ISSN 1749-9097

Accurately Computing the Log-Sum-Exp and Softmax Functions†

PIERRE BLANCHARD
Department of Mathematics, University of Manchester, Manchester, M13 9PL, UK
(pierre.blanchard00@gmail.com)
DESMOND J. HIGHAM
School of Mathematics, University of Edinburgh, Edinburgh, EH9 3FD, UK
(d.j.higham@ed.ac.uk)
NICHOLAS J. HIGHAM
Department of Mathematics, University of Manchester, Manchester, M13 9PL, UK
(nick.higham@manchester.ac.uk)

Evaluating the log-sum-exp function or the softmax function is a key step in many modern data science algorithms, notably in inference and classification. Because of the exponentials that these functions contain, the evaluation is prone to overflow and underflow, especially in low precision arithmetic. Software implementations commonly use alternative formulas that avoid overflow and reduce the chance of harmful underflow, employing a shift or another rewriting. Although mathematically equivalent, these variants behave differently in floating-point arithmetic and shifting can introduce subtractive cancellation. We give rounding error analyses of different evaluation algorithms and interpret the error bounds using condition numbers for the functions. We conclude, based on the analysis and numerical experiments, that the shifted formulas are of similar accuracy to the unshifted ones, so can safely be used, but that a division-free variant of softmax can suffer from loss of accuracy.

Keywords: log-sum-exp, softmax, floating-point arithmetic, rounding error analysis, overflow, underflow, condition number

1. Introduction

In many applications, especially in a wide range of machine learning classifiers such as multinomial linear regression and naive Bayes classifiers (Calafiore *et al.*, 2019), (Murphy, 2012), (Williams & Barber, 1998), one needs to compute an expression of the form

$$y = f(x) = \log \sum_{i=1}^{n} \mathrm{e}^{x_i}, \tag{1.1}$$

where $x = [x_1, x_2, \dots, x_n]^T \in \mathbb{R}^n$ and log is the natural logarithm. The function $f : \mathbb{R}^n \to \mathbb{R}$ is often referred to as log-sum-exp or LSE. Its gradient $g : \mathbb{R}^n \to \mathbb{R}^n$, given by

$$g_j(x) = \frac{\partial}{\partial x_j} f(x) = \frac{\mathrm{e}^{x_j}}{\sum_{i=1}^{n} \mathrm{e}^{x_i}}, \quad j = 1\colon n, \tag{1.2}$$

is called softmax and is also a key function in classification algorithms (Efron & Hastie, 2016, p. 355), (Goodfellow *et al.*, 2016, p. 78), (Higham & Higham, 2019). It is often the case that log-sum-exp and

† Version of May 15, 2020. **Funding**: This work was supported by Engineering and Physical Sciences Research Council grant EP/P020720/1, The MathWorks, and the Royal Society. The opinions and views expressed in this publication are those of the authors, and not necessarily those of the funding bodies.

softmax are required simultaneously.

The most obvious danger in evaluating (1.1) and (1.2) is overflow. We are interested in IEEE arithmetic in the precisions half (fp16), single (fp32), and double (fp64) (IEEE, 2019), as well as the bfloat16 half precision format (Intel Corporation, 2018). Table 1.1 shows the key parameters of interest for these precisions: the unit roundoff u, the largest finite number $r_{\max}$, and the smallest positive normalized and subnormal floating-point numbers. If some x_i exceeds the relevant $\log r_{\max}$ value in Table 1.2 then overflow will occur in evaluating e^{x_i}. Clearly, overflow is possible even for quite modestly sized x, especially for half and single precision.

Underflow is also possible. For example, for $n = 1$, if x_1 is a finite floating-point number with $x_1 < \log r_{\min}^{(s)}/2$ then[1] $\mathrm{fl}(f(x_1)) = \mathrm{fl}(\log(\mathrm{fl}(e^{x_1}))) = \mathrm{fl}(\log 0) = -\infty$ with round to nearest, whereas $f(x_1) = x_1$. For $n > 1$, underflow is damaging when all the x_i are less than $\log r_{\min}^{(s)}$. As well as avoiding harmful underflow, it is desirable to avoid generating subnormal numbers, which incur a performance penalty if handled in software[2]; see (Higham, 2002) or (Muller *et al.*, 2018) for details of subnormal numbers.

A way to avoid overflow, and to attempt to avoid harmful underflow and subnormal numbers, in evaluating log-sum-exp is to rewrite

$$y = \log \sum_{i=1}^{n} e^{x_i} = \log \sum_{i=1}^{n} e^{a} e^{x_i - a} = \log \left(e^{a} \sum_{i=1}^{n} e^{x_i - a} \right).$$

Hence

$$y = a + \log \sum_{i=1}^{n} e^{x_i - a}. \tag{1.3}$$

We note that (1.3) remains valid for complex x_i with log the principal logarithm (the one whose imaginary part lies in $(-\pi, \pi]$), provided that $a \in \mathbb{R}$ (Aprahamian & Higham, 2014, Lem. 2.5). The softmax function can be expressed in a related form:

$$g_j = \frac{e^{x_j - a}}{\sum_{i=1}^{n} e^{x_i - a}}, \quad j = 1\colon n. \tag{1.4}$$

This shifting, typically with $a = \max_i x_i$, is a well known way to attempt to avoid overflow and underflow in the evaluation of f and g, described in many places, including on Wikipedia[3], in blog posts[4], and even in a YouTube video[5]. The functions `logsumexp` in SciPy 1.3.1 (Jones *et al.*, 2001–) and `LogSumExp` in R (Team, n.d.) both implement (1.3) with $a = \max_i x_i$. The function `softmax` in the MATLAB Deep Learning Toolbox (R2019b) (Deep Learning Toolbox, n.d.) uses (1.4) with $a = \max_i x_i$.

An alternative to (1.4), which removes the denominator of (1.2) by rewriting it as e^{y} and moving it into the numerator, is

$$g_j = \exp\left(x_j - \log \sum_{i=1}^{n} e^{x_i} \right). \tag{1.5}$$

[1] $\log 0 = -\infty$ is the value recommended by the IEEE standard (IEEE Computer Society, 2008, p. 43).

[2] `https://devblogs.nvidia.com/cuda-pro-tip-flush-denormals-confidence/`, `https://en.wikipedia.org/wiki/Denormal_number`.

[3] `https://en.wikipedia.org/wiki/LogSumExp`

[4] For example, `https://hips.seas.harvard.edu/blog/2013/01/09/computing-log-sum-exp/`, `http://bayesjumping.net/log-sum-exp-trick/`, and `https://jblevins.org/log/log-sum-exp`. And similarly for the softmax: `https://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/`.

[5] `https://youtu.be/-RVM21Voo7Q`

Table 1.1. Parameters for bfloat16 and IEEE fp16, fp32, and fp64 arithmetics, to three significant figures: unit roundoff u, smallest positive (subnormal) number $r_{\min}^{(s)}$, smallest positive normalized number $r_{\min}$, and largest finite number $r_{\max}$. In Intel's bfloat16 specification, subnormal numbers are not supported, so $r_{\min}^{(s)} = r_{\min}$ (Intel Corporation, 2018).

	u	$r_{\min}^{(s)}$	$r_{\min}$	$r_{\max}$
bfloat16	3.91×10^{-3}	9.18×10^{-41}	1.18×10^{-38}	3.39×10^{38}
fp16	4.88×10^{-4}	5.96×10^{-8}	6.10×10^{-5}	6.55×10^{4}
fp32	5.96×10^{-8}	1.40×10^{-45}	1.18×10^{-38}	3.40×10^{38}
fp64	1.11×10^{-16}	4.94×10^{-324}	2.22×10^{-308}	1.80×10^{308}

Table 1.2. Logarithms of key parameters in Table 1.1, to three significant figures.

