21] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset. Master's thesis, University of Toronto, 2014.

[22] Yuhang Li, Mingzhu Shen, Jian Ma, Yan Ren, Mingxin Zhao, Qi Zhang, Ruihao Gong, Fengwei Yu, and Junjie Yan. Mqbench: Towards reproducible and deployable model quantization benchmark. In *Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2021.

[23] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing CNN model inference on CPUs. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 1025–1040. USENIX Association, 2019.

[24] Raimund Meyer. Error analysis and comparison of FFT implementation structures. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 888–891. IEEE, 1989.

[25] Fan Mo, Hamed Haddadi, Kleomenis Katevas, Eduard Marin, Diego Perino, and Nicolas Kourtellis. PPFL: Privacy-preserving federated learning with trusted execution environments. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 94–108. ACM, 2021.

[26] Jean-Michel Muller. $a \cdot (x \cdot x)$ or $(a \cdot x) \cdot x$? In *Symposium on Computer Arithmetic (ARITH)*, pages 17–24. IEEE, 2021.

[27] NeurIPS. NeurIPS 2021 paper checklist. `https://neurips.cc/Conferences/2021/PaperInformation/PaperChecklist`, 2021.

[28] NVidia Corporation. NVidia TensorCores. `https://www.nvidia.com/en-us/data-center/tensor-cores/`, 2022.

[29] NVidia Corporation. Optimizing convolutional layers in deep learning. `https://docs.nvidia.com/deeplearning/performance/pdf/Optimizing-Convolutional-Layers-User-Guide.pdf`, 2023.

[30] PapersWithCode. Papers with code. `https://paperswithcode.com/`, 2018.

[31] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. Problems and opportunities in training deep learning software systems: An analysis of variance. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 771–783. IEEE, 2020.

[32] Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché Buc, Emily Fox, and Hugo Larochelle. Improving reproducibility in machine learning research (a report from the NeurIPS 2019 reproducibility program). *Journal of Machine Learning Research*, 22(1):7459–7478, 2021.

[33] Edward Raff. A step toward quantifying independently reproducible machine learning research. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[34] Alexander Schlögl, Tobias Kupek, and Rainer Böhme. Forensicability of deep neural network inference pipelines. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 2515–2519. IEEE, 2021.

[35] Alexander Schlögl, Tobias Kupek, and Rainer Böhme. iNNformant: Boundary samples as telltale watermarks. In *Workshop on Information Hiding and Multimedia Security (IH&MMSec)*, pages 81–86. ACM, 2021.

[36] Surat Teerapittayanon and Bradley McDanel. Branchynet: Fast inference via early exiting from deep neural networks. In *International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[37] TensorFlow. TensorFlow GPU guide. `https://www.tensorflow.org/guide/gpu`, 2021.

[38] Aravind Vasudevan, Andrew Anderson, and David Gregg. Parallel multi channel convolution using general matrix multiplication. In *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 19–24. IEEE, 2017.

[39] XLA Developer Team. XLA: TensorFlow, compiled. `https://www.tensorflow.org/xla#enable_xla_for_tensorflow_models`, 2022. Accessed on 2022-07-04.

[40] Da Yan, Wei Wang, and Xiaowen Chu. Optimizing batched winograd convolution on gpus. In *SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 32–44. ACM, 2020.

[41] Donglin Zhuang, Xingyao Zhang, Shuaiwen Song, and Sara Hooker. Randomness in neural network training: Characterizing the impact of tooling. *Proceedings of Machine Learning and Systems*, 4:316–336, 2022.

[42] Dániel Zombori, Balázs Bánhelyi, Tibor Csendes, István Megyeri, and Márk Jelasity. Fooling a complete neural network verifier. In *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2021.

Supplementary Material

Causes and Effects of Unanticipated Numerical Deviations in Neural Network Inference Frameworks

Alexander Schlögl **Nora Hofer** **Rainer Böhme**

Department of Computer Science
Universität Innsbruck, 6020 Innsbruck, Austria

{alexander.schloegl | nora.hofer | rainer.boehme} @ uibk.ac.at

A Detailed results

Table SUP-1 shows the CPU EQCs in full detail, including information on hardware, model, and input. Table SUP-2 shows the same information for GPUs. This data is the source for Figure 1 in the main paper.

Table SUP-3 lists all observed CPU flags and their corresponding cluster index. Some flags were present on all machines, and were thus filtered from Table 1 in the main paper. These are marked with *C*, for common.

B Methodology

Our experiments require instrumentation at various levels of the ML software stack, shown in Figure SUP-1. The interfaces to the ML framework use different programming languages. Parts of the stack are not accessible for analysis (e. g., microcode on CPUs, vendor libraries for GPUs).

Information extraction We captured as much information about the entire inference pipeline as possible, using TensorFlow's own profiler. It includes underlying tools like NVidia's `nvprof` profiler, and allows us to investigate function calls in the accelerator libraries. Information about the computing devices is taken from `/proc/cpuinfo` and the TensorFlow device information, respectively. For CPUs we fill the microarchitecture field by cross-referencing the `family` and `model` fields of the CPUID. GPUs are uniquely identifiable by their names, including the microarchitecture. Device names are given with as much detail as provided by the machine; due to shared tenancy, device information for cloud CPUs may be reported with less detail. Memory sizes are taken from the `psutil` Python module for CPUs, and from the TensorFlow reported `memory_limit` for GPUs.

Containerization To ensure that the same experiments are run on a large number of cloud instances, we use Docker to fix the software environment and package versions. The Docker image is built locally and uploaded to image registries of both cloud providers used. From there, the image is pulled to the respective target machines and used to run the experiments.

