IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 7 September 2023, accepted 21 October 2023, date of publication 30 November 2023, date of current version 8 December 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3338367

RESEARCH ARTICLE

Selective Path Automatic Differentiation: Beyond Uniform Distribution on Backpropagation Dropout

PAUL PESEUX[1,2], **MAXIME BERAR**[1], **THIERRY PAQUET**[1], **AND VICTOR NICOLLET**[2]
[1]Laboratoire d'Informatique, du Traitement de l'Information et des Systèmes (LITIS), 76800 Saint Étienne du Rouvray, France
[2]Lokad, 75013 Paris, France

Corresponding author: Paul Peseux (paul.peseux@gmail.com)

This work was supported in part by the University of Rouen; and in part by Lokad, a French company.

ABSTRACT This paper introduces Selective Path Automatic Differentiation (SPAD), a novel approach to reducing memory consumption and mitigating overfitting in gradient-based models for embedded artificial intelligence. SPAD extends the existing Randomized Automatic Differentiation, proposed by Oktay et al and which draws random paths through the backpropagation graph with matrix injection, by enabling alternative probability distributions on the backpropagation graph, thereby enhancing learning performance and memory management. In a specific iteration, SPAD evaluates and ranks multiple paths within the backpropagation graph. Over subsequent iterations, it preferentially follows these higher-ranked paths. This work also presents a compilation-based technique allowing model-agnostic access to random paths, ensuring generalizability across various model architectures, not restricted to deep models. Experimental evaluations conducted across various optimization functions demonstrate an enhanced minimization performance when employing SPAD. Additionally, deep learning experiments with SPAD notably mitigate overfitting, offering benefits akin to those of traditional dropout methods, but with a concomitant decrease in memory usage. We conclude by discussing the unique stochasticity implications of our work and the potential for it to augment other stochastic techniques in the field.

INDEX TERMS Gradient estimation, memory consumption, deep learning, dropout, embedded AI, automatic differentiation.

I. INTRODUCTION

Artificial intelligence (AI) based on deep learning architectures is being increasingly employed in various industries and everyday life [4], [18]. The next step in this direction is to embed AI on small devices, such as smartphones, with limited computing resources [2]. However, one of the major challenges in this context is the memory required to train deep architectures based on automatic differentiation [1]. This process is resource-intensive, particularly during the reverse mode of automatic differentiation [22], which is essential for training these types of deep architectures. This issue has been inherent to neural networks since the origins of the field. For example, Stochastic Gradient Descent (SGD) [16] is currently the only method suitable for working on huge datasets. Indeed it was a major breakthrough in addressing theoretically the splitting of large datasets into small batches while guaranteeing convergence. Another way to reduce memory usage is checkpointing [21] introduced as a trade-off between execution speed and memory consumption, without affecting gradient computations or updates. Beyond these seminal techniques, current works [15] on embedded AI systems still tries to limit the memory consumption with the additional objective to not compromise the accuracy of the model.

In addition to the resource consumption issue, overfitting of training data is a common unwanted behavior [17] that results in a reduced generalization power of the model. It occurs when the model memorizes the training set and fails to generalize to new data. To mitigate this issue, dropout was introduced in [7] on neural networks. Dropout involves temporarily turning off certain neurons of the model. By doing so, it prevents the network from relying too heavily on any group of neurons and improves the ability of the model to generalize well. Although

The associate editor coordinating the review of this manuscript and approving it for publication was Seifedine Kadry.

dropout reduces overfitting, current implementations do not reduce memory consumption. Dropout is applicable to deep learning models where parameters do not convey a specific meaning and can be randomly deactivated. Therefore it is not suitable for small white box models, where every parameter has a specific purpose and meaning. Thus, dropout cannot be applied to these models as turning off crucial parameters is unfeasible without blasting the model, even though the resulting regularization is a desirable feature.

An intermediate solution will be to apply dropout not during the forward pass, but during the backpropagation of the gradient. As an added benefit beyond generalization, dropping off the computation stage of some parameters gradient limits the resources consumption of the backpropagation. Recently, this was theoretically formulated in Randomized Automatic Differentiation (RAD) [12] that introduced a novel gradient estimator that constructs unbiased estimates by randomly drawing segments of the gradient code with uniform probability to apply dropped-out backpropagation. As a proper way to represent a model code, RAD uses the Linearized Computational Graph (LCG) of the model, in which the nodes represent intermediate variables and the edges represent the mathematical operations. In a LCG, each operation depends only on the output of the previous operation. This simplification allows for efficient calculations, as the operations can be computed one after the other, without the need to store all the intermediate values. An example is given on Figure 1.

The representation of the gradient code as a graph enables drawing paths along the edges thus granting a new form of gradient stochasticity. This stochasticity is based on the gradient decomposition into the contribution of each LCG path from the parameter θ to the output node z. This decomposition as a sum is formalized in Equation 1 with f_θ the function to minimize with respect to θ. RAD uses a uniform probability distribution to draw these paths. This is a valid starting point but is just one possibility. Other distributions may lead to better learning results. However, the concept of the optimal distribution over the backpropagation paths is dependent on the stage of the learning process. As the goal is to limit resource consumption, an optimization scheme for finding the temporal best would be counterproductive. This is why the focus will be on finding heuristics that emphasize the most important paths for gradient propagation.

$$\nabla_\theta f = \sum_{\theta \longrightarrow z} \prod_{z_k \longrightarrow z_l} \frac{\partial z_l}{\partial z_k}, \tag{1}$$

$z_k \longrightarrow z_l$ represents a directed edge connecting two nodes, and z is the output node that represents f_θ. The total gradient is the sum of all the paths contributions.

Selective Path Automatic Differentiation (SPAD) extends RAD by surpassing the limitations of the uniform distribution across various tasks without increasing memory usage. During each iteration of SGD, SPAD evaluates multiple paths within the backpropagation graph and assigns rankings based on their

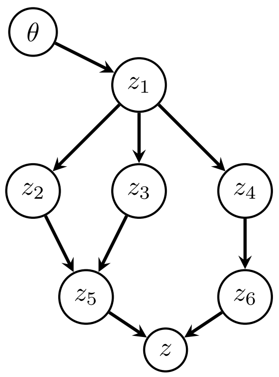


FIGURE 1. Example of the LCG of an objective function f_θ from the parameter θ to the output node z.

contribution to the overall gradient. Subsequently, it prioritizes following these higher-ranked paths in subsequent iterations. Moreover, while RAD proposes a technique for generating random paths within neural networks through random matrix injection, this work generalizes beyond deep neural models to enable the modification of the gradient estimator for any model architecture. The approach is based on compilation choices made during the automatic differentiation process, ensuring that each parameter is utilized only once, even if this necessitates duplicating variables. Automatic differentiation, the main computational technique for calculating function derivatives, has significantly facilitated the emergence of deep learning techniques by automatically computing gradients for custom-designed models, similar to how SGD enables gradient descent on large-scale datasets. Applying automatic differentiation to this model representation directly decomposes the gradient as a sum of path contributions.