	$\log r_{\min}^{(s)}$	$\log r_{\min}$	$\log r_{\max}$
bfloat16	-92.2	-87.3	88.7
fp16	-16.6	-9.70	11.0
fp32	-103	-87.3	88.7
fp64	-744	-708	710

The conciseness of this division-free formula makes it attractive for implementing softmax when a log-sum-exp function is available. This formula is used in the SciPy 1.4.1 function `softmax`, in a MATLAB toolbox (Matlab Code for Machine Learning Algorithms in Book PRML, n.d.) associated with the book Bishop (2006), in the internal function `softmax` in the MATLAB Statistics and Machine Learning Toolbox (R2019b) (Statistics and Machine Learning Toolbox, n.d.), and in Wang *et al.* (2018); in each case the log-sum-exp term is computed by (1.3) with $a = \max_i x_i$. The formula (1.5) can also be found in codes posted in online communities such as Stack Exchange.

Because of the importance of the log-sum-exp and softmax functions, great efforts are made to optimize their implementations in software (Czaja *et al.*, 2019) and hardware (Wang *et al.*, 2018). Yet we are not aware of any investigation of the accuracy of the different formulas in floating-point arithmetic, and indeed the accuracy properties are not clear. In particular, when $a = \max_i x_i < 0$, y in (1.3) is computed as a sum of two terms of opposite sign, so there could potentially be damaging subtractive cancellation that appears not to be present in (1.1). We note that Guo *et al.* (2020, sec. 4.3) limit the size of a in (1.4), stating that the shift causes loss of accuracy by up to a factor e^a.

In this work we carry out a rounding error analysis of the unshifted and shifted formulas and (1.5) in order to determine which choices of formulas give the best accuracy and reliability. We relate the error bounds to the conditioning of f and g. We show that the shifted formulas have broadly similar error bounds to the unshifted ones, and so are entirely appropriate for practical use. We find, however, that the alternative softmax formula (1.5) has a less favorable error bound than the shifted formula and tends to produce larger errors in practice.

We begin, in the next section, by investigating the conditioning of the log-sum-exp and softmax functions. In section 3 we give detailed rounding error analyses of the basic formulas. In section 4 we analyze the shifted formulas and (1.5) and compare their error bounds with those for unshifted formulas. Numerical experiments are given in section 5 to test the accuracy of the evaluations and also to examine how the sum of the computed softmax vector entries compares with the exact value 1. Conclusions are given in section 6.

From now on we write

$$x_{\max} = \max_i x_i, \quad x_{\min} = \min_i x_i. \tag{1.6}$$

We will use the standard model of floating-point arithmetic (Higham, 2002, sec. 2.2)

$$\mathrm{fl}(a \operatorname{op} b) = (a \operatorname{op} b)(1+\delta), \quad |\delta| \leqslant u, \quad \operatorname{op} \in \{+,-,\times,/\}. \tag{1.7}$$

2. Condition numbers and Forward Stability

Before considering algorithms for computing log-sum-exp and softmax we investigate the conditioning of these functions, that is, the sensitivity of $f(x)$ and $g(x)$ in (1.1) and (1.2) to small perturbations in x.

We define the condition number of f in the usual way (see, e.g., (Higham, 2008, chap. 3)), by

$$\operatorname{cond}(f,x) := \lim_{\varepsilon \to 0} \sup_{\|\Delta x\| \leqslant \varepsilon \|x\|} \frac{|f(x+\Delta x) - f(x)|}{\varepsilon |f(x)|}.$$

This definition implies that

$$\frac{|f(x+\Delta x) - f(x)|}{|f(x)|} \leqslant \operatorname{cond}(f,x) \frac{\|\Delta x\|}{\|x\|} + o(\|\Delta x\|), \tag{2.1}$$

so that $\operatorname{cond}(f,x)$ measures the worst-case relative change in f corresponding to a small relative change in x. It is easy to show that for the ∞-norm,

$$\operatorname{cond}_\infty(f,x) = \frac{\|\nabla f(x)\|_1 \|x\|_\infty}{|f(x)|} = \frac{\|x\|_\infty}{|f(x)|} = \frac{\max_i |x_i|}{|\log \sum_i \mathrm{e}^{x_i}|}, \tag{2.2}$$

since $\|\nabla f(x)\|_1 = 1$ by (1.2).

We identify two extreme cases. First, the condition number is infinite when $x_i = -\log n$ for all i, because $f(x) = 0$. Hence when $x_i \approx -\log n$ for all i the condition number must be large. Second, $|f(x)| \geqslant \max_i x_i$, so if $\max_i x_i = \max_i |x_i|$ then $\operatorname{cond}_\infty(f,x) \leqslant 1$ and the problem is perfectly conditioned.

A forward stable algorithm for computing log-sum-exp is one for which the relative error of the computed result is bounded by $p(n) \operatorname{cond}(f,x) u$, for some low degree polynomial p. Ideally, we would like the algorithm that we use to be forward stable. To see whether it is reasonable to expect forward stability, consider the case $n = 1$. Then $y = f(x) = \log \mathrm{e}^x = x$, so $\operatorname{cond}(f,x) = 1$: the problem is perfectly conditioned. When we compute f using standard library functions we can expect to obtain relative errors in the computed exponential and logarithm bounded by u (de Dinechin *et al.*, 2007), (Muller, 2016), (Muller *et al.*, 2018, Chap. 10), that is,

$$\widehat{y} = \mathrm{fl}(f(x)) = \log(\mathrm{e}^x(1+\delta_1))(1+\delta_2), \quad |\delta_1|, |\delta_2| \leqslant u. \tag{2.3}$$

The term $1+\delta_2$ just causes a small relative perturbation of $\widehat{y}$, so we have

$$\widehat{y} \approx \log(\mathrm{e}^x(1+\delta_1)) = x + \log(1+\delta_1) = x + \delta_1 + O(\delta_1^2).$$

Hence, since $y = x$,

$$\frac{|y - \widehat{y}|}{|y|} \lesssim \frac{u}{|x|} + O(u^2). \tag{2.4}$$

This relative error bound is much larger than u for $|x| \ll 1$, even though the problem is perfectly conditioned. So it is not reasonable to expect an algorithm to be unconditionally forward stable in floating-point arithmetic. For this trivial computation, backward error and forward error are the same, so we also conclude that we cannot expect to obtain an algorithm that is unconditionally backward stable.