Our experiments run in a Docker container based on the `tensorflow:2.5.0-gpu` image. This image runs Python version 3.6.9, and we additionally install `clang` version 6.0.0. The TensorFlow version is 2.5.0. In this version, XLA is not enabled by default, and was not explicitly activated. The container already handles the GPU setup, and no additional steps are necessary. For the cloud instances with GPUs, we forward them to the container using Docker's `--gpus` flag.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

Table SUP-1: Full results for CPU instances. EQCs are assigned increasing integers from top to bottom. Table cells identical with their left neighbor are slightly faded. CPUs are separated into CPU classes (CCs) based on available x86 extensions and clustered core count. The first occurrence of an EQC per column is marked in bold.

								CIFAR-10						DeepWeeds		
							Model size	Small			Medium			Large		
							Sample index	0	1	6	0	1	6	0	1	6
	Vendor	**Cores**	**Generation**	**Device name**	**Mem.**	**Cloud**	**CC**									
(1)	AMD	2	Milan	7B13	7.8	GCP	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
	AMD	2	Rome	7B12	7.8	GCP	0	1	1	1	1	1	1	1	1	1
	AMD	4	Milan	7B13	15.6	GCP	1	1	1	1	**2**	**2**	**2**	**2**	**2**	**2**
	AMD	4	Rome	7B12	15.6	GCP	1	1	1	1	2	2	2	2	2	2
	AMD	8	Milan	7B13	31.4	GCP	2	1	1	1	**3**	**3**	**3**	**3**	**3**	**3**
(6)	AMD	8	Rome	7B12	31.4	GCP	2	1	1	1	3	3	3	3	3	3
	AMD	16	Milan	7B13	62.8	GCP	3	1	1	1	**4**	**4**	**4**	**4**	**4**	**4**
	AMD	16	Rome	7B12	62.8	GCP	3	1	1	1	4	4	4	4	4	4
	AMD	32	Milan	7B13	125.9	GCP	4	1	1	1	4	4	4	**5**	**5**	**5**
	AMD	32	Rome	7B12	125.9	GCP	4	1	1	1	4	4	4	5	5	5
(11)	Intel	2	Sandy Br.	Xeon	7.3	GCP	5	**2**	**2**	**2**	**5**	**5**	**5**	**6**	**6**	**6**
	Intel	2	Ivy Br.	Xeon	7.3	GCP	5	2	2	2	5	5	5	6	6	6
	Intel	2	Haswell	Xeon	7.3	GCP	6	1	1	1	**6**	**6**	**6**	**7**	**7**	**7**
	Intel	2	Haswell	E5-2676	3.8	AWS	6	1	1	1	6	6	6	7	7	7
	Intel	2	Broadwell	Xeon	7.3	GCP	6	1	1	1	6	6	6	7	7	7
(16)	Intel	2	Broadwell	E5-2686	7.8	AWS	6	1	1	1	6	6	6	7	7	7
	Intel	2	Skylake	8175M	15.3	AWS	7	**3**	**3**	**3**	**7**	**7**	**7**	**8**	**8**	**8**
	Intel	2	Skylake	Xeon	7.8	GCP	7	3	3	3	7	7	7	8	8	8
	Intel	2	Skylake	Xeon	7.3	GCP	7	3	3	3	7	7	7	8	8	8
	Intel	2	Skylake	8175M	7.5	AWS	7	3	3	3	7	7	7	8	8	8
(21)	Intel	2	Skylake	8259CL	7.6	AWS	7	3	3	3	7	7	7	8	8	8
	Intel	2	Skylake	8259CL	15.3	AWS	7	3	3	3	7	7	7	8	8	8
	Intel	2	Skylake	8259CL	7.7	AWS	7	3	3	3	7	7	7	8	8	8
	Intel	2	Skylake	8151	15.3	AWS	7	3	3	3	7	7	7	8	8	8
	Intel	2	Ice Lake	Xeon	7.8	GCP	8	3	3	3	**8**	**8**	**8**	8	8	8
(26)	Intel	4	Sandy Br.	Xeon	14.7	GCP	9	2	2	2	**9**	**9**	**9**	6	6	6
	Intel	4	Ivy Br.	Xeon	14.7	GCP	9	2	2	2	9	9	9	6	6	6
	Intel	4	Haswell	E5-2666	7.3	AWS	10	1	1	1	**10**	**10**	**10**	7	7	7
	Intel	4	Haswell	Xeon	14.7	GCP	10	1	1	1	10	10	10	7	7	7
	Intel	4	Haswell	E5-2676	15.6	AWS	10	1	1	1	10	10	10	7	7	7
(31)	Intel	4	Haswell	E7-8880	119.9	AWS	10	1	1	1	10	10	10	7	7	7
	Intel	4	Broadwell	Xeon	14.7	GCP	10	1	1	1	10	10	10	7	7	7
	Intel	4	Skylake	8124M	7.4	AWS	11	3	3	3	**11**	**11**	**11**	8	8	8
	Intel	4	Skylake	8275CL	7.5	AWS	11	3	3	3	11	11	11	8	8	8
	Intel	4	Skylake	8124M	9.9	AWS	11	3	3	3	11	11	11	8	8	8
(36)	Intel	4	Skylake	Xeon	15.6	GCP	11	3	3	3	11	11	11	8	8	8
	Intel	4	Skylake	Xeon	14.7	GCP	11	3	3	3	11	11	11	8	8	8
	Intel	4	Ice Lake	Xeon	15.6	GCP	12	3	3	3	11	11	11	8	8	8
	Intel	8	Sandy Br.	Xeon	29.4	GCP	13	2	2	2	**12**	**12**	**12**	6	6	6
	Intel	8	Ivy Br.	Xeon	29.4	GCP	13	2	2	2	12	12	12	6	6	6
(41)	Intel	8	Haswell	Xeon	29.4	GCP	14	1	1	1	**13**	**13**	**13**	7	7	7
	Intel	8	Broadwell	Xeon	29.4	GCP	14	1	1	1	13	13	13	7	7	7
	Intel	8	Skylake	Xeon	31.4	GCP	15	3	3	3	**14**	**14**	**14**	8	8	8
	Intel	8	Skylake	Xeon	29.4	GCP	15	3	3	3	14	14	14	8	8	8
	Intel	8	Coffee Lake	i7-9700	31.2	local	16	1	1	1	13	13	13	**9**	**9**	**9**
(46)	Intel	8	Coffee Lake	E3-1270	31.3	local	16	1	1	1	13	13	13	9	9	9
	Intel	8	Ice Lake	Xeon	31.4	GCP	17	3	3	3	14	14	14	8	8	8
	Intel	12	Ivy Br.	i7-4930K	62.8	local	18	2	2	2	**15**	**15**	**15**	**10**	**10**	**10**
	Intel	12	Ivy Br.	i7-4930K	3.8	local	18	2	2	2	15	15	15	10	10	10
	Intel	16	Sandy Br.	Xeon	58.9	GCP	19	2	2	2	15	15	15	**11**	**11**	**11**
(51)	Intel	16	Ivy Br.	Xeon	58.9	GCP	19	2	2	2	15	15	15	11	11	11
	Intel	16	Haswell	Xeon	58.9	GCP	20	1	1	1	**16**	**16**	**16**	**12**	**12**	**12**
	Intel	16	Broadwell	Xeon	58.9	GCP	20	1	1	1	16	16	16	12	12	12
	Intel	16	Skylake	Xeon	62.8	GCP	21	3	3	3	16	16	16	**13**	**12**	**13**
	Intel	16	Skylake	Xeon	58.9	GCP	21	3	3	3	16	16	16	13	13	13
(56)	Intel	16	Ice Lake	Xeon	62.8	GCP	22	3	3	3	16	16	16	13	13	13
	Intel	32	Sandy Br.	Xeon	117.9	GCP	23	2	2	2	15	15	15	**14**	**14**	**14**
	Intel	32	Ivy Br.	Xeon	117.9	GCP	23	2	2	2	15	15	15	14	14	14
	Intel	32	Haswell	Xeon	117.9	GCP	24	1	1	1	16	16	16	**15**	**15**	**15**
	Intel	32	Broadwell	Xeon	117.9	GCP	24	1	1	1	16	16	16	15	15	15
(61)	Intel	32	Skylake	Xeon	125.9	GCP	25	3	3	3	16	16	16	**16**	**16**	**16**
	Intel	32	Skylake	Xeon	117.9	GCP	25	3	3	3	16	16	16	16	16	16
	Intel	32	Ice Lake	Xeon	125.8	GCP	26	3	3	3	16	16	16	16	16	16
	Intel	48	Skylake	8275CL	92.2	AWS	25	3	3	3	16	16	16	16	16	16

