Numerical influence of ReLU'(0) on backpropagation

David Bertoin, Jérôme Bolte, Sébastien Gerchinovitz, Edouard Pauwels

Numerical influence of ReLU'(0) on backpropagation Erratum

David Bertoin
IRT Saint Exupéry
ISAE-SUPAERO
ANITI
Toulouse, France
david.bertoin@irt-saintexupery.com

Jérôme Bolte
Toulouse School of Economics
Université Toulouse 1 Capitole
ANITI
Toulouse, France
jbolte@ut-capitole.fr

Sébastien Gerchinovitz
IRT Saint Exupéry
Institut de Mathématiques de Toulouse
ANITI
Toulouse, France
sebastien.gerchinovitz@irt-saintexupery.com

Edouard Pauwels
CNRS
IRIT, Université Paul Sabatier
ANITI
Toulouse, France
edouard.pauwels@irit.fr

Erratum

This is an erratum for the paper "Numerical influence of ReLU'(0) on backpropagation" [1]. We would like to express our gratitude to Wonyeol Lee, Sanghyuk Chun, and Sejun Park for bringing to our attention an important issue in our implementation. Their feedback helped us identify a simple, yet impactful bug in the code used for our experiments, specifically in the backward implementation of our modified version of the ReLU function. We deeply regret this error and sincerely apologize for any confusion it may have caused. The purpose of this note is to report results of the same experiments as in [1] with a correct implementation. We then replicate the published paper, without any modification.

Impact on results of the paper: Despite this error in our implementation, the main results of our paper regarding the existence and relative size of the bifurcation zone remain valid. After correcting the bug we obtain qualitatively similar results.

Nevertheless, upon further investigation, we have discovered that the magnitude of relative change (in norm) due to the choice made for the value of ReLU'(0) is much smaller than what we initially announced. This is a consistent observation throughout all experiments. As a consequence, the impact of this phenomenon on actual neural network training is much less significant compared to what we initially believed; see details in Section 4. We found that, on benchmark training scenarios, a significant degradation occurs for very large values, around 10000. Therefore, as stated in [1], the value of ReLU'(0) can numerically influence neural network training, contrary to what (infinite precision) theory predicts, but the corresponding values are unrealistic in practical scenarios.

Content of this note: We provide corrections and discuss the experiments reported in [1] that were impacted by this implementation error. We present revised empirical results obtained after correcting our mistake, that reflect the actual influence of ReLU'(0) on backpropagation. For ease of reading we keep the overall structure of the original paper [1] (numbering of sections and figures) unchanged and only make brief comments focused on qualitative or quantitative change in the experimental results. To keep this note short we refer to [1] for all the experimental details (see also the paper after this note).

Preprint.

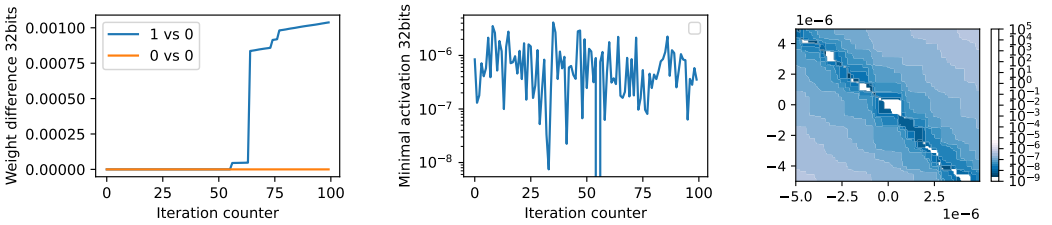

Figure 1: This is similar to [1, Figure 1], the only difference is the order of magnitude of differences on the left plot.

Nature of the mistake: In our original `pytorch` implementation, we unintentionally neglected to properly multiply ReLU'(0) by the output gradients as required by the chain rule. For example the following code provides first an incorrect implementation of ReLU'(0) = 0.5 based on the relu function given in `pytorch`, the one which we used in [1]. A correct implementation is given below and should implement a multiplication, not an absolute assignment, to be in line with the implementation of automatic differentiation in `pytorch`.

```
# Incorrect backward method
grad_input[input == 0] = 0.5

# Correct backward method
grad_input[input == 0] *= 0.5
```

1 Introduction

The main message of this section is not impacted by the implementation error and the observation reported in our preliminary experiments remain valid, see Figure 1. The only noticable difference is a modification of the order of magnitude of weight differences which does not affect the main message.

2 On the mathematics of backpropagation for ReLU networks

The bug has no impact on this section which is a theoretical description of the mechanisms at stake. Let us recall that from a theoretical point of view, any choice of ReLU'(0) in $\mathbb{R}$ provides a valid nonsmooth optimization oracle, even beyond the subgradient $[0, 1]$.

3 Surprising experiments on a simple feedforward network

This section was mostly centered on the volume of the bifurcation zone. Our empirical observations remain unchanged regarding this respect, see Proportions in Table 1 and Figure 2. On the other hand, the relative difference in L^2 norm reported in Table 1 are reduced by several orders of magnitude compared to results reported in [1]. This is consistent with our observation in Figure 1 and constitutes the main difference with the results reported in [1].

Floating-point precision	16 bits	32 bits	64 bits
Proportion of $\{\theta_i\}_{i=1}^M$ in S	100%	43%	0%
Proportion of impacted mini-batches	0.005%	0.0002%	0%
Relative L_2 difference for impacted mini-batches (1st quartile, median, 3rd quartile)	(1e-3, 2e-3, 3e-3)	(1e-3, 2e-3, 3e-3)	(0, 0, 0)

Table 1: The proportions presented in this table are similar as the ones given in [1, Table 1], up to non-significant variations. The main difference is in the L_2 amplitude of the differences which is reduced by a factor 10^5. This is consistent with the preliminary experiment in Figure 1 and explains the behavior of training algorithms described in the next section.

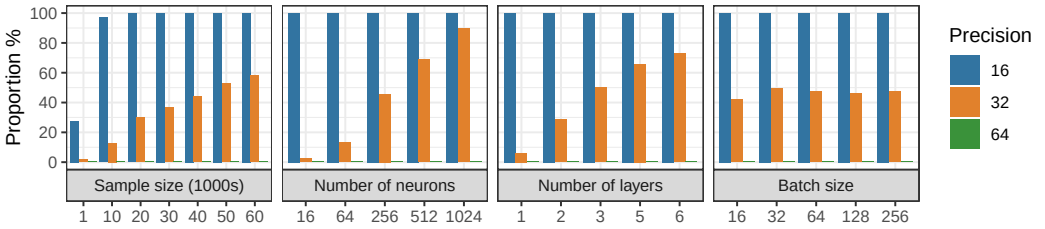

Figure 2: This figure is completly similar to [1, Figure 2], up to non-significant variations.

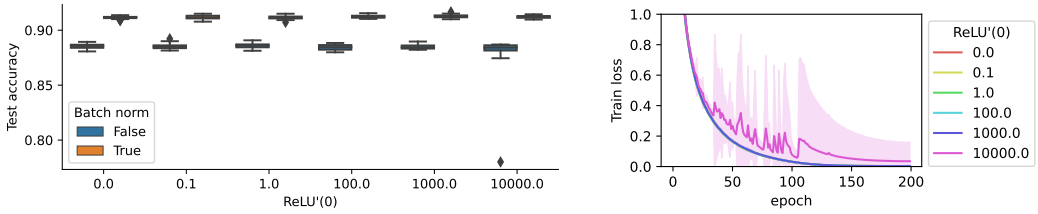

Figure 3: Quantitatively, the behavior described in this figure is drastically different from the one in [1, Figure 3]: the effect of the choice of ReLU'(0) is not significant for reasonable choices (between 0 and 1). There is still an effect, for ReLU'(0) = 10000, which is unreasonably large. We observe oscillations during training and the test error considerably devreases for some runs.

4 Consequences for learning

The implementation error we encountered has a significant impact on the findings presented in Section 4 of [1], where we analyzed the impact of different choices of ReLU'(0) in terms of training on different benchmark architectures.

After correcting our mistake, we found that the choice of ReLU'(0) does have an impact on neural network training, which is qualitatively similar to what was reported in [1]. However, and consistently with preliminary results in Section 3, the magnitude of this impact is considerably smaller than originally described in [1], and occurs only for very large values of ReLU'(0), typically greater than 1000 (see Figure 3 and appendix D).

It is important to note that such large values are not realistic in practical scenarios. When ReLU'(0) takes on these extreme values, it leads to pronounced training oscillations, often resulting in non-numeric (NaN) values. Although qualitatively similar, these observations are quantitatively very different from results reported in [1], which considerably diminishes the overall potential consequences of our findings.

Consistently with [1], we observe that the Adam optimizer as well as batch normalization tame the phenomenon by mitigating the adverse effects caused by the large values of ReLU'(0) and stabilize the training process (see Figure 4). Furthermore, similarly to Section 3, the experiments related to volume and frequency estimation are very similar to the results reported in [1].

4.2 Effect on training and test errors

Our conclusions regarding the impact of the choice of ReLU'(0) in term of deep neural network training are affected by the bug. For reasonable choices of ReLU'(0), we do not observe any effect in term of training loss or test accuracy. We do observe an effect, qualitatively similar as reported in [1], but only for very large values, which do not correspond to practical scenarios. The corrected results are presented in Figure 3 for a VGG network trained on CIFAR 10 with SGD, in Figure 4 for the same problem with the Adam optimizer and for the same network on SVHN trained with SGD. We provide an additional experiment with a ResNet in the appendix (Figure 11), to confirm the effect of very large values for ReLU'(0). For the larger experiment on ImageNet with a ResNet50 reported in [1], we did not observe qualitative changes and we therefore do not report these results.

4.3 Mitigating factors: numerical precision, batch-normalization and Adam

Since the observed effect is much less significant than what we initially believed, the discussion regarding mitigating factors is much less relevant and we will not put emphasis on this aspect. We still reproduced our experiments as reported in Figure 4 which shows that the Adam optimizer as well as batch normalization tame the oscillations for large values of ReLU'(0).

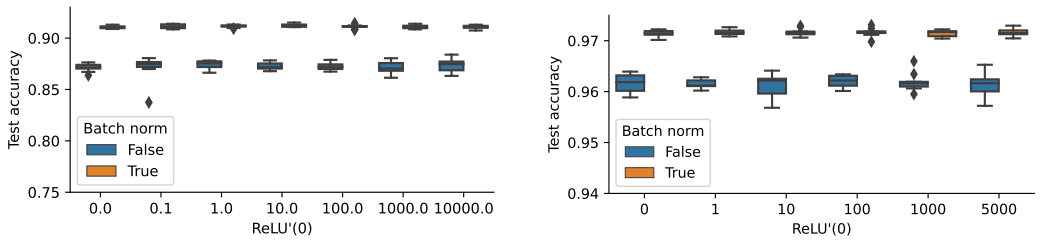

Figure 4: Reproduction of [1, Figure 4], using the Adam optimizer tames the oscillations, even for very large values of ReLU'(0) (left). On the right the experiment on SVHN does not show a significant effect, but some runs (not represented in the figure) did not converge due to oscillations.

4.4 Back to the bifurcation zone: more neural nets and the effects of batch-norm

The experiments of this section are reported in Figure 5 and are qualitatively coherent with the same experiment in [1]. Numerical precision is the main factor behind the bifurcation zone, and we do not observe any bifurcation zone at 64 bits precision. Furthermore batch normalization seems to reduce the size of the bifurcation zone.

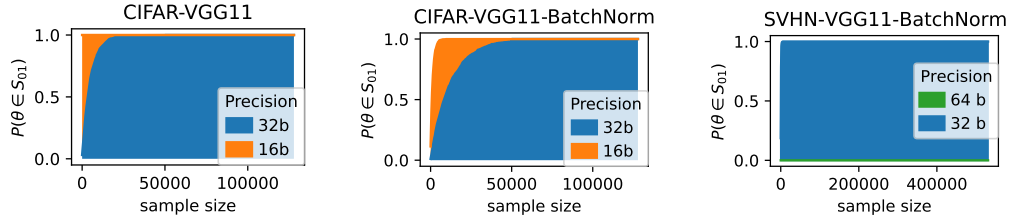

Figure 5: The results reported in this figure are qualitatively similar as the results presented in [1, Figure 5]. The effect of batch norm in 32 bits precision is different and coherent with 16 bits precision, we did not investigate further the significance of this variation.