The softmax function has condition number

$$\operatorname{cond}(g,x) := \lim_{\varepsilon \to 0} \sup_{\|e\| \leqslant \varepsilon \|x\|} \frac{\|g(x+e)-g(x)\|}{\varepsilon \|g(x)\|},$$

which is given explicitly by

$$\operatorname{cond}(g,x) = \frac{\|G(x)\|\,\|x\|}{\|g(x)\|}.$$

Here, the $n \times n$ matrix $G(x) = (\partial g_i / \partial x_j)$ is the Jacobian of g and $\|\cdot\|$ denotes any vector norm and the corresponding subordinate matrix norm. We note in passing that G is the Hessian of f and can be shown to be symmetric positive semidefinite for all x (Boyd & Vandenberghe, 2004, p. 74). Now

$$\frac{\partial g_i}{\partial x_j} = \begin{cases} \dfrac{-\mathrm{e}^{x_i}\mathrm{e}^{x_j}}{\left(\sum\limits_{k=1}^{n} \mathrm{e}^{x_k}\right)^2}, & i \neq j, \\[2em] \dfrac{\mathrm{e}^{x_i}\sum\limits_{k=1}^{n} \mathrm{e}^{x_k} - \mathrm{e}^{2x_i}}{\left(\sum\limits_{k=1}^{n} \mathrm{e}^{x_k}\right)^2}, & i = j. \end{cases}$$

We have, for each i,

$$\sum_{j=1}^{n} \left|\frac{\partial g_i}{\partial x_j}\right| = \frac{2\mathrm{e}^{x_i} \sum\limits_{\substack{j=1 \\ j \neq i}}^{n} \mathrm{e}^{x_j}}{\left(\sum\limits_{k=1}^{n} \mathrm{e}^{x_k}\right)^2} \leqslant 1, \tag{2.5}$$

that is, $\|G(x)\|_\infty \leqslant 1$. Hence

$$\operatorname{cond}_\infty(g,x) \leqslant \frac{\|x\|_\infty}{\|g(x)\|_\infty} \leqslant n\|x\|_\infty,$$

because $\|g\|_\infty \geqslant n^{-1}\|g\|_1 = n^{-1}$.

We note that if $x_i \equiv \xi$ for all i then $\|g(x)\|_\infty = n^{-1}$ and $\|G\|_\infty = 2(n-1)/n^2$ by (2.5), so $\operatorname{cond}_\infty(g,x) = (2(n-1)/n)\|x\|_\infty$. Hence $\operatorname{cond}_\infty(g,x)$ can be arbitrarily large.

We also note that shifting, as in (1.3) and (1.4), does not change the functions so does not change their condition numbers; likewise for (1.5). These reformulations may, of course, affect the accuracy of the floating-point evaluation.

3. Basic algorithms and error analysis

Algorithm 3.1 gives a naive implementation of (1.1) and (1.2).

Algorithm 3.1 Given $x \in \mathbb{R}^n$, this algorithm computes $f(x) = \log \sum_{i=1}^{n} \mathrm{e}^{x_i}$ and the gradient $g(x) = \nabla f(x)$.

```
$s = 0$
for i = 1:n
    $w_i = \exp(x_i)$
    $s = s + w_i$
end
$f = \log(s)$
for i = 1:n
    $g_i = w_i / s$
end
```

What can be said about the accuracy of this algorithm when it is implemented in floating-point arithmetic? To answer this question we carry out a rounding error analysis. Throughout this section, we assume that there is no overflow or underflow.

First, we consider the error in evaluating the sum of positive terms

$$s = \sum_{i=1}^{n} \mathrm{e}^{x_i} \equiv \sum_{i=1}^{n} w_i.$$

Evaluating $w_i = \mathrm{e}^{x_i}$ yields a computed result satisfying

$$\widehat{w}_i = \mathrm{e}^{x_i}(1 + \delta_1), \tag{3.1}$$

where, as noted in Section 2, we can expect the relative error from the exponential evaluation to satisfy $|\delta_1| \leqslant u$. Therefore

$$|\widehat{w}_i - w_i| \leqslant w_i u.$$

Write the (exact) sum of computed quantities as

$$\widetilde{s} = \sum_{i=1}^{n} \widehat{w}_i.$$

The rounding error analysis in Higham (1993) and Higham (2002, sec. 4.2) shows that the computed sum $\widehat{s}$ satisfies[6]

$$|\widetilde{s} - \widehat{s}| \leqslant u \sum_{i=1}^{n-1} |t_i| + O(u^2),$$

where $t_i = \sum_{j=1}^{i+1} \widehat{w}_j$, so that, since $\widehat{w}_i \geqslant 0$,

$$|\widetilde{s} - \widehat{s}| \leqslant u(n-1)(\widehat{w}_1 + \widehat{w}_2) + u \sum_{i=3}^{n} (n+1-i)\widehat{w}_i + O(u^2).$$

[6] We note that more refined error bounds for summation are available under additional assumptions on the floating-point arithmetic (Lange & Rump, 2019), (Rump, 2012).

Here, and throughout this paper, the $O(u^2)$ term is innocuous in that it becomes significant only when the corresponding first order term is already so large that it provides no useful information; see Blanchard *et al.* (2020, sec. 2.1.2) for details of these terms for summation. Writing $s-\widehat{s}=s-\widetilde{s}+\widetilde{s}-\widehat{s}$, we obtain

$$\begin{aligned}|s-\widehat{s}| &\leqslant \sum_{i=1}^{n}|\widehat{w}_i-w_i|+|\widetilde{s}-\widehat{s}|\\ &\leqslant u\sum_{i=1}^{n}w_i+u\sum_{i=1}^{n}(n+1-i)\widehat{w}_i+O(u^2)\\ &=\sum_{i=1}^{n}(n+2-i)w_i+O(u^2), \qquad (3.2)\end{aligned}$$

since $\widehat{w}_i=w_i+O(u)$. Hence

$$\widehat{s}=s+\Delta s, \quad |\Delta s|\leqslant (n+1)us+O(u^2). \qquad (3.3)$$

Then the computed log-sum-exp is

$$\begin{aligned}\widehat{y} &= \mathrm{fl}(\log\widehat{s})=\log(\widehat{s})(1+\varepsilon), \quad |\varepsilon|\leqslant u,\\ &=\log(s+\Delta s)(1+\varepsilon)\\ &=\left(\log s+\frac{\Delta s}{s}+O(u^2)\right)(1+\varepsilon)\\ &=y(1+\varepsilon)+\frac{\Delta s}{s}+O(u^2). \qquad (3.4)\end{aligned}$$

Using (3.3) we obtain

$$|y-\widehat{y}|\leqslant u|y|+(n+1)u+O(u^2),$$

which gives the following result.

THEOREM 3.2 (Basic log-sum-exp algorithm) In the absence of overflow and underflow, the computed log-sum-exp $\widehat{y}$ from Algorithm 3.1 satisfies

$$\left|\frac{y-\widehat{y}}{y}\right|\leqslant\left(1+\frac{n+1}{|y|}\right)u+O(u^2). \qquad (3.5)$$

This bound is a factor $(|y|+n+1)/\|x\|_\infty$ larger than $\mathrm{cond}(f,x)u$ in (2.2). But $|y|\leqslant|x_{\max}|+\log n$ from (1.1) (or see (4.2) below), so this factor is bounded by $1+(n+1+\log n)/\|x\|_\infty$. Hence we have forward stability as long as $\|x\|_\infty\gtrsim 1$, but for $\|x\|_\infty\ll 1$ the bound does not guarantee forward stability. This is consistent with the bound (2.4) for the case $n=1$.