Table SUP-2: Full results for GPU instances. EQCs are assigned increasing integers from top to bottom. Table cells identical with their left neighbor are slightly faded. The first occurrence of an EQC per column is marked in bold. Cells marked with an asterisk (*) indicate that indeterminsm was observed. The equivalence class is based on the most frequently observed output.

			Dataset	CIFAR-10						DeepWeeds		
			Model size	Small			Medium			Large		
			Sample index	0	1	6	0	1	6	0	1	6
	Vendor	**Generation**	**Device name**	**Cloud**								
	NVidia	Kepler	K80	GCP	**1**	1	1	1*	1*	1*	1	1	1
	NVidia	Maxwell	GTX 970	local	**2***	2	2	2	2	2*	2	2	2
	NVidia	Maxwell	GTX 980	local	**3***	3	3	3	3	3	3	3	3
	NVidia	Maxwell	M60	AWS	3	3	3	3*	**4**	4*	4	4	4
(5)	NVidia	Pascal	P100	GCP	**4***	4*	4*	4*	**5***	5*	5	5	5
	NVidia	Volta	V100	GCP	**5***	5*	5*	5*	**6***	6*	6*	6*	6*
	NVidia	Turing	GTX 1650	local	**6**	6	6	6	**7**	7	7	7	7
	NVidia	Turing	RTX 2070	local	**7**	7	7	7	**8**	8*	8	8	8
	NVidia	Turing	T4	AWS	7*	7*	7*	**8**	**9**	9	9*	9*	9
(10)	NVidia	Turing	T4	GCP	7	7	7	8*	9*	9	9*	9*	9*
	NVidia	Ampere	A100	GCP	**8**	8	8	**9**	**10***	10*	10*	10*	10*

To ensure a clean tensor graph for each experiment, we create a new Python session for each inference. To this end we create a small CLI application that takes the model name and input path as arguments and computes the inference inside the container.

Dead ends In additional but eventually unsuccessful steps, we aimed to fully understand the paths taken during execution by instrumenting TensorFlow with the `gdb` debugger, as well as the `perf` and `valgrind` profiling tools. We specifically hoped that `valgrind`'s [7] cachegrind tool would provide insight into the actual code executed on CPU, but the tremendous amount of inlining in TensorFlow's codebase yielded no usable results. Outputs from `perf` were too noisy to be useful. TensorFlow's codebase was too large for `gdb` [12] analysis to be useful, both in interactive and automated scenarios. The record-and-replay debugger `rr` [9] could not deal with the complexities of TensorFlow's codebase and could not successfully record a single inference.