5 Conclusions and future work

In conclusion, this erratum highlights the correction of the implementation error and provides insights into the revised impact of ReLU'(0) on backpropagation. The primary message of the paper, emphasizing the existence of an effect due to the choice of the value of ReLU'(0) on neural network training, remains intact. However, the practical consequences and impact of this finding on actual training is more mitigated than previously believed. Indeed, in reasonable practical scenarios, the quantitative impact experimentally measured on leaning benchmarks is not significant.

References

[1] Bertoin, D., Bolte, J., Gerchinovitz, S., Pauwels, E. (2021). Numerical influence of ReLU'(0) on backpropagation. Advances in Neural Information Processing Systems, 34, 468-479.

D Complements on experiments

D.2 Additional Experiments with MNIST and fully connected networks

Here again the corrected experiments show an impact for large values of ReLU'(0) (around 10000). Experimental details are in [1].

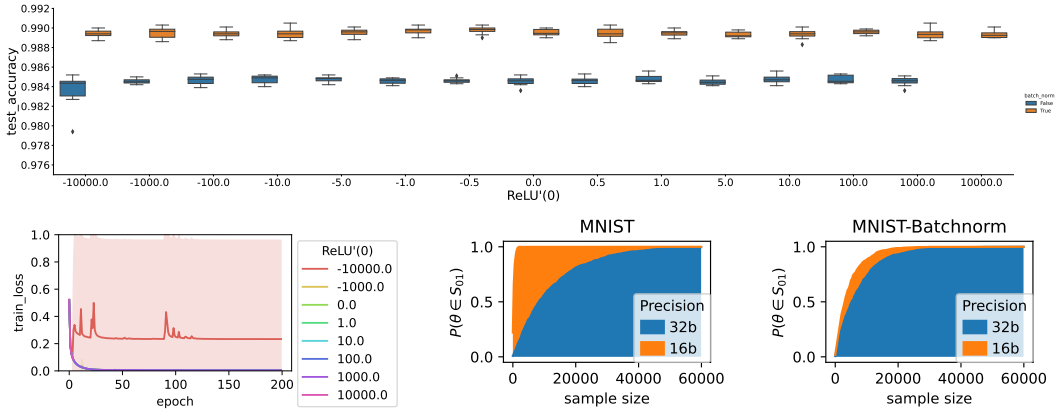

Figure 7: Reproduction of the results presented in [1, Figure 9], the results are coherent with observations in Figure 3.

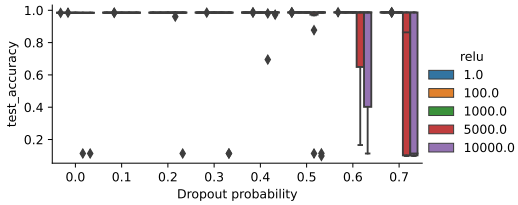

Figure 8: Reproduction of the results presented in [1, Figure 7] illustrating that dropout has a non significative effect on taming oscillations.

D.3 Additional experiments with VGG11

As reported in [1] batch normalization tames oscillations for large values of ReLU'(0) (Figure 9). Regarding the combined influence of ReLU'(0) and numerical precision (16, 32, or 64 bits) [1, Figure 14], the revised experiments reveal a notable impact for higher values of ReLU'(0) (around 10,000). This impact manifests as non-convergence when using 16-bit precision and a decline in test performance when employing 32-bit precision. (Figure 10). Experimental details are in [1].

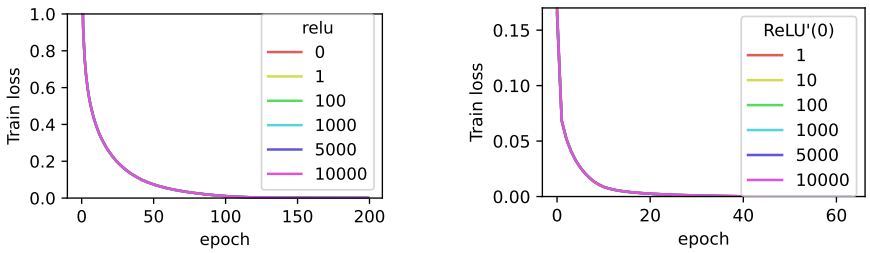

Figure 9: Training loss with VGG11 and batch norm, on CIFAR 10 (left) and SVHN (right). The instability induced by the choice of $\mathrm{ReLU}'(0)$ completely disappears with batch normalization.

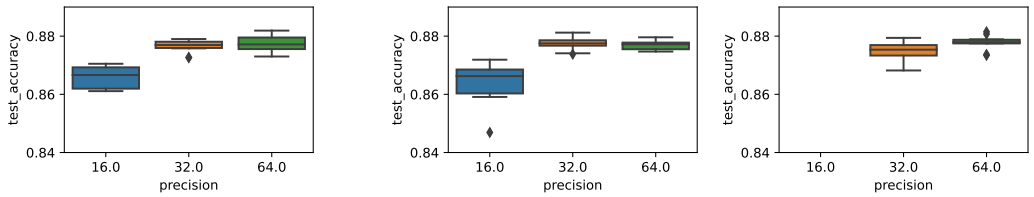

Figure 10: Test accuracy for different numerical precisions with a VGG11 network on CIFAR10. Left: $\mathrm{ReLU}'(0) = 0$. Center:$\mathrm{ReLU}'(0) = 1$. Right: $\mathrm{ReLU}'(0) = 1000$. For this last experiment, we did not manage to get training to converge in 16 bit precision.

D.4 Additional experiments with ResNet18

The revised experiments in Figure 11 confirm that the impact of ReLU'(0) on performance is observed for larger values, specifically when ReLU'(0) exceeds 100. This is coherent with results reported in [1] which observed that Resnet architectures are more sensitive to the choice of ReLU'(0).

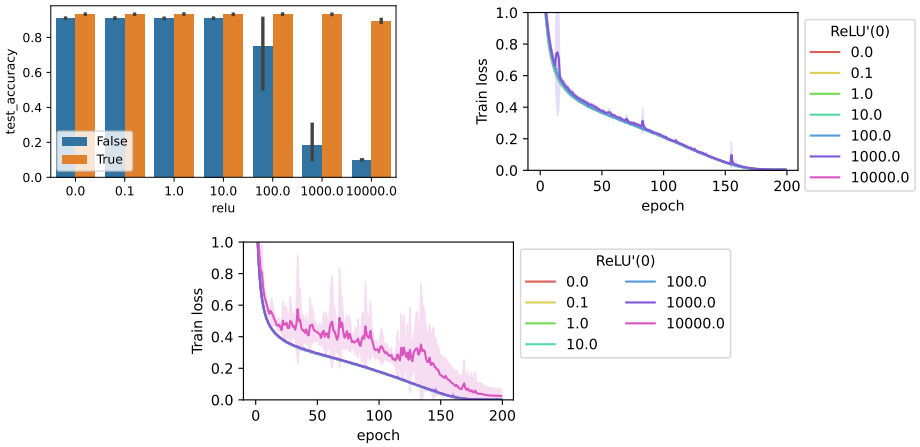

Figure 11: Training experiment on CIFAR10 with Resnet18 and the SGD optimizer. Top left: test accuracy with and without batch normalization (true / false). Top right: training loss during training without batch normalization (runs for ReLU'(0)=10000 led to NaNs). Bottom: training loss during training with batch normalization.

D.5 Additional experiments with ResNet50 on ImageNet

Contrary to our initial findings, which indicated a potential influence of ReLU'(0) on performance, the revised experiments showed no substantial effects. These results were consistent across various choices of ReLU'(0), including large values that were previously believed to have an impact.

Numerical influence of ReLU'(0) on backpropagation

David Bertoin
IRT Saint Exupéry
ISAE-SUPAERO
ANITI
Toulouse, France
david.bertoin@irt-saintexupery.com

Jérôme Bolte
Toulouse School of Economics
Université Toulouse 1 Capitole
ANITI
Toulouse, France
jbolte@ut-capitole.fr

Sébastien Gerchinovitz
IRT Saint Exupéry
Institut de Mathématiques de Toulouse
ANITI
Toulouse, France
sebastien.gerchinovitz@irt-saintexupery.com

Edouard Pauwels
CNRS
IRIT, Université Paul Sabatier
ANITI
Toulouse, France
edouard.pauwels@irit.fr

Abstract

In theory, the choice of $\mathrm{ReLU}'(0)$ in $[0, 1]$ for a neural network has a negligible influence both on backpropagation and training. Yet, in the real world, 32 bits default precision combined with the size of deep learning problems makes it a hyperparameter of training methods. We investigate the importance of the value of $\mathrm{ReLU}'(0)$ for several precision levels (16, 32, 64 bits), on various networks (fully connected, VGG, ResNet) and datasets (MNIST, CIFAR10, SVHN, ImageNet). We observe considerable variations of backpropagation outputs which occur around half of the time in 32 bits precision. The effect disappears with double precision, while it is systematic at 16 bits. For vanilla SGD training, the choice $\mathrm{ReLU}'(0) = 0$ seems to be the most efficient. For our experiments on ImageNet the gain in test accuracy over $\mathrm{ReLU}'(0) = 1$ was more than 10 points (two runs). We also evidence that reconditioning approaches as batch-norm or ADAM tend to buffer the influence of $\mathrm{ReLU}'(0)$'s value. Overall, the message we convey is that algorithmic differentiation of nonsmooth problems potentially hides parameters that could be tuned advantageously.

1 Introduction

Nonsmooth algorithmic differentiation: The training phase of neural networks relies on first-order methods such as Stochastic Gradient Descent (SGD) [14, 9] and crucially on algorithmic differentiation [15]. The fast "differentiator" used in practice to compute mini-batch descent directions is the backpropagation algorithm [29, 4]. Although designed initially for differentiable problems, it is applied indifferently to smooth or nonsmooth networks. In the nonsmooth case this requires surrogate derivatives at the non regularity points. We focus on the famous $\mathrm{ReLU} := \max(0, \cdot)$, for which a value for $s = \mathrm{ReLU}'(0)$ has to be chosen. A priori, any value in $[0, 1]$ bears a variational sense as it corresponds to a subgradient [27]. Yet in most libraries $s = 0$ is chosen; it is the case for TensorFlow [2], PyTorch [26] or Jax [10]. Why this choice? What would be the impact of a different value of s? How this interacts with other training strategies? We will use the notation $\mathrm{backprop}_s$ to denote backpropagation implemented with $\mathrm{ReLU}'(0) = s$ for any given real number s.[1]

[1] Definition in Section 2. This can be coded explicitly or cheaply emulated by $\mathrm{backprop}_0$. Indeed, considering $f_s \colon x \mapsto \mathrm{ReLU}(x) + s(\mathrm{ReLU}(-x) - \mathrm{ReLU}(x) + x) = \mathrm{ReLU}(x)$, we have $\mathrm{backprop}_0[f_s](0) = s$.

Preprint.

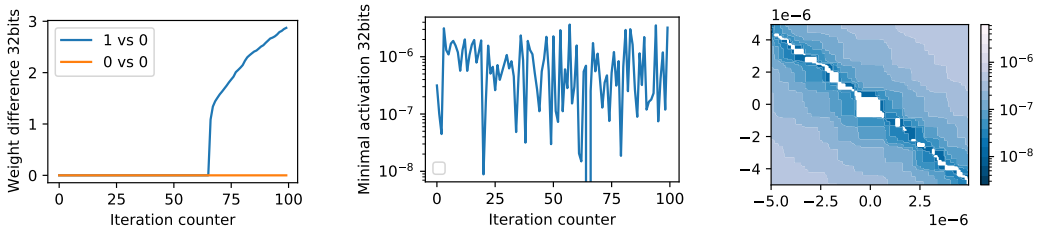

Figure 1: Left: Difference between network parameters (L^1 norm), 100 iterations within an epoch. "0 vs 0" indicates $\|\theta_{k,0} - \tilde{\theta}_{k,0}\|_1$ where $\tilde{\theta}_{k,0}$ is a second run for sanity check, "0 vs 1" indicates $\|\theta_{k,0} - \theta_{k,1}\|_1$. Center: minimal absolute activation of the hidden layers within the k-th mini-batch, before ReLU. At iteration 65, the jump on the left coincides with the drop in the center, and an exact evaluation of $\mathrm{ReLU}'(0)$. Right: illustration of the bifurcation zone at iteration $k = 65$ in a randomly generated network weight parameter plane (iteration 65 in the center). The quantity represented is the absolute value of the neuron of the first hidden layer which is found to be exactly zero within the mini-batch before application of ReLU (exact zeros are represented in white).