In summary, this paper presents two contributions. Firstly, RAD is generalized into SPAD by allowing for alternative probability distributions on the LCG, which reduces overfitting and can be interpreted as a form of dropout during backpropagation. Secondly, a compilation-based technique is introduced that enables automatic access to the random paths without requiring any prior knowledge of the model architecture. The remainder of the paper is structured as follows. In the subsequent section, a novel gradient estimator, Selective Path Automatic Differentiation (SPAD), is introduced. This estimator extends RAD beyond the uniform distribution and a technique for applying it to general models without any prior knowledge of their structure is presented. The third section presents the results from experiments conducted on various objective functions. The paper concludes with a discussion on the novel direction of stochasticity prompted by this work and its similarities with dropout.

II. BEYOND UNIFORM DISTRIBUTION AND MATRIX INJECTION

A. BEYOND UNIFORM DISTRIBUTION: DISTRIBUTION PROBABILITY GENERALIZATION

Thanks to Equation 1, the gradient is decomposed as a sum of all the path contributions. This decomposition can be generalized as Equation 2, regardless of the provenance of each term of the sum.

$$\nabla_\theta f = \sum_{i=1..N} g_{\theta,i}. \tag{2}$$

The formulation will be adhered to, where each $g_{\theta,i}$ is related to a specific backpropagation path, even though all the following applies to optimization problems where the objective function is expressed as a sum, given the linearity of the derivative operator.

In addition to memory consumption reduction, it might help the optimization process by avoiding local minima. In gradient descent a local minimum gives a zero gradient that might slow or even stuck the minimization of the objective function. However the gradient being equal to zero does not mean that all the $g_{\theta,i}$ are zero. Using one of them might help to avoid this unwanted scenario. An example is given on a toy function:

Example 1: Let's consider the function $f_1 : \mathbb{R} \to \mathbb{R}$: $f_1(x) = x^2(2 + \cos(4x))$.

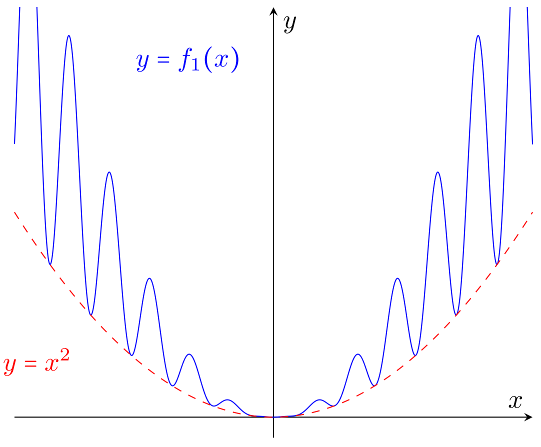


FIGURE 2. Representation of $f_1(x) = x^2(2 + \cos(4x))$. This function has an infinite number of local minima, but its general trend follows x^2.

This function is chosen because it presents multiple local minima. The decomposition of the derivative of f_1 following the backpropagation paths of its LCG is given in Equation 3.

$$\frac{\partial f_1}{\partial x} = 2\,x(2 + \cos(4x)) - 4x^2 \sin(4x) = g_{x,1} + g_{x,2} \tag{3}$$

If one employs the true gradient of f_1 and applies gradient descent with standard optimizers, it will undoubtedly become trapped in a local minimum. However, if one opts to utilize the first component of the gradient, it will reach the global minimum of f_1 at $x = 0$. This claim is supported by Figure 3.

This example underscores the usefulness of approaches based on code stochasticity. Although the impact is minimal on a function such as f_1, it is evident that computing only a fraction of the terms in the gradient sum reduces resource consumption, particularly memory.

The RAD approach utilizes a uniform distribution across all possible paths. However, it is suggested that not all gradient paths are equally important at any given time. Therefore,

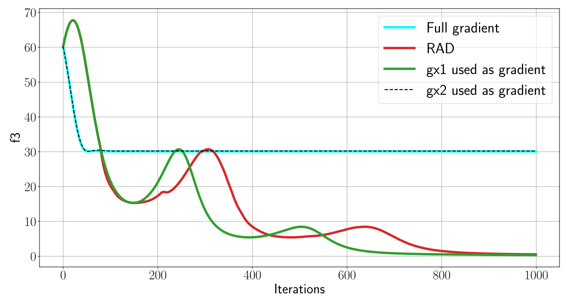


FIGURE 3. Minimization of f_1 through SGD with Adam [9] and its default values as optimizer. The starting point is $x = 5$. The blue curve represents the use of the full gradient from equation 3, which gets trapped in a local minimum. In contrast, the red curve represents the random selection of gradient terms during iterations, which allows for the avoidance of local minima and leads to a decrease in the target function. The green and black-dashed curves correspond to the static selection of one component of the gradient, $g_{x,1}$ and $g_{x,2}$ respectively.

a non-uniform distribution with varying probabilities to draw a $g_{\theta,i}$ is aimed to be utilized during gradient descent. Let $I_t \sim (p^t_{\theta,1}, \ldots, p^t_{\theta,N})$ be defined as the probability to draw $g_{\theta,i}$ to compute gradient descent, defined over the $T \in \mathbb{N}$ epochs. For notational simplicity, θ is omitted, which gives: $I_t \sim (p^t_1, \ldots, p^t_N)$. It follows that:

$$\forall t \leq T \quad \sum_{i=1}^{N} p^t_i = 1.$$

The intuition tells that locally, there is an optimal probability distribution that would decrease faster the objective function f_θ. There is no reason that this distribution is uniform. Supporting this intuition, it is argued that that certain $g_{\theta^t,i}$ may be negligible compared to others at a specific stage of the optimization process, i.e. at a specific iteration t. Drawing such $g_{\theta^t,i}$ would have an almost negligible impact on minimizing the objective function. As a result, resources would be better utilized by computing the $g_{\theta^t,i}$ that significantly reduces the target function. However, the magnitude of the $g_{\theta^t,i}$ depends on the position of the parameter $theta^t$ in the search space; therefore, the probability distribution should be updated alongside the iterations.

One of the consequences of such non-uniform distribution over the $g_{\theta,i}$ is the construction of a gradient estimator that may be biased. This is problematic as many convergence guarantees [5] rely on the unbiasedness of the gradient estimator. To address this issue, two key points are presented. First, the probability distribution I_t varies during the iterations of the learning process. The similarity between a $g_{\theta,i}$ and the exact gradient is not constant over the search space of θ. Therefore, the objective is to continuously update the probability associated with the terms of the gradient sum. Using the uniform distribution gives an unbiased estimator which gives convergence guarantees, so a proper update rule will smooth the probability associated to a backpropagation path $g_{\theta,i}$ over the iterations, i.e $\frac{1}{T}\sum_{t=1}^{T} p^t_i$ will tend toward $\frac{1}{N}$. In that case, the estimator becomes unbiased over the iterations.

Secondly, and of greater significance, a modification to the computed gradient is proposed to guarantee the unbiasedness of the new estimator, irrespective of the evolution of I_t.