C Convolution algorithms and functions

We summarize the main approaches for computing convolutions.

General matrix multiplication (GEMM) Convolution can be calculated via matrix multiplication by extracting the relevant parts of the image into a Toeplitz matrix, replicating the filter as needed, and multiplying the two matrices. This is equivalent to an unrolled loop variant of the naive implementation with data replication for better access [1]. Even with data structures optimized for sparse matrices, the Toeplitz matrix has many duplicate entries, producing significant memory overhead. A divide-and-conquer alternative reduces the memory overhead by utilizing the fact that matrices are stored in memory as contiguous 1-dimensional arrays [1]. The full convolution is split into multiple smaller convolutions, and the results are reconstructed afterwards. The divide-and-conquer approach causes a different order of aggregations, which can lead to numerically different results.

Winograd algorithm Winograd short convolution [3] is a fast convolution algorithm that transforms both inputs and filters to achieve a lower number of multiplications. Its transformation is inherently lossy and involves multiple rounding steps. The Winograd algorithm exists in fused and non-fused variants. The fused variant performs transformation, point-wise multiplication, and inverse transformation in a single step, reducing the number of memory accesses, whereas the non-fused variant performs all steps separately.

Table SUP-3: Full list of CPU flags in alphabetical order and their corresponding cluster index. Flags with cluster index *C* (common) are present on all CPUs and are not show in the original table.

Flag	Cluster	Flag	Cluster	Flag	Cluster
3dnowext	3	fma	0	pse36	*C*
3dnowprefetch	7	fpu	*C*	pti	14
abm	0	fsgsbase	10	rdpid	3
adx	7	fxsr	*C*	rdrand	10
aes	*C*	fxsr_opt	3	rdrnd	10
apic	*C*	gfni	2	rdseed	7
arat	*C*	hle	11	rdtscp	*C*
arch_capabilities	4	ht	*C*	rep_good	*C*
avx	*C*	hypervisor	*C*	rtm	11
avx2	0	ibpb	*C*	sep	*C*
avx512_bitalg	2	ibrs	*C*	sha	6
avx512_vbmi2	2	ibrs_enhanced	8	sha_ni	6
avx512_vnni	8	invpcid	12	smap	7
avx512_vpopcntdq	2	invpcid_single	12	smep	10
avx512bitalg	2	lahf_lm	*C*	ss	4
avx512bw	1	lm	*C*	ssbd	*C*
avx512cd	1	mca	*C*	sse	*C*
avx512dq	1	mce	*C*	sse2	*C*
avx512f	1	md_clear	4	sse4_1	*C*
avx512ifma	2	misalignsse	3	sse4_2	*C*
avx512vbmi	2	mmx	*C*	sse4a	3
avx512vbmi2	2	mmxext	3	ssse3	*C*
avx512vl	1	movbe	0	stibp	*C*
avx512vnni	8	mpx	13	syscall	*C*
avx512vpopcntdq	2	msr	*C*	topoext	3
bmi1	0	mtrr	*C*	tsc	*C*
bmi2	0	nonstop_tsc	*C*	tsc_adjust	*C*
clflush	*C*	nopl	*C*	tsc_known_freq	*C*
clflushopt	5	npt	3	umip	6
clwb	5	nrip_save	3	vaes	2
clzero	3	nx	*C*	vme	*C*
cmov	*C*	osvw	3	vmmcall	3
cmp_legacy	3	osxsave	*C*	vpclmulqdq	2
constant_tsc	*C*	pae	*C*	x2apic	4
cpuid	*C*	pat	*C*	xgetbv1	5
cr8_legacy	3	pcid	9	xsave	*C*
cx16	*C*	pclmulqdq	*C*	xsavec	5
cx8	*C*	pdpe1gb	*C*	xsaveerptr	3
de	*C*	pge	*C*	xsaveopt	*C*
erms	9	pni	*C*	xsaves	1
extd_apicid	3	popcnt	*C*	xtopology	4
f16c	10	pse	*C*		

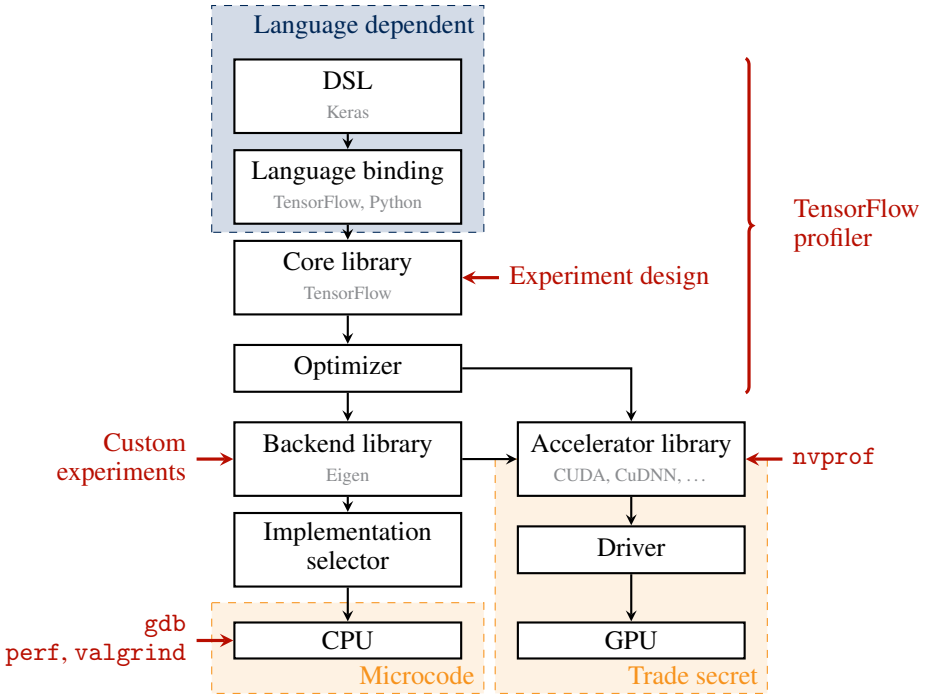


Figure SUP-1: Visualization of the ML software stack. Instrumentation and tools shown in red.