What does backpropagation compute? A popular thinking is that the impact of $s = \mathrm{ReLU}'(0)$ should be limited as it concerns the value at a single point. This happens to be exact in theory but for surprisingly complex reasons related to Whitney stratification (see Section 2 and references therein):

— For a given neural network architecture, $\mathrm{backprop}_s$ outputs a gradient almost everywhere, independently of the choice of $s \in \mathbb{R}$, see [7] and [5] for a detailed treatment of ReLU networks.

— Under proper randomization of the initialization and the step-size of SGD, with probability 1 the value of s has no impact during training with $\mathrm{backprop}_s$ as a gradient surrogate, see [8, 6].

In particular, the set of network parameters such that $\mathrm{backprop}_0 \neq \mathrm{backprop}_1$ is negligible and the vast majority of SGD sequences produced by training neural networks are not impacted by changing the value of $s = \mathrm{ReLU}'(0)$. These results should in principle close the question about the role of $\mathrm{ReLU}'(0)$. Yet, this is not what we observe in practice with the default settings of usual libraries.

A surprising experiment and the bifurcation zone: An empirical observation on MNIST triggered our investigations: consider a fully connected ReLU network, and let $(\theta_{k,0})_{k\in\mathbb{N}}$ and $(\theta_{k,1})_{k\in\mathbb{N}}$ be two training weights sequences obtained using SGD, with the same random initialization and the same mini-batch sequence but choosing respectively $s = 0$ and $s = 1$ in PyTorch [26]. As depicted in Figure 1 (left), the two sequences differ. A closer look shows that the sudden divergence is related to what we call the *bifurcation zone*, i.e., the set $S_{0,1}$ of weights such that $\mathrm{backprop}_0 \neq \mathrm{backprop}_1$. As mentioned previously this contradicts theory which predicts that the bifurcation zone is met with null probability during training. This contradiction is due to the precision of floating point operations and, to a lesser extent, to the size of deep learning problems. Indeed, rounding schemes used for inexact arithmetics over the reals (which set to zero all values below a certain threshold), may "thicken" negligible sets. This is precisely what happens in our experiments (Figure 1).

The role of numerical precision: Contrary to numerical linear algebra libraries such as `numpy`, which operates by default under 64 bits precision, the default choice in PyTorch is 32 bits precision (as in TensorFlow or Jax). We thus modulated machine precision to evaluate the importance of the bifurcation zone in Section 3. In 32 bits, we observed that empirically the zone occupies from about 10% to 90% of the network weight space. It becomes invisible in 64 bits precision even for quite large architectures, while, in 16 bits, it systematically fills up the whole space. Although numerical precision is the primary cause of the apparition of the zone, we identify other factors such as network size, sample size. Let us mention that low precision neural network training is a topic of active research [33, 20, 11, 16], see also [28] for an overview. Our investigations are complementary as we focus on the interplay between nonsmoothness and numerical precision.

Impact on learning: The next natural question is to measure the impact of the choice of $s = \mathrm{ReLU}'(0)$ in machine learning terms. In Section 4, we conduct extensive experiments combining different architectures (fully connected, VGG, ResNet), datasets (MNIST, SVHN, CIFAR10,

ImageNet) and other learning factors (Adam optimizer, batch normalization, dropout). In 32 bits numerical precision (default in PyTorch or Tensorflow), we consistently observe an effect of choosing $s \neq 0$. We observe a significant decrease in terms of test accuracy as $|s|$ increases; this can be explained by chaotic oscillatory behaviors induced during training. In some cases gradients even explode and learning cannot be achieved. The sensitivity to this effect highly depends on the problem at hand, in particular, on the network structure and the dataset. On the other hand the choice $s = 0$ seems to be the most stable. We also observe that both batch normalization [21] and—to a lesser degree—the Adam optimizer [22] considerably mitigate this effect. All our experiments are done using PyTorch [26]; we provide the code to generate all figures presented in this manuscript.

One important message is that, even if the default choice $s = 0$ seems to be the most stable, our experiments show a counter-intuitive phenomenon that illustrates the interplay between numerical precision and nonsmoothness, and calls for caution when learning nonsmooth networks.

Outline of the paper: In Section 2 we recall elements of nonsmooth algorithmic differentiation which are key to understand the mathematics underlying our experiments. Most results were published in [7, 8]; we provide more detailed pointers to this literature in Appendix A.1. In Section 3 we describe investigations of the bifurcation zone and factors influencing its importance using fully connected networks on the MNIST dataset. Neural network training experiments are detailed in Section 4 with further experimental details and additional experiments reported in Appendix D.

2 On the mathematics of backpropagation for ReLU networks

This section recalls recent advances on the mathematical properties of backpropagation, with in particular the almost sure absence of impact of $\mathrm{ReLU}'(0)$ on the learning phase (assuming exact arithmetic over the reals). The main mathematical tools are conservative fields developed in [7]; we provide a simplified overview which is applicable to a wide class of neural networks.

2.1 Empirical risk minimization and backpropagation

Given a training set $\{(x_i, y_i)\}_{i=1\ldots N}$, the supervised training of a neural network f consists in minimizing the empirical risk:

$$\min_{\theta \in \mathbb{R}^P} \quad J(\theta) := \frac{1}{N} \sum_{i=1}^{N} \ell(f(x_i, \theta), y_i) \tag{1}$$

where $\theta \in \mathbb{R}^P$ are the network's weight parameters and ℓ is a loss function. The problem can be rewritten abstractly, for each $i = 1, \ldots, N$ and $\theta \in \mathbb{R}^P$, $\ell(f(x_i, \theta), y_i) = l_i(\theta)$ where the function $l_i \colon \mathbb{R}^P \to \mathbb{R}$ is a composition of the form

$$l_i = g_{i,M} \circ g_{i,M-1} \circ \ldots \circ g_{i,1} \tag{2}$$

where for each $j = 1, \ldots, M$, the function $g_{i,j}$ is locally Lipschitz with appropriate input and output dimensions. A concrete example of what the functions $g_{i,j}$ look like is given in Appendix A.2 in the special case of fully connected ReLU networks. Furthermore, we associate with each $g_{i,j}$ a generalized Jacobian $J_{i,j}$ which is such that $J_{i,j}(w) = \mathrm{Jac}_{g_{i,j}}(w)$ whenever $g_{i,j}$ is differentiable at w and Jac denotes the usual Jacobian. The value of $J_{i,j}$ at the nondifferentiability loci of $g_{i,j}$ can be arbitrary for the moment. The backpropagation algorithm is an automatized implementation of the rules of differential calculus: for each $i = 1, \ldots, N$, we have

$$\mathrm{backprop}\, l_i(\theta) = J_{i,M}\left(g_{i,M-1} \circ \ldots \circ g_{i,1}(\theta)\right) \times J_{i,M-1}\left(g_{i,M-2} \circ \ldots \circ g_{i,1}(\theta)\right) \times \ldots \times J_{i,1}(\theta). \tag{3}$$

Famous autograd libraries such as PyTorch [26] or TensorFlow [1] implement dictionaries of functions g with their corresponding generalized Jacobians J, as well as efficient numerical implementation of the quantities defined in (3).

2.2 ReLU networks training

Our main example is based on the function $\mathrm{ReLU} \colon \mathbb{R} \to \mathbb{R}$ defined by $\mathrm{ReLU}(x) = \max\{x, 0\}$. It is differentiable save at the origin and satisfies $\mathrm{ReLU}'(x) = 0$ for $x < 0$ and $\mathrm{ReLU}'(x) = 1$ for $x > 0$.

The value of the derivative at $x = 0$ could be arbitrary in $[0, 1]$ as we have $\partial \mathrm{ReLU}(0) = [0, 1]$, where ∂ denotes the subgradient from convex analysis. Let us insist on the fact that any value within $[0, 1]$ has a variational meaning. For example PyTorch and TensorFlow use $\mathrm{ReLU}'(0) = 0$.

Throughout the paper, and following the lines of [8], we say that a function $g : \mathbb{R}^p \to \mathbb{R}^q$ is *elementary log-exp* if it can be described by a finite compositional expression involving the basic operations $+, -, \times, /$ as well as the exponential and logarithm functions, inside their domains of definition. Examples include the logistic loss $\log(1 + e^{-x})$ on $\mathbb{R}$, the multivariate Gaussian density $(2\pi)^{-K/2} \exp(-\sum_{k=1}^K x_k^2/2)$ on $\mathbb{R}^K$, and the softmax function $\left(e^{x_i} / \sum_{k=1}^K e^{x_k}\right)_{1 \leq i \leq K}$ on $\mathbb{R}^K$. The expressions x^3/x^2 and $\exp(-1/x^2)$ do not fit this definition because evaluation at $x = 0$ cannot be defined by the formula. Roughly speaking a computer evaluating an elementary $\log$-$\exp$ expression should not output any NaN error for any input.

Definition 1 (ReLU network training)**.** Assume that in (1), the function ℓ is elementary log-exp, the network f has an arbitrary structure and the functions $g_{i,j}$ in (2) are either elementary log-exp or consist in applying the ReLU function to some coordinates of their input. We then call the problem in (1) a ReLU *network training problem*. Furthermore, for any $s \in \mathbb{R}$, we denote by $\mathrm{backprop}_s$ the quantity defined in (3) when $\mathrm{ReLU}'(0) = s$ for all ReLU functions involved in the composition (2).

Other nonsmooth activation functions: The ReLU operation actually allows to express many other types of nonsmoothness such as absolute values, maxima, minima, quantiles (med for median) or soft-thresholding (st). For any $x, y, z \in \mathbb{R}$, we indeed have $|x| = \mathrm{ReLU}(2x) - x$, $2\max(x, y) = (|x - y| + x + y)$, $\min(x, y) = -\max(-x, -y)$, $\mathrm{med}(x, y, z) = \min(\max(x, y), \max(x, z), \max(y, z))$, $\mathrm{st}(x) = \mathrm{ReLU}(x - 1) - \mathrm{ReLU}(-x - 1)$.

Definition 1 is thus much more general than it may seem since it allows, for example, to express convolutional neural networks with max-pooling layers such as VGG or ResNet architectures which correspond to the models considered in the experimental section (although we do not re-program pooling using ReLU). The following theorem is due to [7], with an alternative version in [8].

Theorem 1 ($\mathrm{Backprop}$ returns a gradient a.e.)**.** *Consider a* ReLU *network training problem* (1) *as in Definition 1 and* $T \geq 1$*. Define* $S \subset \mathbb{R}^P$ *as the complement of the set*

$$\left\{\theta \in \mathbb{R}^P,\ l_i \text{ differentiable at } \theta,\ \mathrm{backprop}_s l_i(\theta) = \nabla l_i(\theta), \qquad \forall i \in \{1, \ldots, N\},\ s \in [-T, T]\right\}.$$

Then S *is contained in a finite union of embedded differentiable manifolds of dimension at most* $P - 1$ *(and in particular has Lebesgue measure zero).*

Although this theorem looks natural, this is a nontrivial result about the backpropagation algorithm that led to the introduction of conservative fields [7]. It implies that all choices for $s = \mathrm{ReLU}'(0)$ in $[0, 1] = \partial \mathrm{ReLU}(0)$ are essentially equivalent modulo a negligible set S. Perhaps more surprisingly, s can be chosen arbitrarily in $\mathbb{R}$ without breaking this essential property of $\mathrm{backprop}$. The set S is called the *bifurcation zone* throughout the manuscript. For ReLU network training problems, the bifurcation zone is a Lebesgue zero set and is actually contained locally in a finite union of smooth objects of dimension strictly smaller than the ambient dimension. This geometric result reveals a surprising link between backpropagation and Whitney stratifications, as described in [7, 8]. In any case the bifurcation zone is completely negligible. Note that the same result holds if we allow each different call to the ReLU function to use different values for $\mathrm{ReLU}'(0)$.