Turning to the evaluation of the softmax function g from its definition (1.2), by (3.1) we have

$$\widehat{g}_j=\frac{\mathrm{e}^{x_j}(1+\delta_1)}{\widehat{s}}(1+\delta_2), \quad |\delta_2|\leqslant u,$$

where δ_2 accounts for the division, and so by (3.3),

$$\widehat{g}_j=\frac{\mathrm{e}^{x_j}}{s(1+\eta)}(1+\delta_1)(1+\delta_2), \quad |\eta|\leqslant(n+1)u+O(u^2).$$

Therefore

$$\widehat{g}_j = g_j(1+\tau_j), \quad |\tau_j| \leqslant (n+3)u + O(u^2).$$

This bound guarantees a relative error of order at most nu in every component of g. We weaken the bound into a normwise bound for the next theorem.

THEOREM 3.3 (Basic softmax algorithm) In the absence of overflow and underflow, the computed softmax $\widehat{g}$ from Algorithm 3.1 satisfies

$$\frac{\|g-\widehat{g}\|_\infty}{\|g\|_\infty} \leqslant (n+3)u + O(u^2). \tag{3.6}$$

While the error bounds of Theorem 3.2 and 3.3 have a very satisfactory form, they provide no useful information when $n \gtrsim 1/u$, and for fp16 this happens for n as small as 2048. We note, however, that the n terms, which come from the summation, are pessimistic. It is shown by Higham & Mary (2019, Thm. 3.1) and Higham & Mary (2020, Thm. 2.5) that, under probabilistic models of rounding errors, n in the error bound for summation can be replaced by a small constant multiple of $\sqrt{n}$ with high probability, and the same holds for the bounds of Theorem 3.2 and 3.3.

Next, consider the alternative formula (1.5), which we rewrite here:

$$g_j = \exp\left(x_j - \log\sum_{i=1}^{n} \mathrm{e}^{x_i}\right) = \exp(x_j - y). \tag{3.7}$$

With $y = f(x)$ evaluated in floating-point arithmetic by Algorithm 3.1, we obtain

$$\begin{aligned}
\widehat{g}_j &= (1+\delta)\exp\big[(x_j-\widehat{y})(1+\varepsilon)\big], \quad |\delta|,|\varepsilon| \leqslant u, \\
&= (1+\delta)\exp\big[(x_j-y+(y-\widehat{y}))(1+\varepsilon)\big] && (3.8)\\
&= (1+\delta)g_j\exp\big[(x_j-y)\varepsilon+(y-\widehat{y})(1+\varepsilon)\big] && (3.9)\\
&= (1+\delta)g_j\big[(1+(x_j-y)\varepsilon+(y-\widehat{y})(1+\varepsilon)+O(u^2)\big] \\
&= (1+\tau_j)g_j, && (3.10)
\end{aligned}$$

where, using Theorem 3.2,

$$|\tau_j| \leqslant (|y|+|x_j-y|+n+2)u + O(u^2).$$

We summarize this result as follows.

THEOREM 3.4 (Alternative softmax algorithm) In the absence of overflow and underflow, the computed $\widehat{g}$ from (3.7) with the log-sum-exp computed by Algorithm 3.1 satisfies

$$\frac{\|g-\widehat{g}\|_\infty}{\|g\|_\infty} \leqslant \Big(|y| + \max_j |x_j-y| + n + 2\Big)u + O(u^2). \tag{3.11}$$

From (4.2) and (4.3) below, using the notation (1.6), we have

$$|y| + \max_j |x_j - y| \leqslant |x_{\max}| + |x_{\max} - x_{\min}| + 2\log n.$$

Hence (3.11) is less favorable than (3.6) when $x_{\max} - x_{\min} \gg n$ or $|x_{\max}| \gg n$. The analysis therefore suggests that (1.2) should be preferred to (1.5).

To give an intuitive explanation for the potential inaccuracy in (3.7), we refer to the steps leading to (3.10). A large absolute error in the argument of the exp in (3.9) may lead to a large relative error in the result. This effect can be traced back to the appearance of $x_j - y$ in (3.8).

4. Algorithms with shifting

Now we consider the use of shifts in the log-sum-exp and softmax evaluations in order to avoid overflow and reduce the chance of harmful underflow. We are particularly interested to see whether the potential cancellation caused by the shift can lead to numerical instability.

Recall the definition (1.6) of $x_{\max}$ and $x_{\min}$. Overflow in the exponential evaluations in (1.3) is certainly avoided if we take $a = x_{\max}$, as we then have $x_i - a \leqslant 0$ and hence $0 \leqslant \mathrm{e}^{x_i - a} \leqslant 1$ for all i. We can rewrite (1.3) as

$$y = x_{\max} + \log\left(1 + \sum_{\substack{i=1 \\ i \neq k}}^{n} \mathrm{e}^{x_i - x_{\max}}\right), \tag{4.1}$$

where $x_k = x_{\max}$ (if there is more than one such k, we can take any of them). From this expression we see that

$$x_{\max} \leqslant y \leqslant x_{\max} + \log n. \tag{4.2}$$

It follows that when $x_{\max} \geqslant 0$, the sum "$x_{\max} + \log(\cdot)$" that produces y in (4.1) cannot suffer cancellation.

Note that for $n = 1$, (4.1) trivially provides the exact result $y = x_{\max}$, in contrast to the basic formula (1.1).

For later use, we note that (4.2) implies that, for any j,

$$|y - x_j| \leqslant |x_{\max} - x_j| + \log n \leqslant |x_{\max} - x_{\min}| + \log n. \tag{4.3}$$

The log term in (4.1) has the form $\log(1+s)$, where $s \geqslant 0$. If s is very small then $1+s$ will round to 1 and the logarithm will evaluate as zero, even though $\log(1+s) \approx s \neq 0$. To avoid this loss of information we will use the function $\mathrm{log1p}(s) = \log(1+s)$ provided in, for example, C, MATLAB, and Numpy. These functions guarantee an accurate result for small s (which can be achieved with a simple formula based on log (Hewlett-Packard, 1982), (Higham, 2002, Prob. 1.5)). The improved accuracy brought by $\mathrm{log1p}(s)$ for small s is likely to benefit y when $x_{\max} \approx -s$.

These considerations lead to Algorithm 4.1.

Algorithm 4.1 (log-sum-exp and softmax with shift) This algorithm computes $f(x) = \log \sum_{i=1}^{n} \mathrm{e}^{x_i}$ and the gradient $g(x) = \nabla f(x)$ for $x \in \mathbb{R}^n$.

```
$[a,k] = \max_i x_i$ % $a = x_k = \max_i x_i$
$s = 0$
for $i = 1{:}n$
    $w_i = \exp(x_i - a)$
    if $i \neq k$, $s = s + w_i$, end
end
$f = a + \mathrm{log1p}(s)$
for i = 1:n
    $g_i = w_i/(1+s)$
end
```

Note that while it is important to avoid forming $1+s$ for the f-evaluation, for g we can safely form $1+s$ because if s is small it has little influence on g.

Algorithm 4.1 avoids overflow. If underflow occurs in the exponential then it is in a term in the sum added to 1 in (4.1), so that term is negligible and the underflow is harmless. Note, in particular, that if $x_i \approx x < \log r_{\min}^{(s)}$ for all i then whereas Algorithm 3.1 returns $f = -\infty$, Algorithm 4.1 suffers no underflow and returns $f \gtrsim x_{\max}$.

The main question is how shifting affects the accuracy of the evaluations. We give a rounding error analysis to assess this question. The analysis is a generalization of that in the previous section for the unshifted algorithm.