Fourier transformation approaches Approaches using fast Fourier transformation (FFT) exploit the fact that spatial convolution is equivalent to point-wise multiplication in the frequency domain. As FFT is also used in many other contexts, especially in signal processing, optimized implementations are commonly available. The transformation to and from the frequency domain is inherently lossy for finite numerical precision [6].

Most algorithms come in two major variants: an *explicit* variant, where precomputations happen in a separate step, and an *implicit* variant, where precomputations happen right in the algorithm (e. g., input replication for the im2 algorithm). In addition to the above mentioned algorithms, modern GPUs also include special hardware to compute convolution directly [8]. The performance of the above algorithms varies with the use case. Table SUP-4 compares advantages and disadvantages of each approach based on the literature [2].

D Datasets and preprocessing

We use the CIFAR-10 [5] and Deep Weeds [10] datasets for our experiments, both obtained from the `tensorflow_datasets` Python package. CIFAR-10 has ten classes and consists of 50 000 training and 10 000 test samples. Deep Weeds has nine classes and consists of 17 509 samples, which we split into training (first 85 %) and test set (final 15 %). We transform all samples from their integer range $[0, 255]$ to floating-point numbers in the range $[0, 1]$ by dividing by 255. All training and test sets are shuffled using the TensorFlow `dataset.shuffle` function with a buffer size of the entire dataset, and random seed 42. We batch the samples with a batch size of 32 for all datasets.

Table SUP-5 provides a summary of all models used in this paper.

For CIFAR-10 we use a small custom CNN with two convolutional layers (similar to the VGG architecture [11]), a ResNet18, and a ResNet50v2 [4] followed by a flatten and dense layer with the required 10 neurons and softmax activation. The small custom CNN is documented in Table SUP-6. For Deep Weeds we use the pre-trained model provided by the authors at `https://github.com/AlexOlsen/DeepWeeds`.

Training The CIFAR-10 models are trained for 30 epochs. The entire training set is used for every epoch. The models were trained on an RTX 3080 GPU.

Table SUP-4: Overview of convolution algorithms. Characteristics based on [2].

Approach	Time	Memory	Strided	Generation	Name
direct loop	−	++	++	Volta	fused conv/ReLU
					grouped naive kernel
GEMM	+	−− / +	++ / −−	Ampere	implicit
				Kepler	GEMM
				generic	explicit single precision
				generic	implicit
				generic	precomputed
Winograd	++	−	−	Ampere	Winograd
				Maxwell	Winograd
				Maxwell	non-fused
				Turing	non-fused
				Volta	compiled
				Volta	non-fused
FFT		−	+		FFT GEMM

Table SUP-5: Summary of models used.

		Parameters		
Dataset	ResNet[4]	Convolution layers	Total	Test set set accuracy
CIFAR-10 [5]	`18`	11,170,816	11,191,306	60 %
DeepWeeds [10]	`50v2`	23,556,608	24,744,457	95 %
CIFAR-10	`Cifar10-small`	Custom (cf. Table SUP-6)	464	59,354

Table SUP-6: Summary of model `Cifar10-small`. ReLU activation and max pooling are used except for the experiments in Section E.

Layer name	Layer type	Output shape	# params
`input`	InputLayer	$32 \times 32 \times 3$	0
`conv2d`	Conv2D	$32 \times 32 \times 3$	84
`activation`	Activation	$32 \times 32 \times 3$	0
`pooling2d`	Pooling2	$16 \times 16 \times 3$	0
`conv2d_1`	Conv2D	$16 \times 16 \times 5$	380
`activation`	Activation	$32 \times 32 \times 3$	0
`pooling2d_1`	Pooling2	$8 \times 8 \times 5$	0
`flatten_1`	Flatten	320	0
`dense_1`	Dense	128	41088
`dense_2`	Dense	128	16512
`dense_3`	Dense	10	1290

Evaluation The Deep Weeds model reaches 95 % accuracy on the test set, as reported in the original paper [10]. For CIFAR-10, model `Cifar10-small` reaches 53.18 % accuracy, and the `Cifar10-R18` reaches 60.25 % accuracy. These accuracies are not competitive with the state of the art, but sufficiently better than random guessing. We can safely assume that the kernels learn meaningful weights.

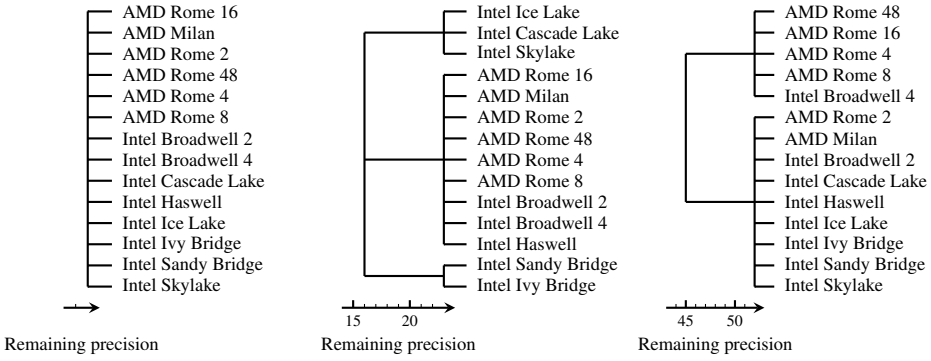
Experiment samples We process three samples for each of our models to measure the consistency of our results. The first sample is the first test sample (for simplicity); we additionally use a sample from a different class (sample index 1 for CIFAR-10, and index 6 for Deep Weeds), a sample from the same class as the first sample is also used (index 6 for CIFAR-10, and index 1 for Deep Weeds). All sample indexes refer to the unshuffled test set of the respective dataset.