Definition 1 (Normalization trick): Let's define $g_{\mathcal{I}}$ the stochastic gradient estimator relative to $I_t \sim (p_1^t, \dots, p_N^t)$:

$$g_{\mathcal{I}_t} = \begin{cases} \frac{1}{p_I^t} g_{\theta, I_t}, & \text{if } p_I^t > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

The corrective term $\frac{1}{p_i^t}$ is introduced in order to preserve the unbiasedness of the gradient estimator, which is necessary to rely on convergence guarantees [5]. $g_{\mathcal{I}_t}$ is unbiased as long as none of the p_i^t values are equal to zero:

$$\forall t \leq T \quad \mathbb{E}[g_{\mathcal{I}_t}] = \sum_{i=1..N} p_i^t \times \frac{1}{p_i^t} \mathbb{E}[g_{\theta,i}] = \mathbb{E}[\nabla_\theta f].$$

This normalization trick evacuates all the possible issues about unbiasedness of a non uniform distribution over the backpropagation paths. Let's remember that the objective is to constrain memory usage and prevent overfitting without excessively lengthen training time. Consequently, seeking the optimal term $g_{\theta,i}$ of the gradient at every iteration is infeasible. Inspired by multi-armed bandits [19], a heuristic is introduced that balances the exploration of the best probability distribution with the utilization of the one established during exploration, also known as exploitation.

B. SELECTIVE PATH AUTOMATIC DIFFERENTIATION

The search for a good path is computationally demanding, as finding the exact best path implies to evaluate all possible paths. An approximation is then to draw and evaluate a subset of path and choose the best path of these subset. But even in this case, repeating the procedure for every iteration will be costly. Notice that if a particular $g_{\theta,i}$ has the highest contribution to the gradient magnitude at a specific point θ^t, then it will also have the highest contribution in the surrounding parameter space as SGD is an iterative method. Hence, using this $g_{\theta,i}$ for a few iterations seems like a reasonable approximation. This approximation is even more reasonable when one assumes that the difference between the $g_{\theta,i}$ values is independent of the batch being used. In other words, the more the observation batch is representative of the dataset, the better the approximation.

SPAD is a new gradient estimator dealing with the trade-off of choosing the best component at a time and maintaining it for the following iterations. The set of the LCG paths is denoted as $\mathcal{P}$. m random paths in the backpropagation graph, denoted as $P_m \subset \mathcal{P}$, are sampled, and the induced gradients $g_{\theta,i}$ (with $i \in [1..m]$ without loss of generality) restricted to these paths are calculated. Among these m paths and for the next $k_{\max}$ iterations, the one yielding the largest gradient $i_{\max}$ is associated to an almost one probability with keeping an $\epsilon > 0$ fraction of exploration for all the other paths (not restricted to the m ones).

The Almost-Dirac notation $D_i^\epsilon(j)$ in 5 below is conveniently introduced to represent SPAD:

$$\forall \epsilon > 0; \forall i, j \leq N; \quad D_i^\epsilon(j) = (1-\epsilon)\delta_{j=i} + \frac{\epsilon}{N-1}\delta_{j\neq i}. \tag{5}$$

For a given $i \leq N$, D_i^ϵ can be used as probability distribution over $[1..N]$ as $\sum_j D_i^\epsilon(j) = 1$. The probability distribution of SPAD described above is formalized by I_t from Equation 6.

$$\forall t \leq T \quad I_t \sim D^\epsilon_{i^\tau_{\max}}, \tag{6}$$

with $i^\tau_{\max} = \underset{i \in P_m}{\arg\max} \|g_{\theta,i}\|$ and $t = \tau + r = qk_{\max} + r$ (euclidean division).

Algorithm 1 SPAD

```
Require: $\mathcal{Z}$ (data)
Require: $\theta \in$ model (model parameters)
Require: epochs, iterations, $k_{\max}$, $m$, $\epsilon$ (hyper parameters)
    $epoch \leftarrow 0$
    while $epoch < epochs$ do
        $k \leftarrow 0$
        for i $\in$ iterations do
5:          batch = $\mathcal{Z}$[i]
            do forward on batch
            for $\theta \in$ rev(model) do
                if $k \equiv 0 \pmod{k_{\max}}$ then
                    draw m random paths
10:                 $i_{\max} = \arg\max_{j \leq m} \|g_{\theta,j}(batch)\|$
                    for $j \leq m$ do
                        $p_{\theta,j} = (1-\epsilon)\delta_{j=i_{\max}} + \frac{\epsilon}{N-1}\delta_{j\neq i_{\max}}$
                    end for
                end if
15:             draw $I$ according to $(p_{\theta,1}, \dots p_{\theta,m})$
                update $\theta$ with $\frac{1}{p_{\theta,1}} g_{\theta,I}(batch)$
            end for
            $k \leftarrow k+1$
        end for
20:     $epoch \leftarrow epoch + 1$
    end while
    Return: $\theta$
```

SPAD pseudocode is presented in Algorithm 1 and is particularly appealing as it avoids the need for a complete evaluation of the gradient, which is a resource-intensive process. Additionally, it does not require additional memory beyond storing the m random paths and their associated gradients. Notice that, an implementation technique, checkpointing does not influence the gradient estimation itself, but rather the manner in which it is obtained. Consequently, all variations of checkpointing are compatible with SPAD. By choosing the largest gradient among the sampled paths, this approach has the potential to enhance the learning process, as the target loss is expected to decrease more rapidly compared to a uniform selection of the path. This heuristic introduces two new parameters, namely m and $k_{\max}$. However, there is a tradeoff to be made as increasing m may lead to a better gradient estimation but also slows down the learning process. With m and $k_{\max}$ both set to 1, SPAD reduces to RAD.

The parameter m represents the number of gradient paths to be selected to determine the one with the largest contribution. It is desirable to have a large value of m in order to ensure that the strongest contribution is well estimated. The estimation of a maximum is always underestimated but it will not have a strong impact on experiments thanks to quite large values of m. The parameter $k_{\max}$ determines the number of consecutive iterations that the chosen gradient path is used for. A large value of $k_{\max}$ can be used if the chosen path is a good one. During the $k_{\max}$ iterations, the parameters corresponding to the unchosen paths are frozen. If $k_{\max}$ is set to a large value, it makes the method similar to freezing layers presented in [6]. If $k_{\max}$ is large, a large value for m is preferable to ensure that the chosen random path is carefully selected for multiple iterations. However, if $k_{\max}$ is small, a smaller m can be tolerated since the path selection has an impact on a limited number of iterations.

The rationale behind using different probability distributions in SPAD is to find the optimal distribution that emphasizes the most important paths for gradient propagation at each iteration. Locally, certain paths may have a higher impact on minimizing the objective function, and prioritizing these paths can lead to faster convergence. By allowing alternative probability distributions on the backpropagation graph, SPAD adapts the distribution based on the current stage of the learning process, which improves learning performance. Additionally, by selectively computing the most significant paths, SPAD reduces memory consumption compared to evaluating all paths.

SPAD is an intermediate solution between RAD that does not try to determine the optimal choice of distribution and optimizations schemes that would need to duplicate the memory for the parameters, which would eliminate the benefits of SPAD. The implementation of SPAD through code stochasticity based on automatic differentiation, irrespective of the shape of the model to optimize, will be demonstrated in the following section. This contrasts with the RAD implementation, which, being based on matrix injections, was only compatible with neural networks.

C. FROM COMPILATION TO RANDOM PATHS: IMPLEMENTATION GENERALIZATION

The implementation of SPAD requires a computational method to obtain the terms of the gradient from Equation 1 written as a sum. This translates into identifying the backpropagation paths within the graph. Given the orientation of the LCG, identifying forward paths or backpropagation paths equates to the same problem. In a general context, without making any assumptions about the form of the LCG, an alternative method for executing this task is proposed on any language suited for automatic differentiation that satisfies the Static Single Assignment (SSA) and Single Access (SA) properties.