We first examine the error in evaluating the sum of nonnegative terms

$$s = \sum_{\substack{i=1 \\ i \neq k}}^{n} \mathrm{e}^{x_i - a} =: \sum_{\substack{i=1 \\ i \neq k}}^{n} w_i. \tag{4.4}$$

Evaluating $w_i = \mathrm{e}^{x_i - a}$ yields a computed result satisfying

$$\widehat{w}_i = \mathrm{e}^{(x_i - a)(1+\delta_1)}(1+\delta_2), \quad |\delta_1| \leqslant u,\ |\delta_2| \leqslant u.$$

Therefore

$$\widehat{w}_i = \mathrm{e}^{x_i - a}\mathrm{e}^{(x_i-a)\delta_1}(1+\delta_2) = \mathrm{e}^{x_i - a}\bigl(1 + (x_i - a)\delta_1 + O\bigl(\delta_1^2\bigr)\bigr)(1+\delta_2),$$

and hence

$$|\widehat{w}_i - w_i| \leqslant ((1 + a - x_i)u + O(u^2))w_i.$$

Assuming for notational simplicity that $k = n$, we can write the (exact) sum of computed quantities as

$$\widetilde{s} = \sum_{i=1}^{n-1} \widehat{w}_i.$$

The rounding error analysis in (Higham, 1993), (Higham, 2002, sec. 4.2) shows that the computed sum $\widehat{s}$ satisfies

$$|\widetilde{s} - \widehat{s}| \leqslant u \sum_{i=1}^{n-2} |t_i| + O(u^2),$$

where $t_i = \sum_{j=1}^{i+1} \widehat{w}_j$, so that, since $\widehat{w}_i \geqslant 0$,

$$|\widetilde{s} - \widehat{s}| \leqslant u \sum_{i=1}^{n-1} (n-i)\widehat{w}_i + O(u^2).$$

Hence

$$\begin{aligned} |s - \widehat{s}| &\leqslant \sum_{i=1}^{n-1} |\widehat{w}_i - w_i| + |\widehat{s} - \widetilde{s}| \\ &\leqslant u \sum_{i=1}^{n-1} (1 + a - x_i)w_i + u \sum_{i=1}^{n-1} (n-i)\widehat{w}_i + O(u^2) \\ &= u \sum_{i=1}^{n-1} \bigl(1 + a - x_i + n - i\bigr)w_i + O(u^2), \end{aligned} \tag{4.5}$$

since $\widehat{w}_i = w_i + O(u)$. Hence

$$\left| \frac{\widehat{s} - s}{s} \right| \leqslant (n + x_{\max} - x_{\min})u + O(u^2), \tag{4.6}$$

which guarantees an accurate computed sum as long as $n+x_{\max}-x_{\min}$ is not too large.

The final stage of the computation is to evaluate $y=x_{\max}+\log(1+s)$ using the computed $\widehat{s}$, for which we have

$$\widehat{y}=\big(x_{\max}+\log(1+\widehat{s})(1+\delta_3)\big)(1+\delta_4), \quad |\delta_3|,|\delta_4|\leqslant u.$$

Here, we are assuming that the log1p function has the property

$$\mathrm{fl}(\mathrm{log1p}(s))=\mathrm{log1p}(s)(1+\delta), \quad |\delta|\leqslant u.$$

Ignoring the innocuous δ_4 term and writing, by (4.6),

$$\widehat{s}=s(1+\eta), \quad |\eta|\leqslant(n+x_{\max}-x_{\min})u+O(u^2), \tag{4.7}$$

we have

$$\begin{aligned}\widehat{y}&=x_{\max}+\log(1+s(1+\eta))(1+\delta_3)\\&=x_{\max}+\log(1+s+s\eta))(1+\delta_3)\\&=x_{\max}+\left(\log(1+s)+\frac{s\eta}{1+s}+O(u^2)\right)(1+\delta_3),\end{aligned}$$

using a Taylor series expansion about $1+s$ of the logarithm. Hence

$$\widehat{y}-y=\log(1+s)\delta_3+\frac{s\eta}{1+s}(1+\delta_3)+O(u^2).$$

Bounding η using (4.7) gives

$$|y-\widehat{y}|\leqslant\log(1+s)u+\frac{s}{1+s}(n+x_{\max}-x_{\min})u+O(u^2) \tag{4.8}$$

or, as a relative error bound, since $s\geqslant 0$,

$$\left|\frac{y-\widehat{y}}{y}\right|\leqslant\left(\frac{\log(1+s)+n+x_{\max}-x_{\min}}{|y|}\right)u+O(u^2). \tag{4.9}$$

Simplifying the bound gives the next result.

THEOREM 4.2 (Shifted log-sum-exp algorithm) The computed log-sum-exp $\widehat{y}$ from Algorithm 4.1 satisfies

$$\left|\frac{y-\widehat{y}}{y}\right|=\left|\frac{y+n-x_{\min}}{y}\right|u+O(u^2). \tag{4.10}$$

The main question is how this result compares with Theorem 3.2 for the unshifted algorithm. The only difference in the bounds is that $|y|+n+1$ in (3.5) is replaced by $|y+n-x_{\min}|$ here. Now $|y+n-x_{\min}|\gg|y|+n$ is possible only if $x_{\min}\ll 0$ and $x_{\min}\ll x_{\max}$, so let us assume that these two inequalities hold. The term $|y+n-x_{\min}|$ comes from bounding the terms $(1+a-x_i+n-i)w_i$ in (4.5), where w_i is defined in (4.4) and $x_i=x_{\min}$, and if $x_{\min}\ll 0$ then $w_i=\mathrm{e}^{x_i-a}=\mathrm{e}^{x_{\min}-x_{\max}}\ll 1$. Hence the potentially large constant is mitigated by the w_i term that it multiplies—something that is lost in the manipulations to achieve a readable bound. We conclude that shifting should have little effect on the accuracy.

We note that (4.10) is weaker than necessary when $s\ll 1$ (recall that $s\geqslant 0$), which happens when $x_i\ll x_{\max}$ for all $i\neq k$, since we bounded $s/(1+s)$ by 1 in going from (4.8) to (4.9). If $s\ll 1$ then (4.8) becomes

$$|y-\widehat{y}|\lesssim s(1+n+x_{\max}-x_{\min})u+O(u^2).$$

Since $s \ll 1$ also implies $x_i \ll x_{\max}$ for $i \neq k$ and hence $y \approx x_{\max}$, we then have

$$\frac{|y-\widehat{y}|}{|y|} \lesssim s\frac{|1+n+y-x_{\min}|}{|y|}u+O(u^2),$$

which is a factor s smaller than (4.10).

Turning to the evaluation of the softmax function g from the shifted formula (1.4), we have, using (4.6),

$$\widehat{g}_j = \frac{\exp((x_j-a)(1+\delta_1))(1+\delta_2)(1+\delta_3)}{s(1+\eta)},$$

where δ_2 corresponds to the exponential evaluation and δ_3 to the division, and

$$|\delta_i| \leqslant u,\, i=1\colon 3, \qquad |\eta| \leqslant (n+x_{\max}-x_{\min})u+O(u^2).$$

Therefore

$$\begin{aligned}\widehat{g}_j &= g_j\frac{\exp((x_j-a)\delta_1)(1+\delta_2)(1+\delta_3)}{1+\eta}\\ &= g_j(1+\theta), \quad |\theta| \leqslant \big(n+2+2(x_{\max}-x_{\min})\big)u+O(u^2).\end{aligned}$$

Hence we have obtained the following result.