E Experiments supporting the rebuttal and author response phase

Switching precision for all models

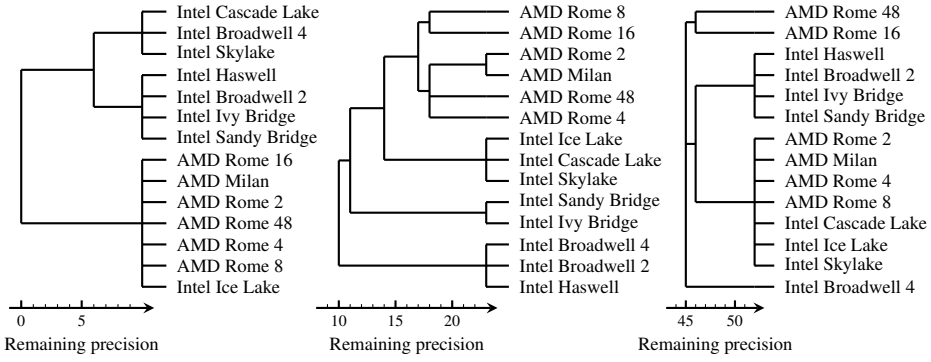
Request: "Maybe other Neural Networks could be tested to see if they follow the same pattern for single-point precision."

We repeat the experiments producing Figure 6 in the main paper for both the `Cifar10-small` and `DeepWeeds-R50v2` models. Results for `Cifar10-small` are shown in Figure SUP-2, and results for `DeepWeeds-R50v2` are shown in Figure SUP-3. The figures are structured in the same way as Figure 6 in the main paper.



(a) Half precision: 1 EQC. (b) Single precision: 3 EQCs. (c) Double precision: 2 EQCs.

Figure SUP-2: Influence of casting the `Cifar10-small` model to different floating-point precisions. Both half and double precision floating-point generate fewer deviations than single precision.



(a) Half precision: 3 EQCs. (b) Single precision: 7 EQCs. (c) Double precision: 4 EQCs.

Figure SUP-3: Influence of casting the `DeepWeeds-R50v2` model to different floating-point precisions. Both half and double precision floating-point generate fewer deviations than single precision.

Table SUP-7: Validation accuracy of modified models used for Figures SUP-4 and SUP-5.

Model name	Activation	Pooling	Validation accuracy
`Cifar10-R18`	Sigmoid	AvgPool	47.7 %
`Cifar10-small`	Sigmoid	AvgPool	54.6 %

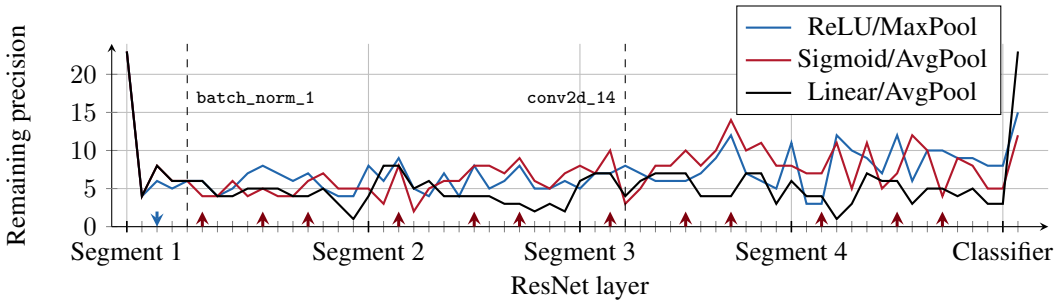


Figure SUP-4: Influence of activation and pooling functions for ResNet-18 architecture. Activation layers are indicated on the x-axis by red upward-facing arrows; pooling layers are indicated by blue downward-facing arrows. Model variants with sigmoid activation use Xavier initialization (`glorot_uniform`).

Again, the number of EQCs is largest for single precision, and decreases for both half and double precision. The distribution of EQCs for single precision is similar to the `Cifar10-R18` model. Changes for double precision are less clear cut, and CPUs with different core counts fall into the same EQC. We conclude that the pattern of EQCs is similar for all models: single precision generates the most EQCs, and half and double precision generate fewer EQCs, but still more than one.

Weight distribution and stable remaining precision

Request: "can the authors try Xavier initialization + sigmoid activation (instead of He + ReLU in typical Resnet) or replace MaxPool to MeanPool to see if this behavior still holds. This should tell which is the cause of non-diminishing remaining precision."

To answer this question we modify models `Cifar10-small` and `Cifar10-R18` to use sigmoid activation with Xavier initialization for all activation layers. `MaxPool` layers are replaced with `AvgPool`[1] layers. The resulting models are trained for 30 epochs on the CIFAR-10 training set, using the same code as the ReLU models. No tuning of hyperparameters is performed. Table SUP-7 reports the final validation accuracy of the models. As with the ReLU models, this is not competitive with the state of the art, but significantly better than random guessing. Figures SUP-4 and SUP-5 show the results for the modified models. Because the reviewer specifically mentions the initialization we include an untrained version of the model in the results, shown in Figures SUP-6 and SUP-7

We follow the same experimental procedure as in Section 3.3 in the paragraph "Architecture and layer types" to obtain the remaining precision and number of EQCs. Figure SUP-4 show the results for model `Cifar10-R18`. The tick marks indicating the activation layers have been replaced by red upward-facing arrows. Blue downward-facing arrows indicate pooling layers.