2.3 ReLU network training with SGD

Let $(B_k)_{k \in \mathbb{N}}$ denote a sequence of mini-batches with sizes $|B_k| \subset \{1, \ldots, N\}$ for all k and $\alpha_k > 0$ the learning rate. Given initial weights $\theta_0 \in \mathbb{R}^P$ and any parameter $s \in \mathbb{R}$, the SGD training procedure of f consists in applying the recursion

$$\theta_{k+1,s} = \theta_{k,s} - \gamma \frac{\alpha_k}{|B_k|} \sum_{n \in B_k} \mathrm{backprop}_s[\ell(f(x_n, \theta_{k,s}), y_n)], \tag{4}$$

where $\gamma > 0$ is a step-size parameter. Note that we explicitly wrote the dependency of the sequence in $s = \mathrm{ReLU}'(0)$. According to Theorem 1 if the initialization θ_0 is chosen randomly, say, uniformly in a ball, a hypercube, or with iid Gaussian entries, then with probability 1, θ_0 does not fall in the bifurcation zone S. Intuitively, since S is negligible, the odds that one of the iterates produced by the

algorithm fall on S are very low. As a consequence, varying s in the recursion (4) does not modify the sequence. This rationale is actually true for almost all values of γ. This provides a rigorous statement of the idea that the choice of $\mathrm{ReLU}'(0)$ "does not affect" neural network training. The following result is based on arguments developed in [8], see also [6] in a probabilistic context.

Theorem 2 ($\mathrm{ReLU}'(0)$ does not impact training with probability 1)**.** *Consider a* ReLU *network training problem as in Definition 1. Let* $(B_k)_{k\in\mathbb{N}}$ *be a sequence of mini-batches with* $|B_k| \subset \{1,\dots,N\}$ *for all* k *and* $\alpha_k > 0$ *the associated learning rate parameter. Choose* θ_0 *uniformly at random in a hypercube and* γ *uniformly in a bounded interval* $I \subset \mathbb{R}_+$*. Let* $s \in \mathbb{R}$*, set* $\theta_{0,s} = \theta_0$*, and consider the recursion given in* (4)*. Then, with probability one, for all* $k \in \mathbb{N}$*,* $\theta_{k,s} = \theta_{k,0}$*.*

3 Surprising experiments on a simple feedforward network

3.1 $\mathrm{ReLU}'(0)$ has an impact

Even though the ReLU activation function is non-differentiable at 0, autograd libraries such as PyTorch [26] or TensorFlow [1] implement its derivative with $\mathrm{ReLU}'(0) = 0$. What happens if one chooses $\mathrm{ReLU}'(0) = 1$? The popular answer to this question is that it should have no effect. Theorems 1 and 2 provide a formal justification which is far from being trivial.

A 32 bits MNIST experiment We ran a simple experiment to confirm this answer. We initialized two fully connected neural networks f_0 and f_1 of size $784 \times 2000 \times 128 \times 10$ with the same weights $\theta_{0,0} = \theta_{0,1} \in \mathbb{R}^P$ which are chosen at random. Using the MNIST dataset [24], we trained f_0 and f_1 with the same sequence of mini-batches $(B_k)_{k\in\mathbb{N}}$ (minibatch size 128), using the recursion in (4) for $s = 0$ and $s = 1$ and with a fixed $\alpha_k = 1$, and γ chosen uniformly at random in $[0.01, 0.012]$. At each iteration k, we computed the sum $\|\theta_{k,0} - \theta_{k,1}\|_1$ of the absolute differences between the coordinates of $\theta_{k,0}$ and $\theta_{k,1}$. As a sanity check, we actually computed $\theta_{k,0}$ a second time, denoting this by $\tilde{\theta}_{k,0}$, using a third network to control for sources of divergence in our implementation. Results are reported in Figure 1. The experiment was run using PyTorch [26] on a CPU.

ReLU'(0) has an impact First we observe no difference between $\theta_{k,0}$ and $\tilde{\theta}_{k,0}$, which shows that we have controlled all possible sources of divergence in PyTorch. Second, while no differences between $\theta_{0,0}$ and $\theta_{0,1}$ is expected (Theorem 2), we observe a sudden deviation of $\|\theta_{k,0} - \theta_{k,1}\|_1$ at iteration 65 which then increases in a smooth way. The deviation is sudden as the norm is exactly zero before iteration 65 and jumps above one after. Therefore this cannot be explained by an accumulation of small rounding errors throughout iterations, as this would produce a smooth divergence starting at the first iteration. So this suggests that there is a special event at iteration 65.

The center part of Figure 1 displays the minimal absolute value of neurons of the first hidden layer evaluated on the current mini-batch, before the application of ReLU. It turns out that at iteration 65, this minimal value is exactly 0, resulting in a drop in the center of Figure 1. This means that the divergence is actually due to an iterate of the sequence falling exactly on the bifurcation zone. According to Theorem 2, this event is so exceptional that it should never been seen.

Practice and Theory: Numerical precision vs Genericity This contradiction can be solved as follows: the minimal absolute value in Figure 1 oscillates between 10^{-6} and 10^{-8} which is roughly the value of machine precision in 32 bits float arithmetic. This machine precision value is of the order 10^{-16} in 64 bits floating arithmetic which is orders of magnitude smaller than the typical value represented in Figure 1. And indeed, performing the same experiment in 64 bits precision, the divergence of $\|\theta_{k,0} - \theta_{k,1}\|_1$ disappears and the algorithm can actually be run for many epochs without any divergence between the two sequences. This is represented in Figure 7 in Appendix B. We also report similar results using $\mathrm{ReLU6}$ [19] in place of ReLU on a similar network.

3.2 Relative volume of the bifurcation zone and relative gradient variation

The previous experiment suggests that mini-batch SGD algorithm (4) crossed the bifurcation zone:

$$S_{01} = \{\theta \in \mathbb{R}^P : \exists i \in \{1,\dots,N\},\ \mathrm{backprop}_0[l_i](\theta) \neq \mathrm{backprop}_1[l_i](\theta)\} \subset S. \tag{5}$$

This unlikely phenomenon is due to finite numerical precision arithmetic which thickens the negligible set S_{01} in proportion to the machine precision threshold. This is illustrated on the right of

Figure 1, which represents the bifurcation zone at iteration 65 in a randomly generated hyperplane (uniformly among 2 dimensional hyperplanes) centered around the weight parameters which generated the bifurcation in the previous experiment (evaluation on the same mini-batch). The white area corresponds to some entries being exactly zero, i.e., below the 32 bits machine precision threshold, before application of ReLU. On the other hand, in 64 bits precision, the same representation is much smoother and does not contain exact zeros (see Figure 7 in Appendix B). This confirms that the level of floating point arithmetic precision explains the observed divergence. We now estimate the relative volume of this bifurcation zone by Monte Carlo sampling (see Appendix C for details). All experiments are performed using PyTorch [26] on GPU.

Experimental procedure – weight randomization: We randomly generate a set of parameters $\{\theta_j\}_{j=1}^{M}$, with $M = 1000$, for a fully connected network architecture f composed of L hidden layers. Given two consecutive layers, respectively composed of m and m' neurons, the weights of the corresponding affine transform are drawn independently, uniformly in $[-\alpha, \alpha]$ where $\alpha = \sqrt{6}/\sqrt{m}$. This is the default weight initialization scheme in PyTorch (Kaiming-Uniform [17]). Given this sample of parameters, iterating on the whole MNIST dataset, we approximate the proportion of θ_j for which $\mathrm{backprop}_0(l_i)(\theta_j) \neq \mathrm{backprop}_1(l_i)(\theta_j)$ for some i, for different networks and under different conditions (see Appendix C for details).

Impact of the floating-point precision: Using a fixed architecture of three hidden layers of 256 neurons each, we empirically measured the relative volume of S_{01} using the above experimental procedure, varying the numerical precision. Table 1 reports the obtained estimates. As shown in Table 1, line 1, at 16 bits floating-point precision, all drawn weights $\{\theta_i\}_{i=1}^{M}$ fall within S_{01}. In sharp contrast, when using a 64 bits precision, none of the sampled weights belong to S_{01}. This proportion is 40% in 32 bits precision. For the rare impacted mini-batches (Table 1 line 2), the relative change in norm is above a factor 20, higher in 16 bits precision (Table 1 line 3). These results confirm that the floating-point arithmetic precision is key in explaining the impact of $\mathrm{ReLU}'(0)$ during backpropagation, impacting both frequency and magnitude of the differences.

Floating-point precision	16 bits	32 bits	64 bits
Proportion of $\{\theta_i\}_{i=1}^{M}$ in S (CI $\pm$ 5%)	100%	40%	0%
Proportion of impacted mini-batches (CI $\pm$ 13%)	0.05%	0.0002%	0%
Relative L^2 difference for impacted mini-batches (1st quartile, median, 3rd quartile)	(98, 117, 137)	(19, 25, 47)	(0, 0, 0)

Table 1: Impact of S according to the floating-point precision on a fully connected neural network ($784 \times 256 \times 256 \times 256 \times 10$) on MNIST. First line: proportion of drawn weights θ_i such that at least one mini-batch results in difference between $\mathrm{backprop}_0$ and $\mathrm{backprop}_1$. CI stands for *Confidence Interval* with 5% risk (Hoeffding CI, see Appendix C). Second line: overall proportion of mini-batch-weight vector pairs causing a difference between $\mathrm{backprop}_0$ and $\mathrm{backprop}_1$ (McDiarmid CI, see Appendix C). Third line: distribution of $\|\mathrm{backprop}_0 - \mathrm{backprop}_1\|_2 / \|\mathrm{backprop}_0\|_2$ for the affected mini-batch-weight vector pairs.

Impact of sample and mini-batch size: Given a training set $\{(x_n, y_n)\}_{n=1\ldots N}$ and a random variable $\theta \in \mathbb{R}^P$ with distribution $\mathbb{P}_\theta$, we estimate the probability that $\theta \in S_{01} \subset S$. Intuitively, this probability should increase with sample size N. We perform this estimation for a fixed architecture of 4 hidden layers composed of 256 neurons each while varying the sample size. Results are reported in Figure 2. For both the 16 and the 32 bits floating-point precisions, our estimation indeed increases with the sample size while we do not find any sampled weights in S_{01} in 64 bits precision. We also found that the influence of mini-batch size is not significative.

Impact of network size: To evaluate the impact of the network size, we carried out a similar Monte Carlo experiment, varying the depth and width of the network. Firstly, we fixed the number of hidden layers to 3. Following the same experimental procedure, we empirically estimated the relative volume of S_{01}, varying the width of the hidden layers. The results, reported in Figure 2, show that increasing the number of neurons by layer increases the probability to find a random $\theta \sim \mathbb{P}_\theta$ in S_{01} for both 16 and 32 floating-point precision. In 64 bits, even with the largest width tested (1024 neurons), no sampled weight parameter is found in S_{01}. Similarly we repeated the experiment varying the network depth and fixing, this time, the width of the layers to 256 neurons. Anew, the

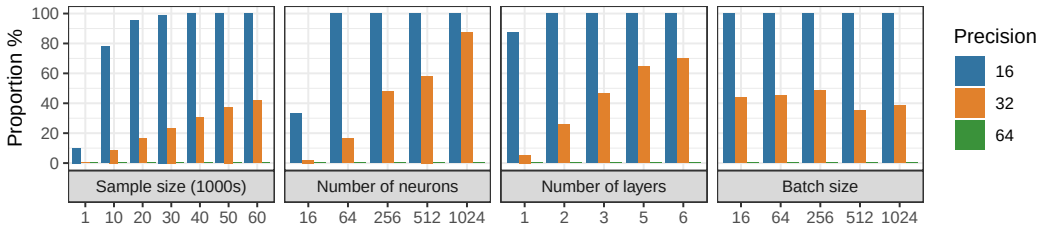

Figure 2: Influence of different size parameters on the proportion of drawn weight vector θ_i such that at least one mini-batch results in difference between $\mathrm{backprop}_0$ and $\mathrm{backprop}_1$ on MNIST dataset. Confidence interval at risk level 5% results in a variation of $\pm 5\%$ for all represented proportions. First: 4 hidden layers with 256 neurons and batch size 256, varying the size of the training set. Second: 3 hidden layer network with mini-batch size 256, varying the number of neuron per layer. Third: 256 neurons per layer and mini-batch size 256, varying the number of layers. Fourth: 3 hidden layers with 256 neurons per layer, varying mini-batch size.

results, reported in Figure 2, show that increasing the network depth increases the probability that $\theta \sim \mathbb{P}_\theta$ belongs to S_{01} for both 16 and 32 bits floating-point precision while this probability is zero in 64 bits precision. This shows that the size of the network, both in terms of number of layers and size of layers is positively related to the effect of the choice of s in backpropagation. On the other hand, the fact that neither the network depth, width, or the number of samples impact the 64 bits case suggests that, within our framework, numerical precision is the primary factor of deviation.