THEOREM 4.3 (Shifted softmax algorithm) The computed $\widehat{g}$ from Algorithm 4.1 satisfies

$$\frac{\|g-\widehat{g}\|_\infty}{\|g\|_\infty} \leqslant \big(n+2+2(x_{\max}-x_{\min})\big)u+O(u^2). \tag{4.11}$$

Again, this is broadly commensurate with Theorem 3.3 for the unshifted evaluation, bearing in mind the comments following Theorem 4.2.

Finally, we consider (1.5) with the log-sum-exp computed by Algorithm 4.1. In floating-point arithmetic we have the same equation (3.8) as for the unshifted algorithm, but now with θ bounded by, using (4.10),

$$|\theta| \leqslant (1+|x_j-y|+|y+n-x_{\min}|)u+O(u^2).$$

We have obtained the following result.

THEOREM 4.4 (Alternative shifted softmax algorithm) The computed $\widehat{g}$ from (1.5) with the log-sum-exp computed by Algorithm 4.1 satisfies

$$\frac{\|g-\widehat{g}\|_\infty}{\|g\|_\infty} \leqslant \Big(1+\max_j|x_j-y|+|y+n-x_{\min}|\Big)u+O(u^2). \tag{4.12}$$

This is broadly similar to Theorem 3.4 for the unshifted alternative softmax algorithm.

5. Computational experiments

We now perform some experiments in a realistic setting, using MATLAB R2020a. The codes and data used for the experiments are available online[7].

[7] https://github.com/higham/logsumexp-softmax-tests

Our aims are to examine the sharpness of the rounding error bounds and to give a pairwise comparison of the accuracy of the algorithms in floating-point arithmetic. Our data comes from a deep learning application. To generate the data, we first set up and trained an artificial neural network, using the MATLAB Deep Learning Toolbox (Deep Learning Toolbox, n.d.). More precisely, we trained a network to classify handwritten digit data from the widely used MNIST data set (LeCun *et al.*, n.d.). Here each data point is a grayscale 28×28 pixel image and there are ten categories: 0, 1, ..., 9. We used a network whose architecture has the following general form:

1. Image Input $28 \times 28 \times 1$ with normalization.
2. Convolution 8 $3 \times 3 \times 1$ stride [1 1] padding 'same'.
3. Batch Normalization 8 channels.
4. ReLU
5. Max Pool 2×2 stride [2 2] padding [0 0 0 0].
6. Convolution 16 $3 \times 3 \times 8$ stride [1 1] padding 'same' .
7. Batch Normalization 16 channels.
8. ReLU.
9. Max Pool 2×2 stride [2 2] padding [0 0 0 0].
10. Convolution 32 $3 \times 3 \times 16$ stride [1 1] padding 'same'.
11. Batch Normalization 32 channels.
12. ReLU.
13. Fully Connected 10 layer.
14. Softmax.
15. Classification Output crossentropy.

This is the default architecture from (Deep Learning Toolbox, n.d.), where further details may be found.

The network was trained on 7500 images (750 from each of the ten categories), with 2500 further images (250 from each of the ten categories) used for validation.

The network takes as input a 28×28 matrix corresponding to the pixels in the image and returns a nonnegative 10×1 vector whose ith component may be interpreted as the probability that the image came from category i. If we categorize according to the highest probability from the output, then the trained network misclassifed 27 of the 2500 validation images, corresponding to a 98.9% success rate.

The network uses single precision arithmetic, `fp32`. In our experiments, we are concerned only with floating-point arithmetic issues, and we treat the trained network as a means to produce a realistic data set. To do this, we extracted the 2500 single precision vectors from the validation set that were passed into the softmax layer and converted them to fp16 or bfloat16. We then used this data in our implementation of the softmax and log-sum-exp algorithms that we have studied in the previous sections.

To record errors in computed results we applied the basic algorithm, Algorithm 3.1, in single precision to provide a reference solution and used the `chop` function of Higham & Pranesh (2019) to simulate half precision arithmetic, in both the fp16 format and the bfloat16 format.

We first describe experiments in fp16. The components in the 2500 test vectors $x \in \mathbb{R}^{10}$ vary between about -19 and $+20$. As indicated in Table 1.2, e^x overflows in fp16 for $x \gtrsim 11$. Hence, in these tests, overflow is an issue for the basic log-sum-exp implementation in Algorithm 3.1: it generated an `Inf` for 475 of the 2500 test vectors. The shifted version of log-sum-exp in Algorithm 4.1 did not overflow. In the plots below, we do not include results for the cases where Algorithm 3.1 produced overflow.

First, we look at the log-sum-exp algorithms. In the upper left plot of Figure 5.1 we used the basic implementation of log-sum-exp, Algorithm 3.1. We scatter plot over the 2025 vectors where no overflow occurred. For each such vector, the horizontal coordinate is the leading term in the error bound

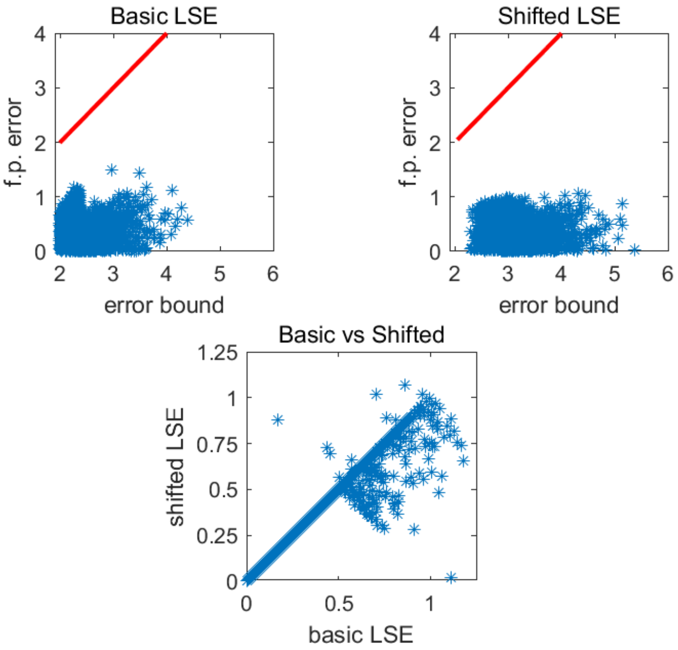

FIG. 5.1. Scatter plots of errors and error bounds, scaled by unit roundoff, over 2025 vectors in $\mathbb{R}^{10}$ for log-sum-exp algorithms in fp16. See the text for a description of the axes. Upper left: basic implementation of log-sum-exp from Algorithm 3.1. According to the error analysis, all points should lie below the reference line $y = x$ (shown in red). Upper right: corresponding results for the shifted implementation of log-sum-exp in Algorithm 4.1. Lower: scaled error from Algorithm 3.1 versus scaled error from Algorithm 4.1.

of Theorem 3.2, scaled by u, that is, $1+(n+1)/|y|$. Here, as shown in Table 1.1, $u = 4.88\times 10^{-4}$ for fp16. The vertical coordinate is the actual scaled relative error $|\widehat{y}-y|/(u|y|)$. The plot also gives a reference line of slope 1 from the origin. We see that the bound is always satisfied and is reasonably sharp in many cases.