The remaining precision for ReLU activation in Figure SUP-4 is the same as in Figure 5 in the main paper, and activation layers either increase the remaining precision or leave it unaffected. In contrast, the first and last sigmoid activation layers decrease the remaining precision. The remaining sigmoid activation layers also either increase the remaining precision or leave it unaffected, same as the ReLU activation layers.

The single pooling layer after the first convolution increases the remaining precision for both `MaxPool` and `AvgPool`. The number of EQCs is the same for both variants of the model, and is not shown to save space.

[1]We stay consistent with TensorFlow naming and refer to MeanPool as AvgPool.

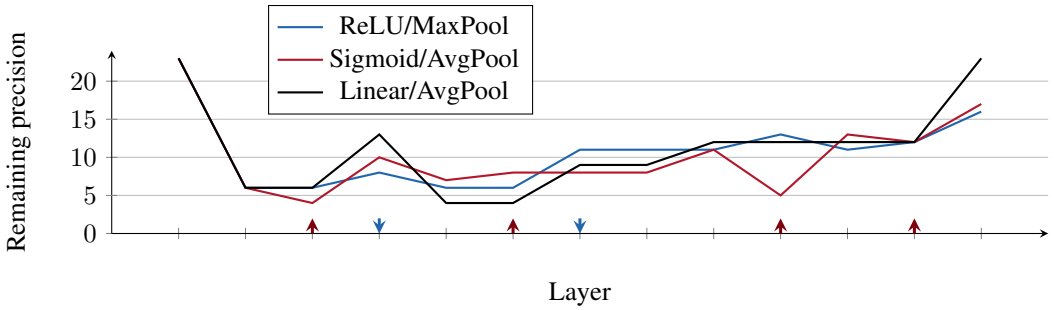
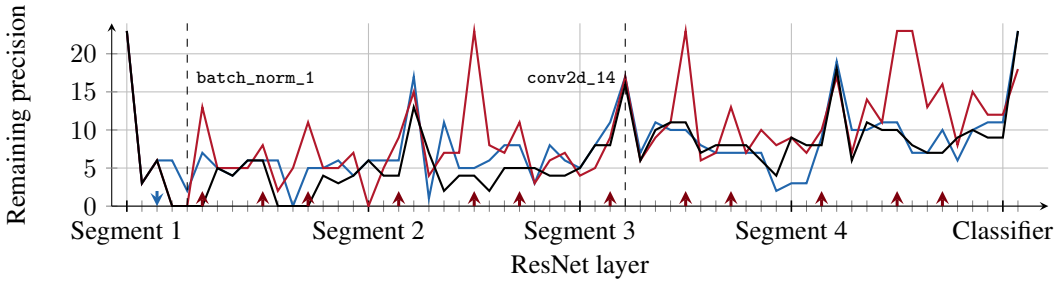
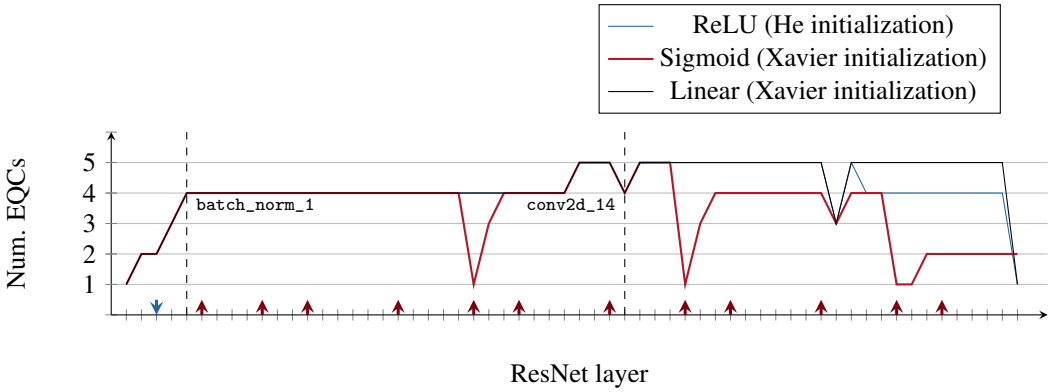


Figure SUP-5: Influence of activation and pooling functions for `Cifar10-small` (cf. Table SUP-6). Activation layers are indicated on the x-axis by red upward-facing arrows; pooling layers are indicated by blue downward-facing arrows. Model variants with sigmoid activation use Xavier initialization (`glorot_uniform`).



(a) Remaining precision

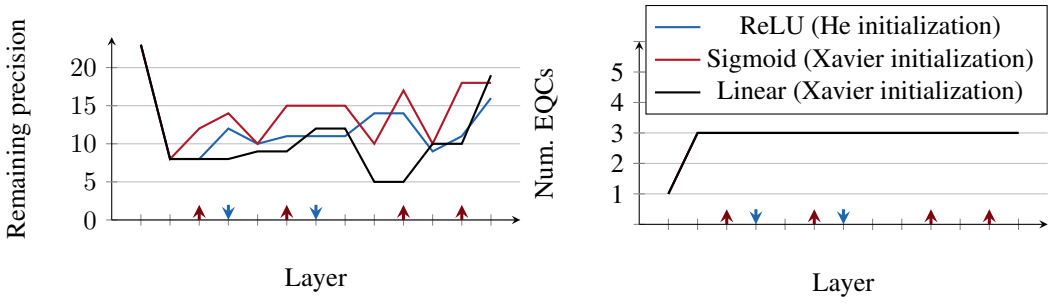


(b) Number of equivalence classes

Figure SUP-6: Variant of Figure 5 of the main paper, with initialized weights and no training. Sigmoid uses Xavier initialization, ReLU uses He. Linear uses initialized weights of the sigmoid model.