4 Consequences for learning

4.1 Benchmarks and implementation

Datasets and networks We further investigate the effect of the phenomenon described in Section 3 in terms of learning using the CIFAR10 dataset [23] and the VGG11 architecture [31]. To confirm our findings in alternative settings, we also use the MNIST [24], SVHN [25] and ImageNet [12] datasets, fully connected networks (3 hidden layers of size 2048), and the ResNet18 and ResNet50 architectures [18]. Additional details on the different architectures and datasets are found in Appendix D.1. By default, unless stated otherwise, we use the SGD optimizer. We also investigated the effect of batch normalization [21], dropout [32], the Adam optimizer [22] as well as numerical precision. All the experiments in this section are run in PyTorch [26] on GPU. For each training experiment presented in this section (except ImageNet experiments), we use the `optuna` library [3] to tune learning rates for each experimental condition; see also Appendix D.2.

4.2 Effect on training and test errors

We first consider training a VGG11 architecture on CIFAR10 using the SGD optimizer. For different values of $\mathrm{ReLU}'(0)$, we train this network ten times with random initializations under 32 bits arithmetic precision. The results are depicted in Figure 3. Without batch normalization, varying the value of $\mathrm{ReLU}'(0)$ beyond a magnitude of 0.1 has a detrimental effect on the test accuracy, resulting in a concave shaped curve with maximum around $\mathrm{ReLU}'(0) = 0$. On the other hand, the decrease of the training loss with the number of epochs suggests that choosing $\mathrm{ReLU}'(0) \neq 0$ induces jiggling behaviors with possible sudden jumps during training. Note that the choice $\mathrm{ReLU}'(0) = 0$ leads to a smooth decrease and that the magnitude of the jumps for other values is related to the magnitude of the chosen value. This is consistent with the interpretation that changing the value of $\mathrm{ReLU}'(0)$ has an excitatory effect on training. We observed qualitatively similar behaviors for a fully connected network on MNIST and a ResNet18 on CIFAR10 (Appendix D). Sensitivity to the magnitude of $\mathrm{ReLU}'(0)$ depends on the network architecture: our fully connected network on MNIST is less sensitive to this value than VGG11 and ResNet18. The latter shows a very high sensitivity since for values above 0.2, training becomes very unstable and almost impossible.

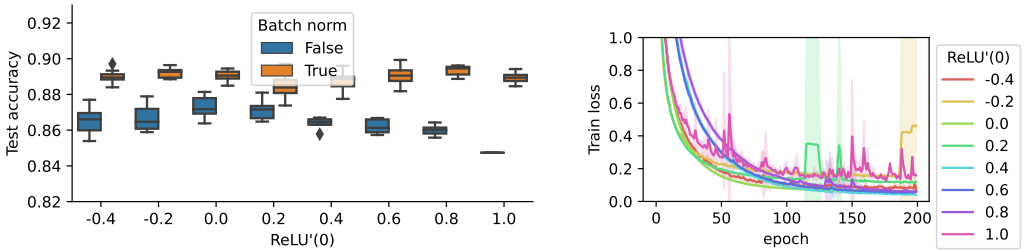

Figure 3: Training a VGG11 network on CIFAR10 with SGD. Left: test accuracy with and without batch normalization. Right: training loss without batchnorm. For each experiment we performed 10 random initializations, depicted by the boxplots on the left and the filled contours on the right (standard deviation).

These experiments complement the preliminary results obtained in Section 3. In particular choosing different values for $\mathrm{ReLU}'(0)$ has an effect, it induces a chaotic behavior during training which affects test accuracy. The default choice $\mathrm{ReLU}'(0) = 0$ seems to provide the best performances.

To conclude, we conducted four training experiments for ResNet50 on the ImageNet dataset [12] using the SGD optimizer. These were conducted with fixed learning rate, contrary to results reported above. We observe that switching from $\mathrm{ReLU}'(0) = 0$ to 1 results in a massive drop from around 75% to 63% or 55% for two runs.

4.3 Mitigating factors: numerical precision, batch-normalization and Adam

We analyze below the combined effects of the variations of $\mathrm{ReLU}'(0)$ with numerical precision or classical reconditioning methods: Adam, batch-normalization and dropout.

Batch-normalization: As represented in Figure 3, batch normalization [21] not only allows to attain higher test accuracy, but it also completely filters out the effect of the choice of the value of $\mathrm{ReLU}'(0)$, resulting in a flat shaped curve for test accuracy. This is consistent with what we observed on the training loss (Figure 12 in Appendix D.3) for which different choices of $\mathrm{ReLU}'(0)$ lead to indistinguishable training loss curves. This experiment suggests that batch normalization has a significant impact in reducing the effect of the choice of $\mathrm{ReLU}'(0)$. This observation was confirmed with a very similar behavior on the MNIST dataset with a fully connected network (Figure 9 in Appendix D.2). We could observe a similar effect on CIFAR 10 using a ResNet18 architecture (see Appendix D.5), however in this case the value of $\mathrm{ReLU}'(0)$ still has a significative impact on test error, the ResNet18 architecture being much more sensitive.

Using the Adam optimizer: The Adam optimizer [22] is among the most popular algorithms for neural network training; it combines adaptive step-size strategies with momentum. Adaptive step-size acts as diagonal preconditioners for gradient steps [22, 13] and therefore can be seen as having an effect on the loss landscape of neural network training problems. We trained a VGG11 network on both CIFAR 10 and SVHN using the Adam optimizer. The results are presented in Figure 4. We observe a qualitatively similar behavior as in the experiments of Section 4.2 but a much lower sensitivity to the magnitude of $\mathrm{ReLU}'(0)$. In other words, the use of the Adam optimizer mitigates the effect of this choice, both in terms of test errors and by buffering the magnitude of the sometimes chaotic effect induced on training loss optimization (Figure 13 in Appendix D.3).

Increasing numerical precision: As shown in Appendix D.3, using 64 bits floating precision on VGG11 with CIFAR10 cancels out the effect of $\mathrm{ReLU}'(0) = 1$, in coherence with Section 3. More specifically $\mathrm{ReLU}'(0) = 1$ in 64 bits precision obtains similar performances as $\mathrm{ReLU}'(0) = 0$ in 32 bits precision. Furthermore, the numerical precision has barely any effect when $\mathrm{ReLU}'(0) = 0$. Finally, we remark that in 16 bits with $\mathrm{ReLU}'(0) = 1$, training is extremely unstable so that we were not able to train the network in this setting.

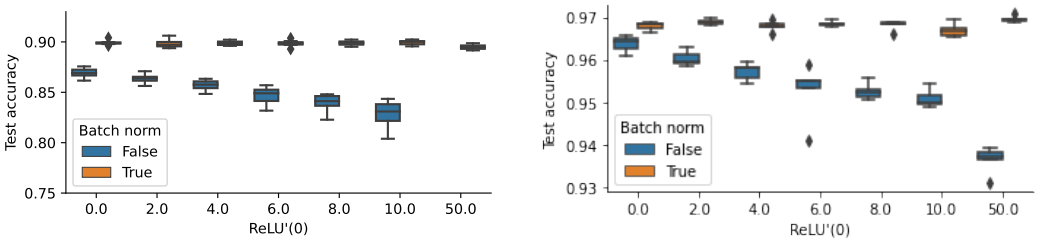

Figure 4: Similar to Fig.3, with VGG11 and Adam optimizer, on CIFAR 10 (left) and SVHN (right).

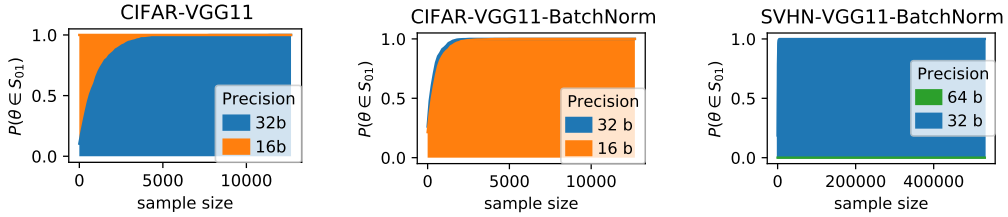

Figure 5: Monte Carlo estimation of relative volume on CIFAR10 and SVHN with VGG11 network

Combination with dropout: Dropout [32] is another algorithmic process to regularize deep neural networks. We investigated its effect when combined with varying choices of $\mathrm{ReLU}'(0)$. We used a fully connected neural network trained on the MNIST dataset with different values of dropout probability. The results are reported in Figure 11 in Appendix D.2. We did not observe any joint effect of dropout probability and magnitude of $\mathrm{ReLU}'(0)$ in this context.

4.4 Back to the bifurcation zone: more neural nets and the effects of batch-norm

The training experiments are complemented by a similar Monte Carlo estimation of the relative volume of the bifurcation zone as performed in Section 3 (same experimental setting). To avoid random outputs we force the GPU to compute convolutions deterministically. Examples of results are given in Figure 5. Additional results on fully connected network with MNIST and ResNet18 on CIFAR10 are shown in Section D. We consistently observe a high probability of finding an example on the bifurcation zone for large sample sizes. Several comments are in order.

Numerical precision: Numerical precision is the major factor in the thickening of the bifurcation zone. In comparison to 32 bits experiments, 16 bits precision dramatically increases its relative importance. We also considered 64 bits precision on SVHN, a rather large dataset. Due to the computational cost, we only drew 40 random weights and observed no bifurcation on any of the terms of the loss whatsoever. This is consistent with the experiments conducted in Section 3 and suggests that, within our framework, 64 bit precision is the main mitigating factor for our observations.

Batch normalization: In all our relative volume estimation experiments, we observe that batch normalization has a very significant effect on the proportion of examples found in the bifurcation zone. In 32 bits precision, the relative size of this zone increases with the addition of batch normalization, similar observations were made in all experiments presented in Appendix D. This is a counter-intuitive behavior as we have observed that batch normalization increases test accuracy and mitigates the effect of $\mathrm{ReLU}'(0)$. Similarly in 16 bits precision, the addition of batch normalization seems to actually decrease the size of the bifurcation zone. Batch normalization does not result in the same qualitative effect depending on arithmetic precision. These observations open many questions which will be the topic of future research.

4.5 Total number of ReLU calls during training

We consider the MNIST dataset with a fully connected ReLU network, varying the number of layers in $1, \ldots, 6$ and neuron per layers in $16, 64, 256, 512$. For 16 and 32 bits precisions, we perform 100

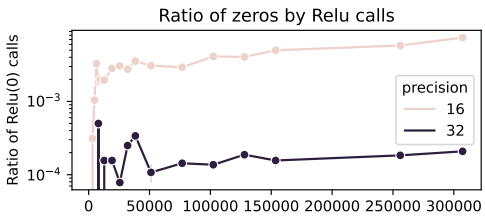

Figure 6: Proportion of $\mathrm{ReLU}(0)$ calls by total number of ReLU calls after 100 epochs for ReLU networks on MNIST with number of layers in $1, \dots, 6$ and neuron per layers in $16, 64, 256, 512$.

epochs and proportion of $\mathrm{ReLU}(0)$ over all ReLU calls during training. Figure 6 suggests that, for a given precision, the number of times the bifurcation zone is met is roughly proportional to the total number of ReLU calls during training, independently of the architecture used (number of layers, neurons per layers). This suggests that the total number of ReLU calls during training is an important factor in understanding the bifurcation phenomenon. Broader investigation of this conjecture will be a matter of future work.