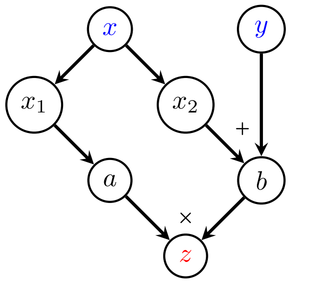


FIGURE 4. SA-LCG of $f_2(x, y) = e^x(x + y)$. The node x is a tupling node.

Definition 2 (SSA): Static Single Assignment (SSA) form is a property of a lower-level representation of a program that mandates each variable to be assigned precisely once, with every variable being defined before its use.

Definition 3 (SA): Single Access (SA) form is a property of a lower-level representation of a program which mandates that every variable is read at most once.

An example of such programming language crafted for automatic differentiation, satisfying both of these properties can be found in [14]. Moreover it is a simple operation to turn an SSA differentiable language like [11] and [20] into a SSA-SA one.

Due to the SA property, the LCG of a program will contain tupling nodes, as highlighted in Example 2. They make possible the construction of program using a variable multiple times by duplicating it. With the exception of these tupling nodes, there is only one edge that exits a node, which is a strict translation of the SA property on the LCG. Consequently, choosing a contribution to the gradient from Equation 1 involves following the path from a parameter node to the output node and selecting one of the edges emanating from the encountered tupling nodes.

Example 2: Let's consider the function $f_2(x, y) = e^x \times (x + y)$. To satisfy the Single Access (SA) property, since x is utilized twice in the program, its node is tupled, resulting in the LCG (and the corresponding program) as depicted in Figure 4 (5, respectively):

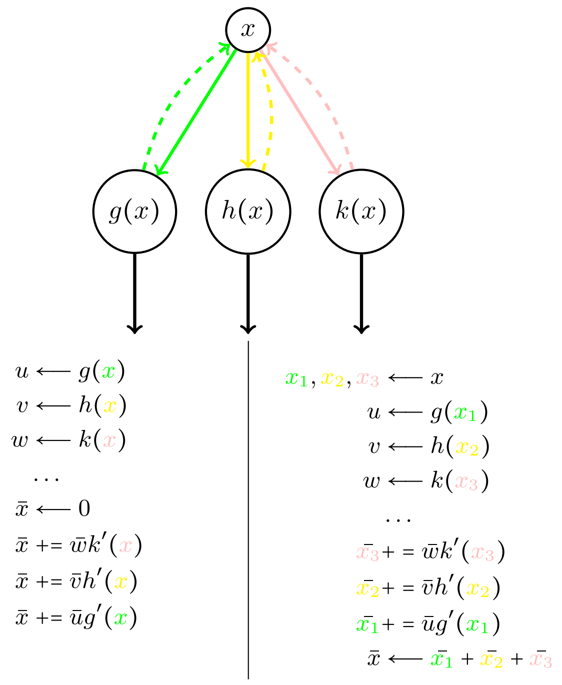
```
$x \leftarrow Param_0$
$y \leftarrow Param_1$
$x_1, x_2 \leftarrow x$
$a \leftarrow e^{x_1}$
$b \leftarrow x_2 + y$
$z \leftarrow a \times b$
$Return \quad z$
```

FIGURE 5. SSA-SA version of the program relative to f_2.

The tupling of variables in order to fulfill the SA property results in the gradient being expressed as a sum as proved in Equations 7. This is a key aspect of reverse mode automatic differentiation, also known as backpropagation. It is given by letting x be a parameter of f tupled in N variables $\{x_i\}_{i=1..N}$, Equation 1 turns into:

$$\begin{aligned}\frac{\partial f}{\partial x} &= \sum_{x\longrightarrow z} \prod_{z_k\longrightarrow z_l} \frac{\partial z_l}{\partial z_k} \\ &= \sum_{i=1}^{N} \frac{\partial x_i}{\partial x} \sum_{x_i\longrightarrow z} \prod_{z'_k\longrightarrow z'_l} \frac{\partial z_{l'}}{\partial z_{k'}} \\ &= \sum_{i=1}^{N} 1.\frac{\partial f}{\partial x_i} = \sum_{i=1}^{N} \frac{\partial f}{\partial x_i}. \end{aligned} \tag{7}$$

The chain rule of differentiation in reverse yields $\bar{x} = \frac{\partial f}{\partial x}$ called the adjoint of x, which is the desired output. Figure 6 highlights how the SSA-SA property directly gives the gradient as a sum.



```
$u \longleftarrow g(x)$
$v \longleftarrow h(x)$
$w \longleftarrow k(x)$
...
$\bar{x} \longleftarrow 0$
$\bar{x}$ += $\bar{w}k'(x)$
$\bar{x}$ += $\bar{v}h'(x)$
$\bar{x}$ += $\bar{u}g'(x)$
```

```
$x_1, x_2, x_3 \longleftarrow x$
    $u \longleftarrow g(x_1)$
    $v \longleftarrow h(x_2)$
    $w \longleftarrow k(x_3)$
    ...
    $\bar{x}_3$+ = $\bar{w}k'(x_3)$
    $\bar{x}_2$+ = $\bar{v}h'(x_2)$
    $\bar{x}_1$+ = $\bar{u}g'(x_1)$
    $\bar{x} \longleftarrow \bar{x}_1 + \bar{x}_2 + \bar{x}_3$
```

FIGURE 6. (Top) Generic non SA LCG, with *x* being used three different times. The dashed lines represent backpropagation. *g*, *h* and *k* are arbitrary differentiable functions. (Left) SSA version of the derivative program. (Right) SSA-SA version of the derivative program.

As previously framed, SPAD can be conceptualized as a form of dropout during backpropagation. By implementing SPAD independently of any specific model architecture, a generalized form of dropout can be introduced to a wider range of machine learning models. While dropout is a viable technique for deep learning models comprising numerous parameters without distinct significance for each individual one, it may not be suitable for smaller models.

The two approaches to obtain the gradient expressed as a sum, matrix injection or differentiation of SSA-SA languages, both rely on the multiple use of the parameters in the model implementation. If there is one and only one path from the parameter to the output node of the LCG, SPAD is pointless. Hopefully, this does not happen in many cases, as presented in the experiments Section III.

III. EXPERIMENTS

Experiments were conducted on two different types of tasks. Firstly, our novel gradient estimator was applied to a set of functions commonly used for evaluating optimization algorithms. These functions are not well-suited for gradient descent due to their numerous local minima. However, SPAD may overcome this issue by following gradient estimations rather than relying on the exact gradient. Evaluating SPAD on these functions provides further validation of its usefulness beyond the domain of neural networks. The emphasis in these functions lies on the terminal point of the optimization, whether it is in the proximity or not to the globally optimal solution that is known beforehand. Secondly, the estimator was tested[1] on the MNIST and CIFAR10 datasets using standard deep architectures to compare our method to existing ones. In order to assess the different approach, the considerations are placed on the accuracy achieved on the test data as well as the maximum memory utilization observed during the training process. These experiments vary significantly in several aspects. Firstly, the data varies greatly, as the first search space is 2-dimensional, while our dense architecture for MNIST, as presented in IV-D, has over 410k parameters. Additionally, the minimum of the optimization functions is known, which is not the case for neural networks. The diversity of these tasks provides a deeper understanding of the implications of the proposed method.