In the upper right plot of Figure 5.1 we show corresponding results for the shifted log-sum-exp implementation in Algorithm 4.1, using the bound from Theorem 4.2.

In the lower part of Figure 5.1 we scatter plot the floating-point errors for the basic and shifted algorithms. Here, for 1863 out of the 2025 cases (92%) the two errors were identical to all digits in the half precision computation. In more detail, over all the data points the ratio of the error in the basic log-sum-exp (horizontal axis) divided by the error in the shifted version (vertical axis) varied between 0.19 and 59, with a mean of 1.07 and a standard error of 0.03. This indicates that the two versions perform similarly, with the shift producing slightly better results.

We now move on to the four softmax implementations. In Figure 5.2 we use the shifted softmax implementation from Algorithm 4.1, analysed in Theorem 4.3, as the basis for comparison. The upper left plot has the scaled error $\|\widehat{g}-g\|_\infty/(u\|g\|_\infty)$ from Algorithm 4.1 on the horizontal axis and the scaled error from the basic softmax in Algorithm 3.1 on the vertical axis. The upper right plot compares the

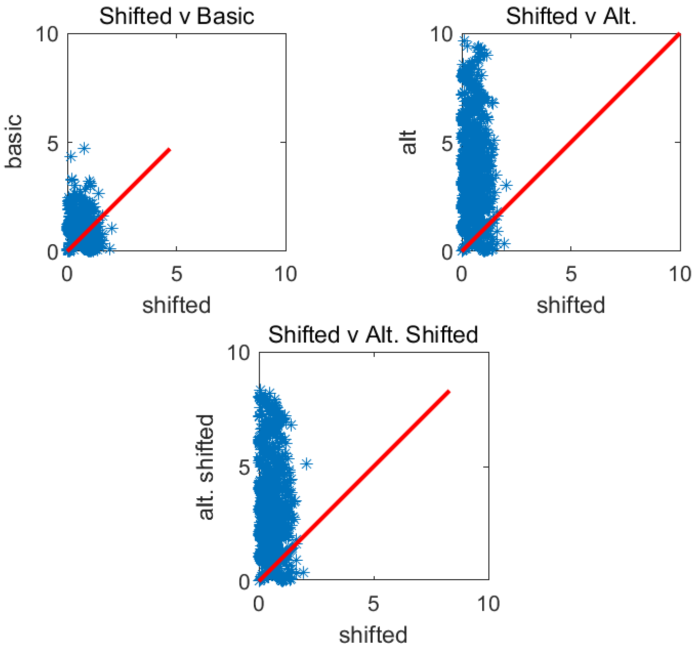

FIG. 5.2. Scatter plots of errors, scaled by unit roundoff, for softmax algorithms in fp16. See the text for a description of the axes. Reference line is $y = x$.

shifted softmax against the alternative algorithm using (3.7) and the unshifted log-sum-exp analyzed in Theorem 3.4. Similarly, the lower plot compares against the alternative shifted softmax algorithm that uses (3.9) with the shifted log-sum-exp, which is analyzed in Theorem 4.4. We see that the softmax values obtained from Algorithms 3.1 are generally slightly more accurate than those from Algorithm 3.1, whereas the alternative softmax versions based on the rewrite in (1.5) are mostly less accurate than those from Algorithm 4.1.

Overall, the results in Figures 5.1 and 5.2 are consistent with our floating-point error analysis.

A further test is to compute the sum of each softmax vector, which should equal 1. In Figure 5.3 we compare the softmax sums for the basic algorithm (Algorithm 3.1, red circles) analyzed in Theorem 3.3 and the alternative version (blue crosses) analyzed in Theorem 3.4. Similarly, Figure 5.4 compares the shifted softmax algorithm analyzed in Theorem 4.3 and its alternative analyzed in Theorem 4.4. The order along the x-axis is arbitrary; it corresponds to the order in which the data vectors were generated. These figures provide further evidence that the alternative softmax algorithms are less accurate than the basic or shifted algorithms.

We also conducted the corresponding experiments in simulated bfloat16 arithmetic. Here, as indicated in Tables 1.1 and 1.2, the number range is increased at the expense of reduced precision. In this case there was no overflow in any of the algorithms. The results were very similar to those for fp16, so

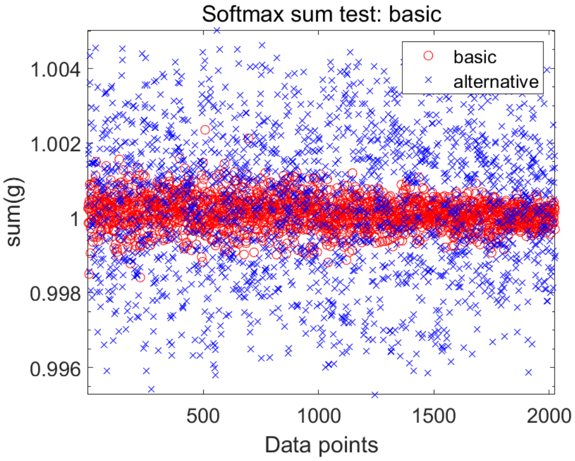

FIG. 5.3. Sum of entries of computed softmax vector for Algorithm 3.1 (red circles), analyzed in Theorem 3.3, and the alternative (blue crosses) analyzed in Theorem 3.4.

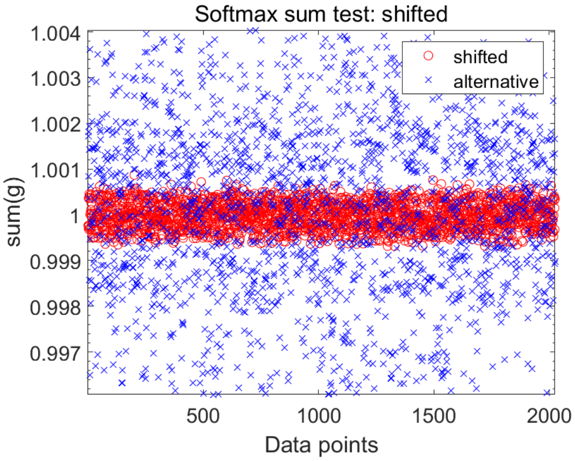

FIG. 5.4. Sum of entries of computed softmax vector for Algorithm 4.1 (red circles), analyzed in Theorem 4.3, and the alternative (blue crosses) analyzed in Theorem 4.4.

they are not shown here.

6. Conclusions

The log-sum-exp and softmax functions both feature in many computational pipelines, so it is important to compute them accurately and to avoid generating infs or NaNs because of overflow or underflow. To this end, a shift is usually incorporated into the defining formulas, yielding (1.3) and (1.4). It is important to understand the effect of the shift on the accuracy of the computed result, especially when computations are carried out in a low precision arithmetic such as bfloat16 or fp16, which have the equivalent of only 3 or 4 decimal digits of precision.

Our rounding error analysis shows that shifting by the largest element of the input vector does not lessen the accuracy of the computed log-sum-exp and softmax, so the shifted formulas can be safely used. Underlying this pleasing fact is the phenomenon that any large error constants caused by shifting are canceled by multiplication with small exponentials.

We obtained an explicit formula for the condition number of log-sum-exp and bounds for the condition number of softmax, and we were able to identify situations in which the log-sum-exp algorithms are guaranteed to be forward stable.