5 Conclusions and future work

The starting point of our work was to determine if the choice of the value $s = \mathrm{ReLU}'(0)$ affects neural network training. Theory tells that this choice should have negligible effect. Performing a simple learning experiment, we discovered that this is false in the real world and the first purpose of this paper is to account for this empirical evidence. This contradiction between theory and practice is due to finite floating point arithmetic precision while idealized networks are analyzed theoretically within a model of exact arithmetic on the field of real numbers. Owing to the size of deep learning problems, rounding errors due to numerical precision occur at a relatively high frequency, and virtually all the time for large architectures and datasets under 32 bit arithmetic precision (the default choice for TensorFlow and PyTorch libraries).

Our second goal was to investigate the impact of the choice of $s = \mathrm{ReLU}'(0)$ in machine learning terms. In 32 bits precision it has an effect on test accuracy which seems to be the result of inducing a chaotic behavior in the course of empirical risk minimization. This was observed consistently in all our experiments. However we could not identify a systematic quantitative description of this effect; it highly depends on the dataset at hand, the network structure as well as other learning parameters such as the presence of batch normalization and the use of different optimizers. Our experiments illustrate this diversity. We observe an interesting robustness property of batch normalization and the Adam optimizer, as well as a high sensitivity to the network structure.

Overall, the goal of this work is to draw attention to an overlooked factor in machine learning and neural networks: nonsmoothness. The ReLU activation is probably the most widely used nonlinearity in this context, yet its nondifferentiability is mostly ignored. We highlight the fact that the default choice $\mathrm{ReLU}'(0) = 0$ seems to be the most robust, while different choices could potentially lead to instabilities. For a general nonsmooth nonlinearity, it is not clear *a priori* which choice would be the most robust, if any, and our investigation underlines the potential importance of this question. Our research opens new directions regarding the impact of numerical precision on neural network training, its interplay with nonsmoothness and its combined effect with other learning factors, such as network architecture, batch normalization or optimizers. The main idea is that mathematically negligible factors are not necessarily computationally negligible.

Acknowledgments and Disclosure of Funding

The authors thank anonymous referees for constructive suggestions which greatly improved the paper. The authors acknowledge the support of the DEEL project, the AI Interdisciplinary Institute ANITI funding, through the French "Investing for the Future – PIA3" program under the Grant agreement ANR-19-PI3A-0004, Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant numbers FA9550-19-1-7026, FA9550-18-1-0226, and ANR MaSDOL 19-CE23-0017-01. J. Bolte also acknowledges the support of ANR Chess, grant ANR-17-EURE-0010 and TSE-P.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[3] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[4] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18, 2018.

[5] J. Berner, D. Elbrächter, P. Grohs, and A. Jentzen. Towards a regularity theory for relu networks–chain rule and global error estimates. In *2019 13th International conference on Sampling Theory and Applications (SampTA)*, pages 1–5. IEEE, 2019.

[6] P. Bianchi, W. Hachem, and S. Schechtman. Convergence of constant step stochastic gradient descent for non-smooth non-convex functions. *arXiv preprint arXiv:2005.08513*, 2020.

[7] J. Bolte and E. Pauwels. Conservative set valued fields, automatic differentiation, stochastic gradient methods and deep learning. *Mathematical Programming*, pages 1–33, 2020.

[8] J. Bolte and E. Pauwels. A mathematical model for automatic differentiation in machine learning. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

[9] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311, 2018.

[10] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[11] M. Courbariaux, Y. Bengio, and J.-P. David. Training deep neural networks with low precision multiplications. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[13] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[14] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press Cambridge, 2016.

[15] A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[16] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pages 1737–1746. PMLR, 2015.

[17] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[18] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[19] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[20] K. Hwang and W. Sung. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6. IEEE, 2014.

[21] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[22] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations, Conference Track Proceedings*, 2015.

[23] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-10 (canadian institute for advanced research). *URL http://www. cs. toronto. edu/kriz/cifar. html*, 5, 2010.

[24] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 1998.

[25] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011.

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[27] R. T. Rockafellar and R. J.-B. Wets. *Variational analysis*, volume 317. Springer Science & Business Media, 2009.

[28] A. Rodriguez, E. Segal, E. Meiri, E. Fomenko, Y. J. Kim, H. Shen, and B. Ziv. Lower numerical precision deep learning inference and training. *Intel White Paper*, 3:1–19, 2018.

[29] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[30] S. Scholtes. *Introduction to piecewise differentiable equations*. Springer Science & Business Media, 2012.

[31] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[32] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[33] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning NIPS Workshop.*, 2011.

A Mathematical details for Section 2

In Section A.1, we provide some elements of proof for Theorems 1 and 2. In Section A.2, we explain how to check the assumptions of Definition 1 by describing the special case of fully connected ReLU networks.

A.1 Elements of proof of Theorems 1 and 2

The proof arguments were described in [7, 8]. We simply concentrate on justifying how the results described in these works apply to Definition 1 and point the relevant results leading to Theorems 1 and 2.

It can be inferred from Definition 1 that all elements in the definition of a ReLU network training problem are piecewise smooth, where each piece is an elementary $log - exp$ function. We refer the reader to [30] for an introduction to piecewise smoothness and recent use of such notions in the context of algorithmic differentiation in [8]. Let us first argue that the results of [8] apply to Definition 1.

- We start with an explicit selection representation of $\mathrm{backprop}_s\ \mathrm{ReLU}$. Fix any $s \in \mathbb{R}$ and consider the three functions $f_1 \colon x \mapsto 0$, $f_2 \colon x \mapsto x$ and $f_3 \mapsto sx$ with the selection index $t(x) = 1$ if $x < 0$, 2 if $x > 0$ and 3 if $x = 0$. We have for all x

  $$f_{t(x)} = \mathrm{ReLU}(x)$$

  Furthermore, differentiating the active selection as in [8, Definition 4] we have

  $$\hat{\nabla}^t f = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ s & \text{if } x = 0 \end{cases}$$

  and the right hand side is precisely the definition of $\mathrm{backprop}_s\ \mathrm{ReLU}$. This shows that we have a selection derivative as used in [8].
- Given a ReLU network training problem as in Definition 1, we have the following property.
  - All elements in the ReLU network training problem are piecewise elementary $log - exp$. That is each piece can be identified with an elementary $log - exp$ function. Furthermore the selection process describing the choice of active function can similarly be described by elementary $log - exp$ functions with equalities and inequalities.

Therefore, we meet the definition of $log - exp$ selection function in [8] and all corresponding results apply to any ReLU network training problem as given in Definition 1. Fix $T \geq 1$, getting back to problem (1), using [8, Definition 5] and the selection derivative described above, for each $i = 1, \ldots, N$, there is a conservative field $D_i \colon \mathbb{R}^P \rightrightarrows \mathbb{R}^P$ (definition of conservativity is given in [7] and largely described in [8]) such that for any $s \in [0, T]$, and $\theta \in \mathbb{R}^P$

$$\mathrm{backprop}_s l_i(\theta) \in D_i(\theta).$$

Using [7, Corollary 5] we have $D_i(\theta) = \{\nabla l_i(\theta)\}$ for all θ outside of a finite union of differentiable manifolds of dimension at most $P-1$. This leads to Theorem 1 for $s \in [0, T]$. Theorem 2 is deduced from the proof of [8, Theorem 7] (last paragraph of the proof) that with probability 1, for all $k \in \mathbb{N}$, for all $n = 1, \ldots, N$ and $s \in [0, T]$

$$\mathrm{backprop}_s[\ell(f(x_n, \theta_{k,s}), y_n)] = \nabla_\theta \ell(f(x_n, \theta_{k,s}), y_n)$$

since we have $\theta_{0,s} = \theta_0$ for all s, the generated sequence in (4) does not depend on $s \in [0, T]$. This is Theorem 2 for $s \in [0, T]$, note that a similar probabilistic argument was developped in [6]. We may repeat the same arguments fixing $T < 0$, so that both results actually hold for all $s \in [-T, T]$.

A.2 The special case of fully connected ReLU networks

The functions $g_{i,j}$ in the composition (2) can be described explicitly for any given neural network architecture. For the sake of clarity, we detail below the well-known case of fully connected ReLU networks for multiclass classification. We denote by $K \geq 2$ the total number of classes.

Consider any fully connected ReLU network architecture of depth H, with the softmax function applied on the last layer. We denote by d_h the size of each layer $h = 1, \dots, H$, and by d_0 the input dimension. In particular $d_H = K$ equals the number of classes. All the functions $f_\theta : \mathbb{R}^{d_0} \to \mathbb{R}^{d_H}$ represented by the network when varying the weight parameters $\theta \in \mathbb{R}^P$ are of the form:

$$f_\theta(x) = f(x, \theta) = \mathrm{softmax} \circ A_H \circ \sigma \circ A_{H-1} \circ \cdots \sigma \circ A_1(x) \ ,$$

where each mapping $A_h : \mathbb{R}^{d_{h-1}} \to \mathbb{R}^{d_h}$ is affine (i.e., of the form $A_h(z) = W_h z + b_h$), where $\sigma(u) = \big(\mathrm{ReLU}(u_i)\big)_i$ applies the ReLU function component-wise to any vector u, and where $\mathrm{softmax}(z) = \big(e^{z_i} / \sum_{k=1}^{d_H} e^{z_k}\big)_{1 \le i \le d_H}$ for any $z \in \mathbb{R}^{d_H}$. The weight parameters $\theta \in \mathbb{R}^P$ correspond to stacking all weight matrices W_h and biases b_h in a single vector (in particular, we have here $P = \sum_{h=1}^{H} d_h(d_{h-1} + 1)$). In the sequel, we set $P_h = \sum_{j=h}^{H} d_j(d_{j-1} + 1)$ and write $\theta_{h:H} \in \mathbb{R}^{P_h}$ for the vector of all parameters involved from layer h to the last layer H. We also write $\mathrm{concatenate}(x_1, \dots, x_r)$ to denote the vector obtained by concatenating any r vectors $x_1, \dots, x_r$. In particular, we have $\theta_{h:H} = \mathrm{concatenate}(W_h, b_h, \theta_{h+1:H})$.

Note that the decomposition above took x as input, not θ. We now explain how to construct the $g_{i,j}$ in (2). For each $i = 1, \dots, N$, the function $\theta \in \mathbb{R}^P \mapsto f(x_i, \theta)$ can be decomposed as

$$f(x_i, \theta) = \mathrm{softmax} \circ g_{i,2H-1} \circ \ldots \circ g_{i,2} \circ g_{i,1}(\theta) \ , \tag{6}$$

where, roughly speaking, the $g_{i,2h-1}$ apply the affine mapping A_h to the output $z_{h-1} \in \mathbb{R}^{d_{h-1}}$ of layer $h-1$ and pass forward all parameters $\theta_{h+1:H} \in \mathbb{R}^{P_{h+1}}$ to be used in the next layers, while the $g_{i,2h}$ apply the ReLU function to the first d_h coordinates. More formally, $g_{i,1} : \mathbb{R}^P \to \mathbb{R}^{d_1 + P_2}$ is given by

$$g_{i,1}(\theta) = \mathrm{concatenate}(W_1 x_i + b_1, \theta_{2:H}) \ ,$$

$g_{i,2} : \mathbb{R}^{d_1 + P_2} \to \mathbb{R}^{d_1 + P_2}$ maps any $(z_1, \theta_{2:H}) \in \mathbb{R}^{d_1} \times \mathbb{R}^{P_2}$ to

$$g_{i,2}(z_1, \theta_{2:H}) = \mathrm{concatenate}\big(\sigma(z_1), \theta_{2:H}\big)$$

and, for each layer $h = 2, \dots, H$, the functions $g_{i,2h-1} : \mathbb{R}^{d_{h-1} + P_h} \to \mathbb{R}^{d_h + P_{h+1}}$ and $g_{i,2h} : \mathbb{R}^{d_h + P_{h+1}} \to \mathbb{R}^{d_h + P_{h+1}}$ are given by

$$g_{i,2h-1}(z_{h-1}, \theta_{h:H}) = \mathrm{concatenate}(W_h z_{h-1} + b_h, \theta_{h+1:H})$$

and

$$g_{i,2h}(z_h, \theta_{h+1:H}) = \mathrm{concatenate}\big(\sigma(z_h), \theta_{h+1:H}\big) \qquad (\text{for } h < H).$$

Consider now the cross-entropy loss function $\ell : \Delta(K) \times \{1, \dots, K\} \to \mathbb{R}_+$ which compares any probability vector $q \in \Delta(K)$ of size K (with non-zero coordinates $q_i > 0$) with any true label $y \in \{1, \dots, K\}$, given by

$$\ell(q, y) = -\log q(y) \ .$$

Finally, using (6), the functions $l_i : \mathbb{R}^P \to \mathbb{R}$ appearing in (1)-(2) can be decomposed as

$$l_i(\theta) = \ell\big(f(x_i, \theta), y_i\big) = \big(q \in \Delta(K) \mapsto \ell(q, y_i)\big) \circ \mathrm{softmax} \circ g_{i,2H-1} \circ \ldots \circ g_{i,2} \circ g_{i,1}(\theta) \ .$$

The last decomposition satisfies (2) with $M = 2H + 1$. Since ℓ is the cross-entropy loss function, all M functions involved in this decomposition are either elementary log-exp or consist in applying ReLU to some coordinates of their input, and they are all locally Lipschitz, as required in Definition 1. This provides an explicit description of fully connected ReLU network and a similar description can be done for all architectures studied in this work.