Remember that the theoretical probability distribution given by SPAD is an Almost-Dirac on the largest gradient contribution. In practice, the exact estimator $g_{\mathcal{I}}$ is not utilized. Instead, the largest gradient contribution is selected for $k_{\max}$ iterations, resulting in the simplified estimator $g_{\theta,I}$. It removes the necessity of a random draw at each iteration for choosing the backpropagation path. Consequently a proper implementation of this version requires the storage of only two random paths as our goal is not to sort the gradient norms, but rather to find the arg max. Therefore, the memory usage is independent of the value of m. This alternative version of SPAD is depicted in Algorithm 2.

A. OPTIMIZATION FUNCTIONS

The performance of the methods is evaluated on four optimization functions, employing the ϵ-success rate from Definition 4. This metric quantifies the ratio of optimizations with varying initializations that terminate at an ϵ value equal to or less than the known global minimum for the specified function.

Definition 4 (ϵ-success): $X_T \in \mathcal{Z}$ is an ϵ-success for the minimization of f if and only if $f(X_T) - \arg\min_{X\in\mathcal{Z}} f(X) < \epsilon$.

Experiments are conducted using three different setups. The baseline method, SGD with the classical full gradient

[1]https://github.com/ppmdatix/RandomizedAutomaticDifferentiation which is a fork from RAD.

estimator, is compared against RAD and SPAD. The functions utilized for evaluation, described in IV-B, have a confirmed minimum, thus enabling the definition of the ϵ-success. These functions, being incapable of representation as neural networks, are run on Envision, Lokad's domain specific language, where the random paths are extracted from the differentiation of the SSA-SA form of the language. Given that dropping out one of the few parameters of these functions is deemed insignificant, 1000 experiments are performed for each configuration using Adam [9] as the optimizer with its default values (learning rate of 0.01, $\beta_1 = 0.9$ $\beta_2 = 0.999$) over $T = 2000$ epochs. Tests were carried out for $k_{\max} = 5$ and $k_{\max} = 50$, and the ϵ-success rate is reported in Table 1. (and 2. respectively) for $\epsilon = 0.05$ ($\epsilon = 0.01$ respectively). Multiple values of m were not tested due to the parameter's dependency on the function in use. A single branch from each tupling node in the backpropagation graph execution was selected. For instance, in function f_1 from Example 2, when the input x is utilized twice in the function, m is set to 2. Furthermore, owing to the nature of the minimization objective, the construction of a validation set is infeasible.

TABLE 1. ϵ-success table with $\epsilon = 0.05$. In bold, the method with the higher ϵ-success rate for the corresponding function.

Function	baseline	RAD	$\text{SPAD}_{k_{\max}=5}$	$\text{SPAD}_{k_{\max}=50}$
Ackley	**12.2** %	0.1 %	2.1 %	1.6 %
Beale	70.8 %	65.2 %	67.0 %	**75.2** %
Levi	0.0 %	0.7 %	**2.2** %	1.4 %
Schaffer_2	**14.2** %	8.8 %	10.1 %	11.4 %

TABLE 2. ϵ-success table with $\epsilon = 0.01$. In bold, the method with the higher ϵ-success rate for the corresponding function. All the ϵ-success rate are lower than in Table 1 as ϵ is smaller.

Function	baseline	RAD	$\text{SPAD}_{k_{\max}=5}$	$\text{SPAD}_{k_{\max}=50}$
Ackley	**12.2** %	0.1 %	2.1 %	1.6 %
Beale	65.4 %	58.2 %	62.7 %	**70.5** %
Levi	0.0 %	0.2 %	**1.7** %	1.1 %
Schaffer_2	**13.9** %	8.8 %	10.0 %	11.3 %

Although gradient methods are not particularly suited for these functions, better results are observed with SPAD when the gradient expression is particularly suitable in the form of a sum, as seen in the Beale function. In more challenging cases, such as the Levi function, the baseline never manages to find a minimum, whereas using SPAD allows, albeit in a limited number of cases, to find the minimum.

The ϵ-success rate for varying values of $k_{\max}$ on the Beale and Levi functions is presented in Figure 7. On these examples the proposed method SPAD (with the appropriate $k_{\max}$) outperforms the baseline and RAD, which is very promising.

It also demonstrates that there is no universal optimal value of $k_{\max}$, as the performance seems increasing with $k_{\max}$ on the *Beale* function but decreases on the *Levi* function.

The choices made to conduct these experiments are motivated by two observations. Firstly, The choice of the functions in this section is motivated by the fact that they employ several times their input variables. As highlighted in Section II-C, it is necessary to use SPAD. Secondly, although SPAD is promoted as a way to reduce overfitting, this concept is not relevant in optimization problems where the goal is to find the optimal parameters that maximize the objective function, without considering factors such as the model's generalization capabilities.

B. DEEP LEARNING

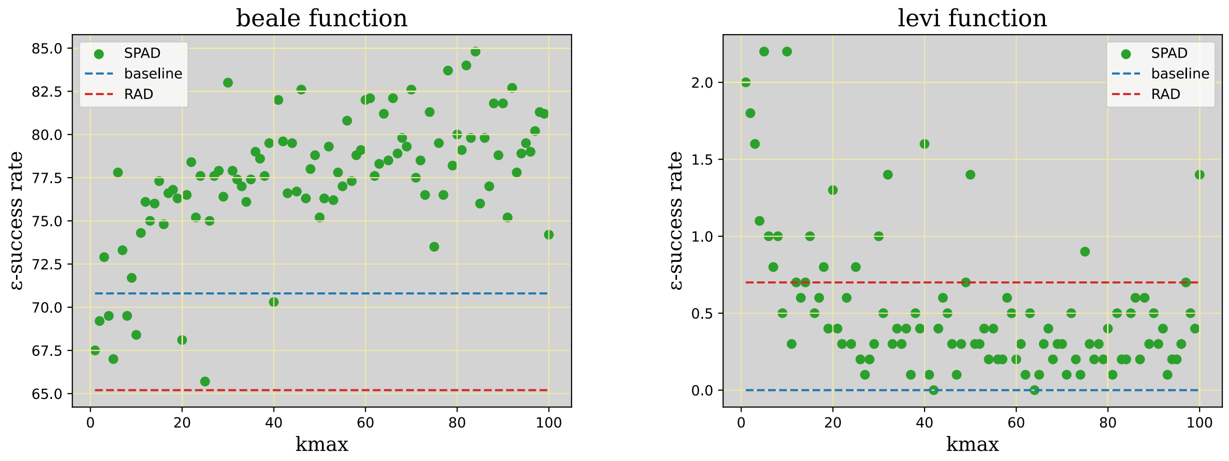
Experiments were conducted on the MNIST and CIFAR10 datasets, comparing SPAD with the standard stochastic gradient estimator (baseline), RAD and the dropout technique. The experimental framework described in [12] was employed, which does not involve early stopping to prevent increased memory requirements. However, this framework may result in overfitting, which is intended to be mitigated. This explains why dropout runs were made, to be able to compare with a method that is designed to mitigate overfitting.

The objective of our approach is to preserve learning quality while reducing the memory peak in comparison to traditional SGD.