Figure SUP-5 shows the results for model `Cifar10-small`, which features a second pooling layer. The figure is structured the same way as Figure SUP-4. For the `Cifar10-small` model, sigmoid activation reduces the remaining precision in three out of four cases. `MaxPool` increases the remaining precision for both layers, whereas `AvgPool` increases it for one and leaves it unaffected for the other.

Because the reviewer explicitly mentioned activation functions, we also include a variant of the graphic with initialized weights and no training, shown in Figure SUP-6 for the `Cifar10-R18` model, and in Figure SUP-7 for the `Cifar10-small` model. Remaining precision for the `Cifar10-R18` model without training fluctuates across the entire possible range $[0, 23]$. A remaining precision of 23 indicates only a single EQC. The EQC plot in Figure SUP-6b shows that even after



(a) Remaining precision

(b) Number of equivalence classes (all overlap)

Figure SUP-7: Variant of Figure SUP-5 with initialized weights and no training. Methodology is identical to Figure SUP-6.

Table SUP-8: Number of deviations before and after activation layers in `Cifar10-R18` and `Cifar10-small`. Deviations are counted over the flattened output of the layer, and averaged across used in Tables SUP-1 and SUP-2. A value deviates if it is not identical for all hardware platforms.

Model name	**Deviations**			
	ReLU		Sigmoid	
Layer index	Before	After	Before	After
`Cifar10-R18`				
5	95.3 %	38.4 %	94.1 %	79.5 %
9	97.5 %	34.6 %	97.7 %	87.8 %
12	98.5 %	33.8 %	89.9 %	80.8 %
18	99.3 %	49.7 %	99.5 %	91.7 %
23	98.6 %	43.1 %	98.4 %	88.2 %
26	97.9 %	35.2 %	96.4 %	87.9 %
32	97.9 %	47.2 %	98.5 %	91.8 %
37	93.9 %	43.3 %	93.1 %	70.8 %
40	75.6 %	30.8 %	19.9 %	7.2 %
46	90.8 %	61.0 %	96.5 %	96.0 %
51	71.4 %	54.8 %	97.9 %	97.5 %
54	79.9 %	30.1 %	12.5 %	8.9 %
`Cifar10-small`				
2	86.4 %	49.8 %	72.3 %	44.0 %
5	97.0 %	37.6 %	91.7 %	71.2 %
9	96.6 %	26.8 %	82.3 %	63.8 %
11	99.2 %	30.5 %	93.2 %	77.1 %

all EQCs collapse, new EQCs can arise. This implies that similar behavior is possible for trained weights, and having identical results in intermediate layers does not guarantee identical results in subsequent layers.

Notably, we find more cases where sigmoid activation increases remaining precision for the untrained model. A possible cause for this is the fact that the initialized weights have a lower energy (sum of values), which causes the sigmoid activation to be closer to its linear regime.

In addition to the figures, we count the ratio of deviating values before and after activation. Table SUP-8 shows the results for the trained models.

Table SUP-9: Number of layers that increase, decrease, or leave the remaining precision unaffected. Results are averaged over all samples used in Tables SUP-1 and SUP-2.

	ReLU			Sigmoid		
Model name	Increase	Unaffected	Decrease	Increase	Unaffected	Decrease
`Cifar10-R18`	6.000	6.000	0.000	7.667	2.000	2.333
`Cifar10-small`	3.000	1.000	0.000	1.000	0.333	2.667

Altering the network's size

Request: "Was any experiment performed to alter the depth/size of the network under test, to see if that would impact the probability of a divergence occurring as depth increased?"

There is no experiment that explicitly cuts, shrinks or enlarges middle layers to investigate the effects. However, the results in Figure 5 were obtained by outputting intermediate layer results, and the results in Figure 4 can be interpreted as varying the size of a single convolutional layer. Both figures show clear trends on how the size affects the number of EQCs as well as the remaining precision. Our analysis of TensorFlow and its underlying Eigen library tells us that actually cutting the layers would result in the same remaining precision and EQCs as shown in Figure 5 because implementation choice on CPUs depends only on the hardware and not on the model. On GPUs, cutting layers will affect these metrics, as early layers will take up memory on the GPU and affect the microbenchmarks of subsequent layers.

References

[1] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. High-performance low-memory lowering: GEMM-based algorithms for DNN convolution. In *Interantional Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 99–106. IEEE, 2020.

[2] Richard E. Blahut. *Fast Algorithms for Signal Processing*. Cambridge University Press, 2010.

[3] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778. IEEE, 2016.

[5] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset. Master's thesis, University of Toronto, 2014.

[6] Raimund Meyer. Error analysis and comparison of FFT implementation structures. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 888–891. IEEE, 1989.

[7] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[8] NVidia Corporation. NVidia TensorCores. `https://www.nvidia.com/en-us/data-center/tensor-cores/`, 2022.

[9] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 377–389. USENIX Association, 2017.

[10] Alex Olsen, Dmitry A Konovalov, Bronson Philippa, Peter Ridd, Jake C Wood, Jamie Johns, Wesley Banks, Benjamin Girgenti, Owen Kenny, James Whinney, et al. DeepWeeds: A multiclass weed species image dataset for deep learning. *Scientific Reports*, 9(1):1–12, 2019.

[11] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations (ICLR)*, 2015.

[12] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 675, 1988.