B First experiment in 64 bits precision, and using a different activation

The code and results associated with all experiments presented in this work are publicly available here: `https://github.com/deel-ai/relu-prime`.

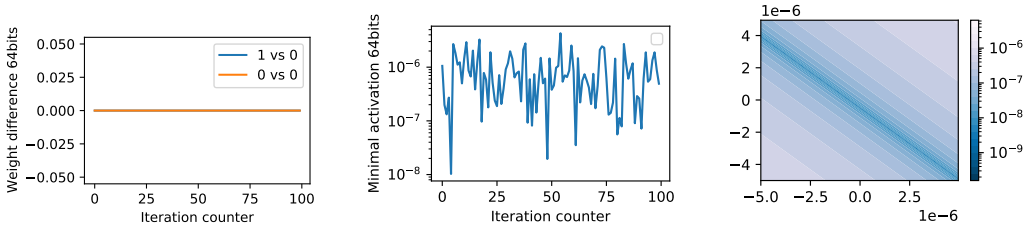

Figure 7: Same experiment as Figure 1 in 64 bits precision. Left: Difference between network parameters (L^1 norm), 100 iterations within an epoch. "0 vs 0" indicates $\|\theta_{k,0} - \tilde{\theta}_{k,0}\|_1$ where $\tilde{\theta}_{k,0}$ is a second run for sanity check, "0 vs 1" indicates $\|\theta_{k,0} - \theta_{k,1}\|_1$. Center: minimal absolute activation of the hidden layers within the k-th mini-batch, before ReLU. At iteration 65, there is no jump on the left and no drop in the center anymore. Right: illustration of the bifurcation zone at iteration $k = 65$ (same weight parameter plane as in Figure 1, but in 64 bits precision). The quantity represented is the absolute value of the neuron of the first hidden layer which was exactly zero in 32 bits (see Figure 1) before application of ReLU. Exact zeros are represented in white.

64 bits precision. We reproduce the same bifurcation experiment as in Section 3 under 64 bits arithmetic precision. The results are represented in Figure 7 which is to be compared with its 32 bits counterpart in Figure 1. As mentioned in the main text, the bifurcation does not occur anymore. Indeed the magnitude of the smallest activation before application of ReLU is of the same order, but this time it is well above machine precision which is around 10^{-16}. When depicting the same neighborhood as in Figure 1, the effect of numerical error completely disappears, the bifurcation zone being reduced to a segment in the picture, which is consistent with Theorems 1 and 2.

ReLU6 activation. We conducted the same experiment with the $\mathrm{ReLU6}$ activation function in place of ReLU and found similar results on a slightly larger network $(754, 4000, 256)$. Recall that $\mathrm{ReLU6}$ is equal to ReLU for $x < 6$ and equal to 6 for $x \geq 6$ and the default choice of derivatives at non differentiable points are zero. The illustration is given in Figure 8.

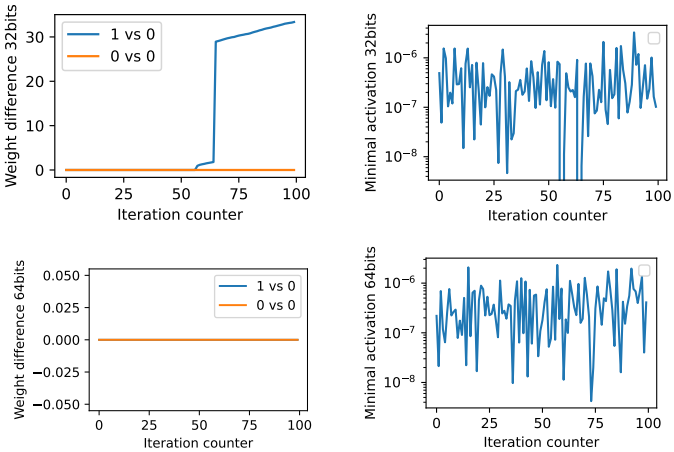

Figure 8: Same experiment as Figure 1 with $\mathrm{ReLU6}$ in place of ReLU. Top: 32 bits weight difference and minimal activation before application of $\mathrm{ReLU6}$. Bottom: 64 bits weight difference and minimal activation before application of $\mathrm{ReLU6}$

C Details on Monte Carlo sampling in Table 1

The code and results associated with all experiments presented in this work are publicly available here: `https://github.com/deel-ai/relu-prime`.

Recall that we want to estimate the relative volume of the set

$$S_{01} = \{\theta \in \mathbb{R}^P : \exists i \in \{1, \dots, N\}, \ \mathrm{backprop}_0[l_i](\theta) \neq \mathrm{backprop}_1[l_i](\theta)\} \subset S.$$

by Monte Carlo sampling. We randomly generate a set of parameters $\{\theta_j\}_{j=1}^M$, with $M = 1000$, for a fully connected network architecture f composed of L hidden layers using Kaiming-Uniform [17] random weight generator. Given this sample of parameters, iterating on the whole MNIST dataset, we approximate the proportion of θ_j for which $\mathrm{backprop}_0(l_i)(\theta_j) \neq \mathrm{backprop}_1(l_i)(\theta_j)$ for some i, for different networks and under different conditions. More precisely, denoting by Q the number of mini-batches considered in the MNIST dataset, and by $B_q \subset \{1, \dots N\}$ the indices corresponding to the mini-batch q, for $q = 1, \dots Q$, the first line of Table 1 is given by the formula

$$\frac{1}{M} \sum_{m=1}^{M} \mathbb{I} \left(\exists q \in \{1, \dots, Q\}, \ \mathrm{backprop}_0 \left[\sum_{j \in B_q} l_j(\theta_m) \right] \neq \mathrm{backprop}_1 \left[\sum_{j \in B_q} l_j(\theta_m) \right] \right),$$

where the function $\mathbb{I}$ takes value 1 or 0 depending on the validity of the statement in its argument. Similarly, the second line of Table 1 is given by the formula

$$\frac{1}{MQ} \sum_{m=1}^{M} \sum_{q=1}^{Q} \mathbb{I} \left(\mathrm{backprop}_0 \left[\sum_{j \in B_q} l_j(\theta_m) \right] \neq \mathrm{backprop}_1 \left[\sum_{j \in B_q} l_j(\theta_m) \right] \right),$$

while the last line provides statistics of the quantity

$$\frac{\left\| \mathrm{backprop}_0 \left[\sum_{j \in B_q} l_j(\theta_m) \right] - \mathrm{backprop}_1 \left[\sum_{j \in B_q} l_j(\theta_m) \right] \right\|}{\left\| \mathrm{backprop}_0 \left[\sum_{j \in B_q} l_j(\theta_m) \right] \right\|},$$

conditioned on q, m being such that $\mathrm{backprop}_0 \left[\sum_{j \in B_q} l_j(\theta_m) \right] \neq \mathrm{backprop}_1 \left[\sum_{j \in B_q} l_j(\theta_m) \right]$.

The error margin associated with the confidence interval on the first line of Table 1 is computed using Hoeffding's inequality at risk level 5%. It is given by the formula

$$\sqrt{\frac{\ln\left(\frac{2}{0.05}\right)}{2M}}.$$

As for the confidence interval of the second line of Table 1, we use the bounded differences inequality (a.k.a. McDiarmid's inequality) at risk level 5%. The error margin is given by the formula

$$\sqrt{\frac{1}{2}\left(\frac{1}{M} + \frac{1}{Q}\right) \ln\left(\frac{2}{0.05}\right)}.$$

D Complements on experiments

The code and results associated with all experiments presented in this work are publicly available here: `https://github.com/deel-ai/relu-prime`.

D.1 Benchmark datasets and architectures

Overview of the datasets used in this work. These are image classification benchmarks, the corresponding references are respectively [24, 23, 25].

Dataset	Dimensionality	Training set	Test set
MNIST	28×28 (grayscale)	60K	10K
CIFAR10	32×32 (color)	60K	10K
SVHN	32×32 (color)	600K	26K
ImageNet	224×224 (color)	1300K	50K

Overview of the neural network architectures used in this work. The corresponding references are respectively [32, 31, 18].

Name	Type	Layers	Loss function
Fully connected	fully connected	4	Cross-entropy
VGG11	convolutional	9	Cross-entropy
ResNet18	convolutional	18	Cross-entropy
ResNet50	convolutional	50	Cross-entropy

Fully connected architecture: This architecture corresponds to the one used in [32]. We only trained this network on MNIST, the resulting architecture has an input layer of size 784, three hidden layers of size 2048 and the ouput layer is of size 10.

VGG11 architecture: We used the implementation proposed in the following repository `https://github.com/kuangliu/pytorch-cifar.git` which adapts the VGG11 implementation of the module `torchvision.models` for training on CIFAR10. The only modification compared to the standard implementation is the fully connected last layers which only consist in a linear 512×10 layer. When adding batch normalization layers, it takes place after each convolutional layer.

ResNet18 architecture: We use PyTorch implementation for this architecture found in the module `torchvision.models`. We only modified the size of the output layer (10 vs 1000), the size of the kernel in the first convolutional layer (3 vs 7) and replaced batch normalization layers by the identity (when we did not use batch normalization).

D.2 Additional Experiments with MNIST and fully connected networks

We conducted the same experiments as in Section 4.2 with a fully connected 784-2048-2048-2048-10 network on MNIST. The results are represented in Figure 9 which parallels the results in Figure 3 on VGG11 with CIFAR10. We observe a similar qualitative behavior, but the fully connected architecture is less sensitive to the magnitude chosen for $\mathrm{ReLU}'(0)$. Note that in this case, learning rate tuning with optuna [3] induces a lot of spurious variability. Indeed, the same experiment with fixed learning rate results in a much smoother bell shape in Figure 10.