Due to the substantial number of parameters in the utilized networks, drawing a single path in the backpropagation graph would lead to negligible updates. Instead, considering the fraction of the path to be drawn, experiments were conducted wherein 10% of the network was updated at each iteration. To elaborate, m sets of random paths were drawn, with each set covering 10% of the network. In contrast, the theoretical implementation of SPAD generates m random paths, with each path covering $\frac{1}{N}$% of the model. The same proportion (i.e., 10%) was applied during dropout runs. Our practical implementation of SPAD is described in Appendix IV-C.

The two metrics aimed to be optimized, namely the final accuracy on the testing dataset and the memory peak in percentage required during training, are depicted in Figure 8. The objective is to get the higher accuracy on testing with the lowest memory consumption, i.e. ending in the green zone. On these examples, the many variants of SPAD are competitive with the baseline and RAD, and it achieves strictly superior results on CIFAR10. In this context, seeking hyperparameters unrelated to SPAD is irrelevant. This explains why a validation set is dispensable, and assessing methods on the test dataset alone suffices. Note that all the runs share the same neural architectures, which is a fully connected network on MNIST and a convolutional one on CIFAR10. More details are given in IV-D.

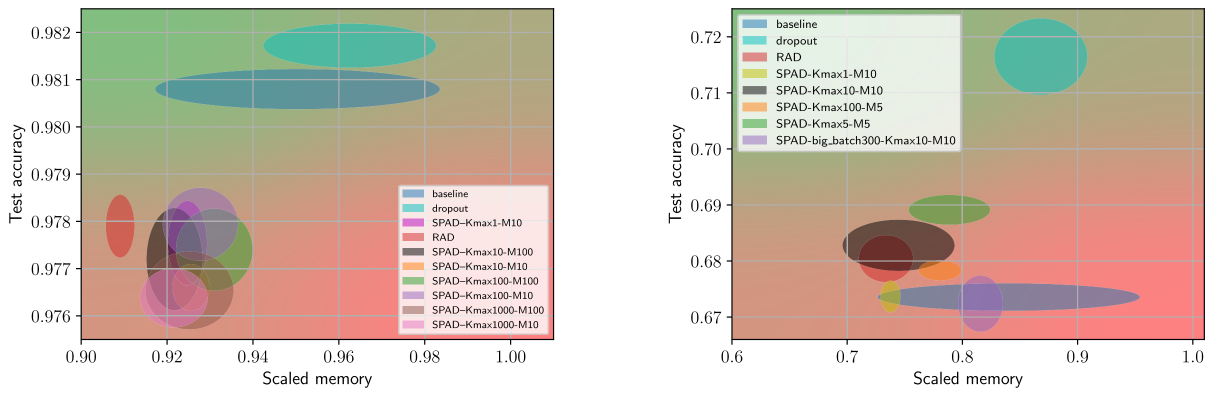
Concerning overfitting, detailed results on the CIFAR10 dataset are presented in Figure 9, while more details on the MNIST dataset are given in the appendices. They tend to confirm that our method effectively reduces it compared to the baseline. While the training loss of the baseline quickly decreases during the first iterations, its test loss quickly increases. On the contrary, SPAD slowly decreases its loss on the training dataset and its testing loss increases slowly compared to the baseline. This observation highlights the



(a) $k_{\max}$ impact on the ϵ-success of beale function minimization, with ϵ = 0.05.

(b) $k_{\max}$ impact on the ϵ-success of levi function minimization, with ϵ = 0.05.

FIGURE 7. ϵ-success as a function of $k_{\max}$. On these graphs, the higher the better. Both experiments show an impact of $k_{\max}$ on the ϵ-success of the gradient descent. On Figure 7a on the beale function, a bigger $k_{\max}$ upgrades the optimization while it is the opposite on Figure 7b and the levi function. In both cases, the better results are obtained with a version of SPAD that outperforms the baseline and RAD.



(a) SmallFCNet on MNIST. The network is made of 4 linear layers with Rectified Linear Unit activations.

(b) SmallConvNet on CIFAR10. The network is made of 4 convolutional layers with Rectified Linear Unit activations.

FIGURE 8. Accuracy on test versus memory peak tradeoff. The displayed memory is a fraction of the biggest memory peak of the baseline, the same is used for every run. The objective is to achieve the highest accuracy while minimizing memory usage. A run is considered as strictly better than another if it reaches higher accuracy with less memory. Otherwise one cannot rank two runs. The superior results are located in the upper left quadrant of the graph, indicated by the green color. With regards to the MNIST dataset, as shown in Figure 8a, none of the methods outperform the baseline, although the differences are minimal, as every model achieves over 97% accuracy. The least accurate results occur when $k_{\max} = 1000$. This outcome is reasonable since the selected paths may be utilized for an excessive number of iterations and might lose relevance at a specific stage. On 8b which concerns the CIFAR10 dataset, some versions of SPAD like ($k_{\max} = m = 10$) are strictly better than the baseline.

similarities between the process of randomly drawing paths during backpropagation and the dropout technique. Turning off a fraction of the network, on the forward pass for dropout and on the backpropagation for SPAD, tends to reduce overfitting.

The dropout runs attain the highest test accuracy with significant memory consumption. This approach effectively mitigates overfitting, as the testing loss increases at a much slower rate compared to other heuristics in Figures 11 and 12 from the appendices. Incorporating random matrix injections could prove highly beneficial for such learning techniques.

The primary objective of SPAD is to minimize memory consumption. From the perspective of a fixed memory budget, employing SPAD liberates resources that can be reallocated to increase the batch size, for instance. This hypothesis was evaluated by employing the SPAD method, utilizing a batch size twice as large as that in the other experiments, denoted as *big batch* in the legend of Figure 8b. While this approach led to increased memory consumption, it also resulted in heightened overfitting, exhibiting behavior akin to the baseline in both training and testing data sets. This observation is consistent with the findings of [8], which assert that large batch training methods are more prone to overfitting compared to the same network trained using smaller batch sizes.

IV. CONCLUSION AND PERSPECTIVES

From the perspective of deep learning, SPAD can be regarded as a combination of dropout and layer freezing within a neural network. By drawing backpropagation paths, SPAD proposes a similar technique to dropout for any gradient based model.