For the alternative and widely used softmax formula that avoids division, (1.5), we obtained larger error bounds than for the shifted formula (1.4). Since our numerical experiments confirm that larger errors are typically obtained in practice, we recommend using (1.4) instead of (1.5) to evaluate softmax.

In summary, Algorithm 4.1 is our recommendation for computing log-sum-exp and softmax. It avoids overflow, reduces the chance of harmful underflow, and generally produces results as accurate as those from the unshifted formulas.

Acknowledgements

We thank the referees for their helpful comments.

References

APRAHAMIAN, MARY, & HIGHAM, NICHOLAS J. 2014. The Matrix Unwinding Function, with an Application to Computing the Matrix Exponential. *SIAM J. Matrix Anal. Appl.*, **35**(1), 88–109.

BISHOP, CHRISTOPHER M. 2006. *Pattern Recognition and Machine Learning*. New York: Springer-Verlag.

BLANCHARD, PIERRE, HIGHAM, NICHOLAS J., & MARY, THEO. 2020. A Class of Fast and Accurate Summation Algorithms. *SIAM J. Sci. Comput.*, **42**(3), A1541–A1557.

BOYD, STEPHEN, & VANDENBERGHE, LIEVEN. 2004. *Convex Optimization*. Cambridge, UK: Cambridge University Press.

CALAFIORE, GIUSEPPE C., GAUBERT, STEPHANE, & POSSIERI, CORRADO. 2019. Log-Sum-Exp Neural Networks and Posynomial Models for Convex and Log-Log-Convex Data. *IEEE Trans. Neural Networks and Learning Systems*, 1–12.

CZAJA, JACEK, GALLUS, MICHAL, PATEJKO, TOMASZ, & TANG, JIAN. 2019 (May). *Softmax Optimizations for Intel Xeon Processor-Based Platforms*. ArXiv preprint 1904.12380.

DE DINECHIN, FLORENT, LAUTER, CHRISTOPH, & MULLER, JEAN-MICHEL. 2007. Fast and Correctly Rounded Logarithms in Double-Precision. *RAIRO-Inf. Theor. Appl.*, **41**(1), 85–102.

DEEP LEARNING TOOLBOX. The MathWorks, Inc., Natick, MA, USA. http://www.mathworks.co.uk/products/deep-learning/.

EFRON, BRADLEY, & HASTIE, TREVOR. 2016. *Computer Age Statistical Inference. Algorithms, Evidence, and Data Science*. Cambridge, UK: Cambridge University Press.

GOODFELLOW, IAN, BENGIO, YOSHUA, & COURVILLE, AAARON. 2016. *Deep Learning*. Cambridge, MA, USA: The MIT Press.

GUO, CHUAN, HANNUN, AWNI, KNOTT, BRIAN, VAN DER MAATEN, LAURENS, TYGERT, MARK, & ZHU, RUIYU. 2020 (Jan.). *Secure Multiparty Computations in Floating-Point Arithmetic*. ArXiv preprint 2001.03192.

HEWLETT-PACKARD. 1982. *HP-*15*C Advanced Functions Handbook*. Portable Computer Division, Corvallis, OR, USA: Hewlett-Packard. Part number 00015-90011 Rev. C.

HIGHAM, CATHERINE F., & HIGHAM, DESMOND J. 2019. Deep Learning: An Introduction for Applied Mathematicians. *SIAM Rev.*, **61**(4), 860–891.

HIGHAM, NICHOLAS J. 1993. The Accuracy of Floating Point Summation. *SIAM J. Sci. Comput.*, **14**(4), 783–799.

HIGHAM, NICHOLAS J. 2002. *Accuracy and Stability of Numerical Algorithms*. Second edn. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

HIGHAM, NICHOLAS J. 2008. *Functions of Matrices: Theory and Computation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

HIGHAM, NICHOLAS J., & MARY, THEO. 2019. A New Approach to Probabilistic Rounding Error Analysis. *SIAM J. Sci. Comput.*, **41**(5), A2815–A2835.

HIGHAM, NICHOLAS J., & MARY, THEO. 2020 (Jan.). *Sharper Probabilistic Backward Error Analysis for Basic Linear Algebra Kernels with Random Data*. MIMS EPrint 2020.4. Manchester Institute for Mathematical Sciences, The University of Manchester, UK.

HIGHAM, NICHOLAS J., & PRANESH, SRIKARA. 2019. Simulating Low Precision Floating-Point Arithmetic. *SIAM J. Sci. Comput.*, **41**(5), C585–C602.

IEEE. 2019. *IEEE Standard for Floating-Point Arithmetic, IEEE Std* 754-2019 (*Revision of IEEE* 754-2008). New York, USA: The Institute of Electrical and Electronics Engineers.

IEEE COMPUTER SOCIETY. 2008. *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008 (revision of IEEE Std 754-1985).* New York: IEEE Computer Society.

INTEL CORPORATION. 2018 (Nov.). *BFLOAT16—Hardware Numerics Definition*. White paper. Document number 338302-001US.

JONES, ERIC, OLIPHANT, TRAVIS, PETERSON, PEARU, *ET al.*. 2001–. *SciPy: Open Source Scientific Tools for Python*. http://www.scipy.org/.

LANGE, MARKO, & RUMP, SIEGFRIED M. 2019. Sharp Estimates for Perturbation Errors in Summations. *Math. Comp.*, **88**(315), 349–368.

LECUN, YANN, CORTES, CORINNA, & BURGES, CHRISTOPHER J. C. *The MNIST Database of Handwritten Digits*. Accessed June 17, 2019.

MATLAB CODE FOR MACHINE LEARNING ALGORITHMS IN BOOK PRML. `https://github.com/PRML/PRMLT`.

MULLER, JEAN-MICHEL. 2016. *Elementary Functions: Algorithms and Implementation*. Third edn. Boston, MA, USA: Birkhäuser.

MULLER, JEAN-MICHEL, BRUNIE, NICOLAS, DE DINECHIN, FLORENT, JEANNEROD, CLAUDE-PIERRE, JOLDES, MIOARA, LEFÈVRE, VINCENT, MELQUIOND, GUILLAUME, REVOL, NATHALIE, & TORRES, SERGE. 2018. *Handbook of Floating-Point Arithmetic*. Second edn. Boston, MA, USA: Birkhäuser.

MURPHY, KEVIN P. 2012. *Machine Learning: a Probabilistic Approach*. Cambridge, UK: Cambridge University Press.

RUMP, SIEGFRIED M. 2012. Error Estimation of Floating-Point Summation and Dot Product. *BIT*, **52**(1), 201–220.

STATISTICS AND MACHINE LEARNING TOOLBOX. *Statistics and Machine Learning Toolbox*. The MathWorks, Inc., Natick, MA, USA. `https://uk.mathworks.com/products/statistics.html`.

TEAM, R CORE. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

WANG, MEIQI, LU, SIYUAN, ZHU, DANYANG, LIN, JUN, & WANG, ZHONGFENG. 2018 (Oct.). A High-Speed and Low-Complexity Architecture for Softmax Function in Deep Learning. *Pages 223–226 of: 2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*.

WILLIAMS, CHRISTOPHER K. I., & BARBER, DAVID. 1998. Bayesian Classification with Gaussian Processes. *IEEE Trans. Pattern Analysis and Machine Intelligence*, **20**(12), 1342–1351.