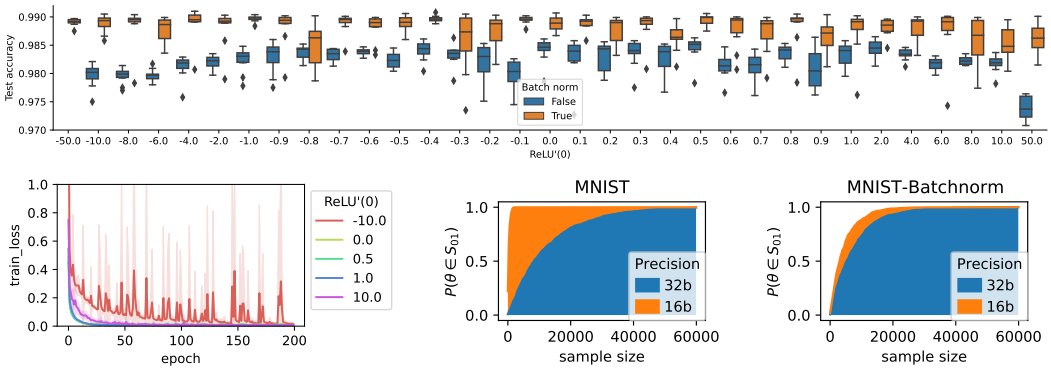

Figure 9: Top: Test error on MNIST with a fully connected 784-2048-2048-2048-10 network. The boxplots and shaded areas represent variation over ten random initializations. We recover the bell shaped curve, but the sensitivity to $\mathrm{ReLU}'(0)$ is less important. Bottom left: corresponding training loss, higher magnitude of $\mathrm{ReLU}'(0)$ induces chaotic oscillation explaining the decrease in test accuracy. Bottom center and right: relative volume estimation of the bifurcation zone without and with batch normalization. Batch normalization increases the size of the bifurcation zone with 32 bits arithmetic and decreases it under 16 bits arithmetic precision.

We investigated further the effect of combining different choices of $\mathrm{ReLU}'(0)$ with dropout [32]. Dropout is another algorithmic way to regularize deep networks and it was natural to wonder if it could have a similar effect as batch normalization. Using the same network, we combined different choices of dropout probability with different choices of $\mathrm{ReLU}'(0)$. The results are represented in Figure 11 and suggests that dropout has no conjoint effect.

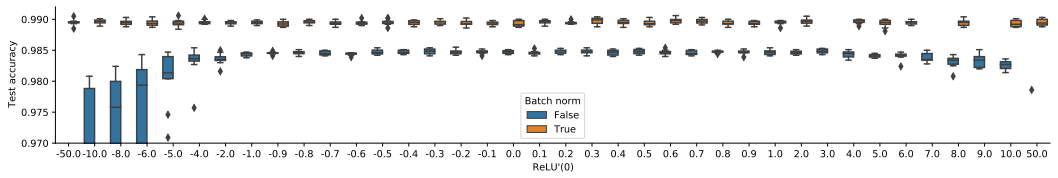

Figure 10: Same experiment as in Figure 9 without learning rate tuning.

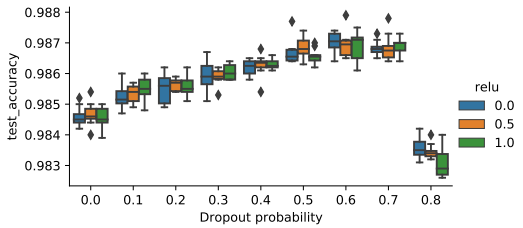

Figure 11: Experiment on combination of the choice of $\mathrm{ReLU}'(0)$ with dropout on MNIST with a fully connected 784-2048-2048-2048-10 network. The boxplots represent 10 random initializations.

D.3 Additional experiments with VGG11

This section complements Sections 4.2 and 4.3, with additional experiments with VGG11.

Batch normalization. As suggested by the experiment shown in Section 4.3, batch normalization stabilizes the choice of $\mathrm{ReLU}'(0)$, leading to higher test performances. We display in Figure 12 the decrease of training loss on CIFAR 10 and SVHN, for VGG11 with batch normalization. We see that the choice of $\mathrm{ReLU}'(0)$ has no impact and that the chaotic oscillations induced by this choice have completely disappeared.

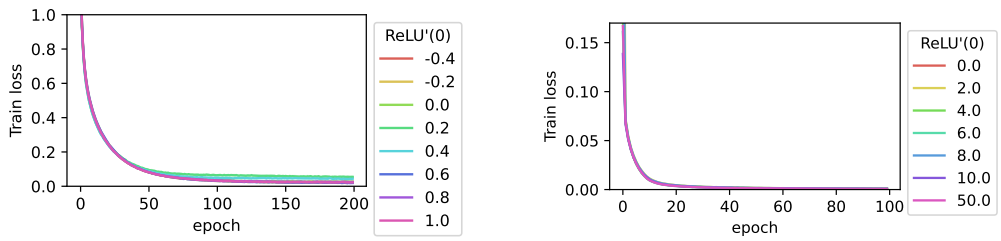

Figure 12: Training loss on CIFAR10 with VGG11 (left) and SVHN with VGG11 (right). The instability induced by the choice of $\mathrm{ReLU}'(0)$ completely disappears with batch normalization.

Adam optimizer. The training curves corresponding to Figure 4 are shown in Figure 13. They suggest that the Adam optimizer features much less sensitivity than SGD to the choice of $\mathrm{ReLU}'(0)$. This is seen with a relatively efficient buffering effect on the induced oscillatory behavior on training loss decrease.

Numerical precision. For this neural network we investigated the joint effect of $\mathrm{ReLU}'(0)$ and numerical precision (16, 32 or 64 bits). The results are displayed in Figure 14. The choice $\mathrm{ReLU}'(0) = 1$ leads to such a high instability in 16 bits precision that we were not able to tune the learning rate to train the network without explosion of the weights. In 32 bits, a few experiments resulted in non-convergent training—these were removed. We observe first that for $\mathrm{ReLU}'(0) = 0$ numerical precision has barely any effect while for $\mathrm{ReLU}'(0) = 1$ it leads to an increase in test accuracy. Furthermore, we observe that $\mathrm{ReLU}'(0) = 1$ with 64 bits precision leads to the same test accuracy as $\mathrm{ReLU}'(0) = 0$ in 32 bits precision.

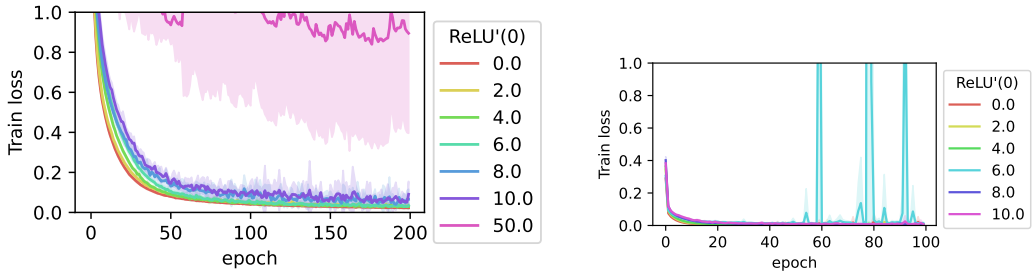

Figure 13: Training losses on CIFAR10 (left) and SVHN (right) on VGG network trained with Adam optimizer. The filled area represent standard deviation over ten random initializations.

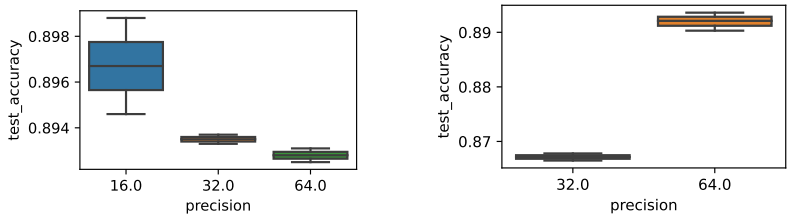

Figure 14: Test accuracy for different numerical precisions with a VGG11 network on CIFAR10. Left: $\mathrm{ReLU}'(0) = 0$. Right: $\mathrm{ReLU}'(0) = 1$.

D.4 Additional experiments with ResNet18

We performed the same experiments as the ones described in Section 4 using a ResNet18 architecture trained on CIFAR 10. The test error, training loss evolution with or without batch normalization are represented in Figure 15. We have similar qualitative observations as with VGG11. We note that the ResNet18 architecture is much more sensitive to the choice of $\mathrm{ReLU}'(0)$:

- Test performances degrade very fast. Actually, beyond a magnitude of 0.2, we could not manage to train the network without using batch normalization.
- Even when using batch normalization, the choice of $\mathrm{ReLU}'(0)$ seems to have an effect for relatively small variations. This is qualitatively different from what we observed with VGG11 and fully connected architectures.

Similar Monte Carlo relative volume experiments were carried out for this network architecture; the results are presented in Figure 17. The results are qualitatively similar to what we observed for the VGG11 architecture: the bifurcation zone is met very often for 16 bits precision, and the addition of batch normalization increases this frequency in 32 bits precision. Note that we did not observe a significant variation in 16 bits precision.

D.5 Additional experiments with ResNet50 on ImageNet

E Complimentary information, total amount of compute and resources used

All the experiments were run on a 2080ti GPU. The code corresponding to the experiments and experiments results are available at `https://github.com/deel-ai/relu-prime` Details about each test accuracy experiments are reported on Table 2. CIFAR10 is released under MIT license, MNIST, SVHN and R are released under GNU general public license, ImageNet is released under BSD license Numpy and pytorch are released under BSD license, python is released under the python sofware fondation license.

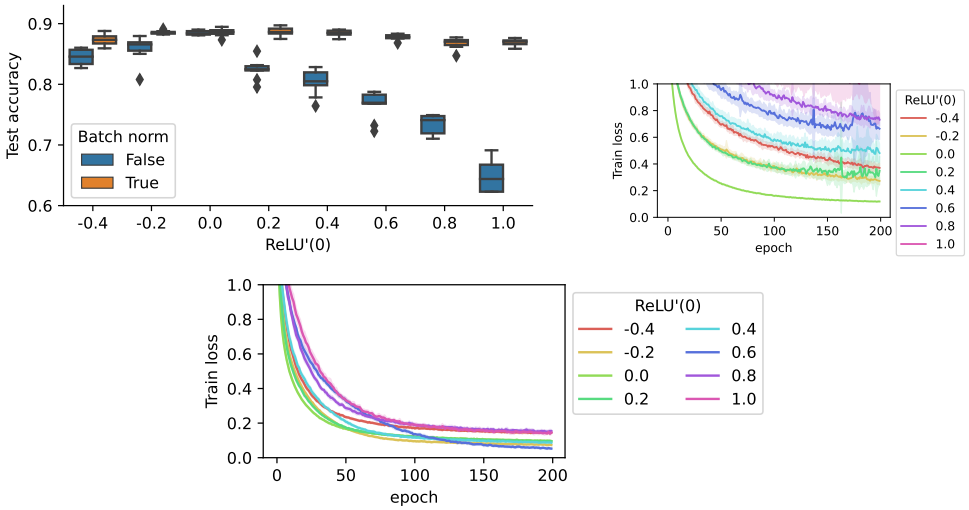

Figure 15: Training experiment on CIFAR10 with Resnet18 and the SGD optimizer. Top left: test accuracy with and without batch normalization. Top right: training loss during training without batch normalization. Bottom: training loss during training with batch normalization.

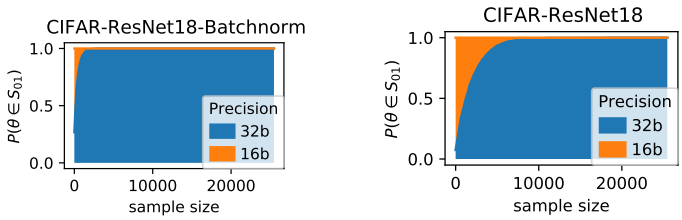

Figure 16: Relative volume Monte Carlo estimation on CIFAR10 with Resnet18 with and without batch normalization under 16 bits or 32 bits precision.

Dataset	Network	Optimizer	Batch size	Epochs	Time by epoch	Repetitions
CIFAR10	VGG11	SGD	128	200	9 seconds	10 times
CIFAR10	VGG11	Adam	128	200	10 seconds	10 times
CIFAR10	ResNet18	SGD	128	200	13 seconds	10 times
SVHN	VGG11	Adam	128	64	85 seconds	10 times
MNIST	MLP	SGD	128	200	2 seconds	10 times

Table 2: Experimental setup

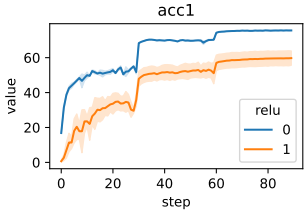

Figure 17: Test accuracy during training of a Resnet50 on ImageNet with SGD. The shaded area represents two runs. We can see a massive drop in test accuracy with $\mathrm{ReLU}'(0) = 1$.