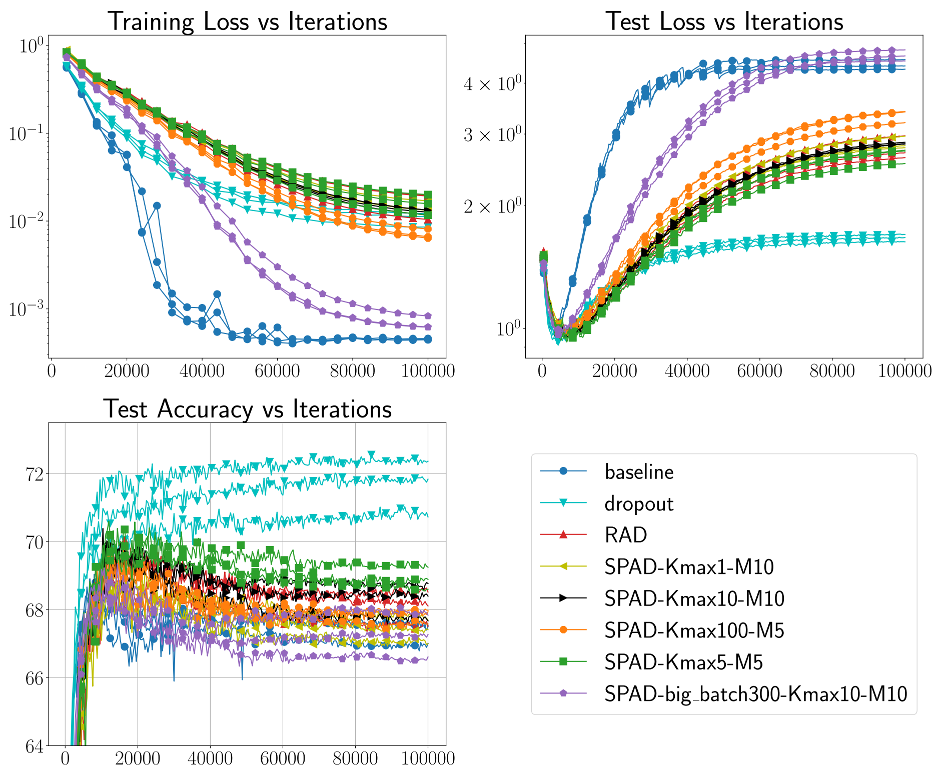


FIGURE 9. **Learning curves of the SmallConvNet on CIFAR10. The baseline model exhibits rapid performance improvement on the training dataset, while its performance on the testing dataset deteriorates just as quickly. This behavior is characteristic of overfitting, whereas the various versions of SPAD effectively mitigate this undesired decrease in generalization. The upper right plot suggests a continuum ranging from mild overfitting (dropout) to severe overfitting (baseline and SPAD with large batches).**

It is based on reverse mode automatic differentiation of SSA-SA languages.

Currently, the most significant limitation is the lack of an efficient implementation of the method to reduce memory consumption in deep learning applications. Further efforts are needed to properly code this method to enable an efficient implementation of the method in real-world environments, maximizing its benefits in terms of memory reduction in embedded artificial intelligence models. Another limitation is the lack of an appropriate heuristic or algorithm to determine the optimal parameter values for SPAD. Finding efficient parameter values that strike a balance between memory reduction and learning performance remains a challenge. Additionally, the effectiveness of SPAD may vary depending on the specific model architectures and datasets used, so generalizability across different scenarios would need to be investigated.

Overall the main idea is to draw more frequent examples that have a bigger impact on the loss minimization. Concerning this code's stochasticity, the result shows the advantages of a non uniform probability distribution. This is aligned with multiple works [3], [10] that use non-uniform distributions on the observations and outperforms the uniform one.

Table 3 summarizes the construction of gradient stochasticity based on the chosen stochasticity. The sampling process can be conducted at the observation or code level, with uniform or non-uniform distribution.

TABLE 3. **Small review of the stochasticity origin of gradient estimators.**

Granularity	GD	SGD	RAD	SPAD	[10]
Observation	**NO**	Uniform	Uniform	Uniform	Non-Uniform
Code	**NO**	**NO**	Uniform	Non-Uniform	**NO**

An interesting future work would be about non-uniform distributions on the observations and on the code, which could hopefully get better learning results without increasing the training memory needs. Such direction would help parameters updates on embedded artificial intelligence, which would open many industrial applications, like embedded machine learning on devices with constrained computational resources.

APPENDIX A
SPAD IMPLEMENTATION

A. SPAD IN PRACTICE

See Algorithm 2.

B. OTHER HEURISTICS

We have tested other probability distribution construction over the $g_{\theta,i}$ like

$$I_t \sim D^{\epsilon}_{s_{qk_{\max}}} \quad \text{with } s_t = \arg\min_{p \in P_m} \left\| \nabla f_\theta - g_{\theta,p} \right\| \tag{8}$$

Algorithm 2 SPAD in practice

```
Require: $\mathcal{Z}$ (data)
Require: $\theta \in$ model (model parameters)
Require: epochs, iterations, $k_{\max}$, $m$, $\epsilon$ (hyper parameters)
    $epoch \leftarrow 0$
    while $epoch < epochs$ do
        $k \leftarrow 0$
        for i $\in$ iterations do
5:          batch = $\mathcal{Z}$[i]
            do forward on batch
            for $\theta \in$ rev(model) do
                if $k \equiv 0 \pmod{k_{\max}}$ then
                    draw m random paths
10:                 $i_{\max} = \arg\max_{j \leq m} \left\| g_{\theta,j}(batch) \right\|$
                end if
                update $\theta$ with $\frac{1}{p_{\theta,1}} g_{\theta,i_{\max}}(batch)$
            end for
            $k \leftarrow k + 1$
15:     end for
        $epoch \leftarrow epoch + 1$
    end while
    Return: $\theta$
```

Nevertheless, none of the other tested methods yielded superior results compared to SPAD. Furthermore, SPAD exhibits the lowest memory consumption, as it eliminates the need to compute the full gradient even once, in contrast to the heuristic presented in Equation 8.

APPENDIX B OPTIMIZATION FUNCTIONS

Ackley

$$ackley(x, y) = -20 \exp\left[-0.2\sqrt{0.5\left(x^2 + y^2\right)}\right] \times \exp\left[0.5\left(\cos 2\pi x + \cos 2\pi y\right)\right] + e + 20$$

Beale

$$beale(x, y) = (1.5 - x + xy)^2 + \left(2.25 - x + xy^2\right)^2 + \left(2.625 - x + xy^3\right)^2$$

Levi

$$levi(x, y) = \sin^2(3\pi x) + (x-1)^2\left(1 + \sin^2 3\pi y\right) + (y-1)^2\left(1 + \sin^2 2\pi y\right)$$

Schaffer2

$$schaffer_2(x, y) = 0.5 + \frac{\sin^2\left(x^2 - y^2\right) - 0.5}{\left[1 + 0.001\left(x^2 + y^2\right)\right]^2}$$

APPENDIX C DEEP LEARNING

C. IMPLEMENTATION TRICK

The following paragraph is Pytorch-specific. The deep learning experiments were performed using Pytorch [13]. The main challenge was persisting tensors from the backward pass to the forward pass. The random paths implemented by the random matrix injection P needs to be persisted for multiple iterations. But they are chosen during the gradient calculation in the backward pass. Although intermediate tensors can be saved using the *save_for_backward*[2] function, there is no similar function for saving tensors from the forward pass to the backward pass. To address this issue, we pass P as a ghost input to the forward pass, manually updated its version in each of the k_{max} iterations into the corresponding gradient, and finally replaced P with the value artificially stored in its gradient. More detailed can be found in the released implementation.

Moreover, as presented by RAD, the practical implementation of the random matrix injections is not optimal. To get a fair comparison, the baseline is computed with identity matrix injection, which does not change anything to the final accuracy but makes the memory result comparable.

D. EXPERIMENTS

For consistence of comparison with RAD [12] results, the same methodology has been used

The fully-connected architecture on MNIST consists of:

1) Input: 784-dimensional flattened Image
2) Linear layer with 300 neurons (+ bias) (+ ReLU)
3) Linear layer with 300 neurons (+ bias) (+ ReLU)
4) Linear layer with 300 neurons (+ bias) (+ ReLU)
5) Linear layer with 10 neurons (+ bias) (+ softmax)

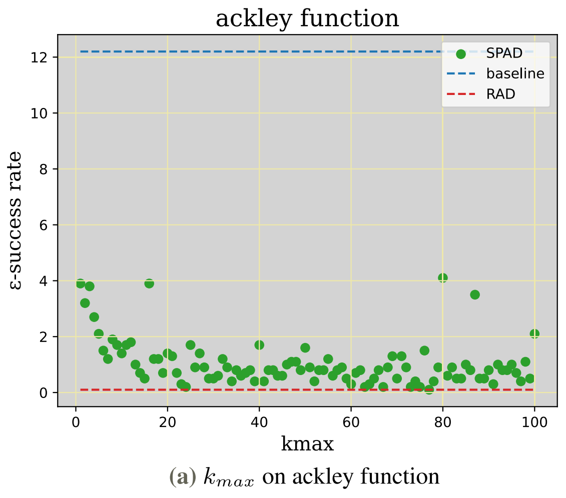
The convolutional architecture on CIFAR consists of:

1) Input: $3 \times 32 \times 32$-dimensional Image
2) 5×5 convolutional layer with 16 feature maps (+ 2 zero-padding) (+ bias) (+ ReLU)
3) 5×5 convolutional layer with 32 feature maps (+ 2 zero-padding) (+ bias) (+ ReLU)
4) 2×2 average pool 2-d
5) 5×5 convolutional layer with 32 feature maps (+ 2 zero-padding) (+ bias) (+ ReLU)
6) 5×5 convolutional layer with 32 feature maps (+ 2 zero-padding) (+ bias) (+ ReLU)
7) 2×2 average pool 2-d (+ flatten)
8) Linear layer with 10 neurons (+ bias) (+ softmax)

The MNIST models were trained using gradient descent with a fixed mini-batch size of 150 for 20,000 iterations with the learning rate $5.27.10^{-4}$ multiplied by 0.6 every 10% of the iterations. These hyperparameters were used throughout the training process and were not modified.

Similarly, the CIFAR-10 models were trained for 100,000 iterations using gradient descent with a fixed mini-batch size. The learning rate was reduced by 0.6 every 10,000 iterations, and we used the Adam optimizer.

[2] *torch.autograd.function.FunctionCtx.save_for_backward*



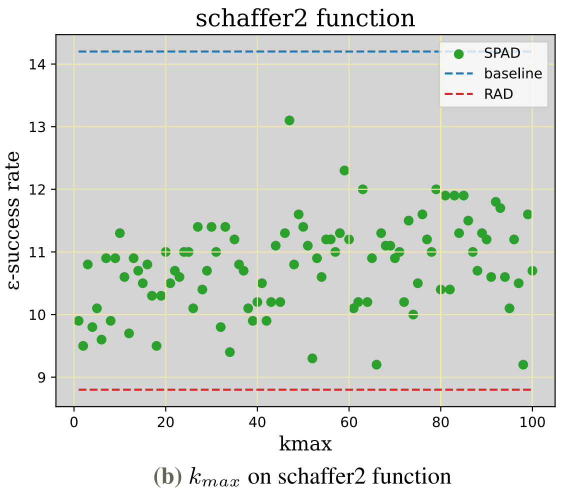


FIGURE 10. ϵ**-success in function of** k_{max}**. see Figure 8 for more context.**

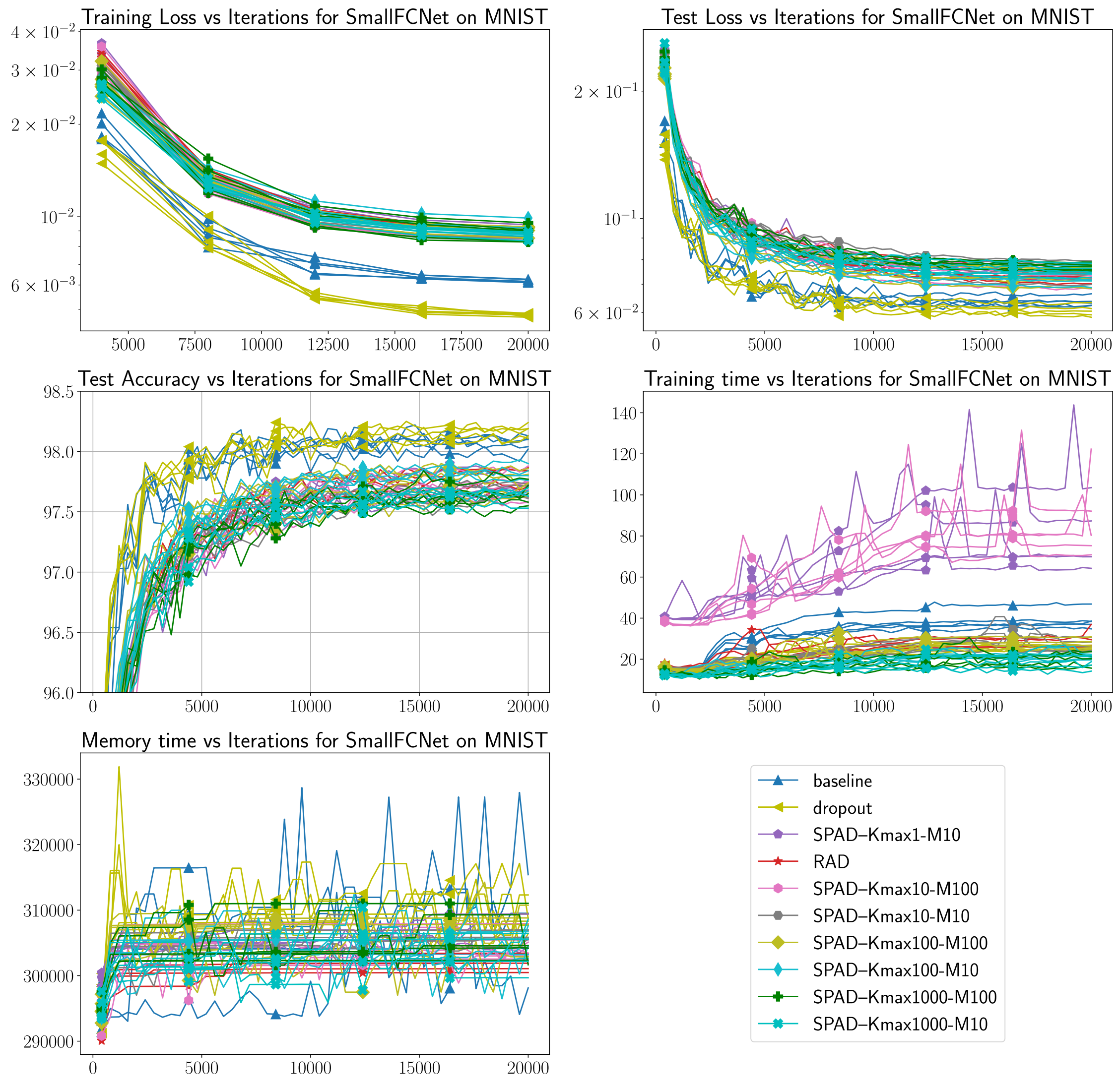


FIGURE 11. **Full train/test curves and memory consumption per iteration on MNIST.**

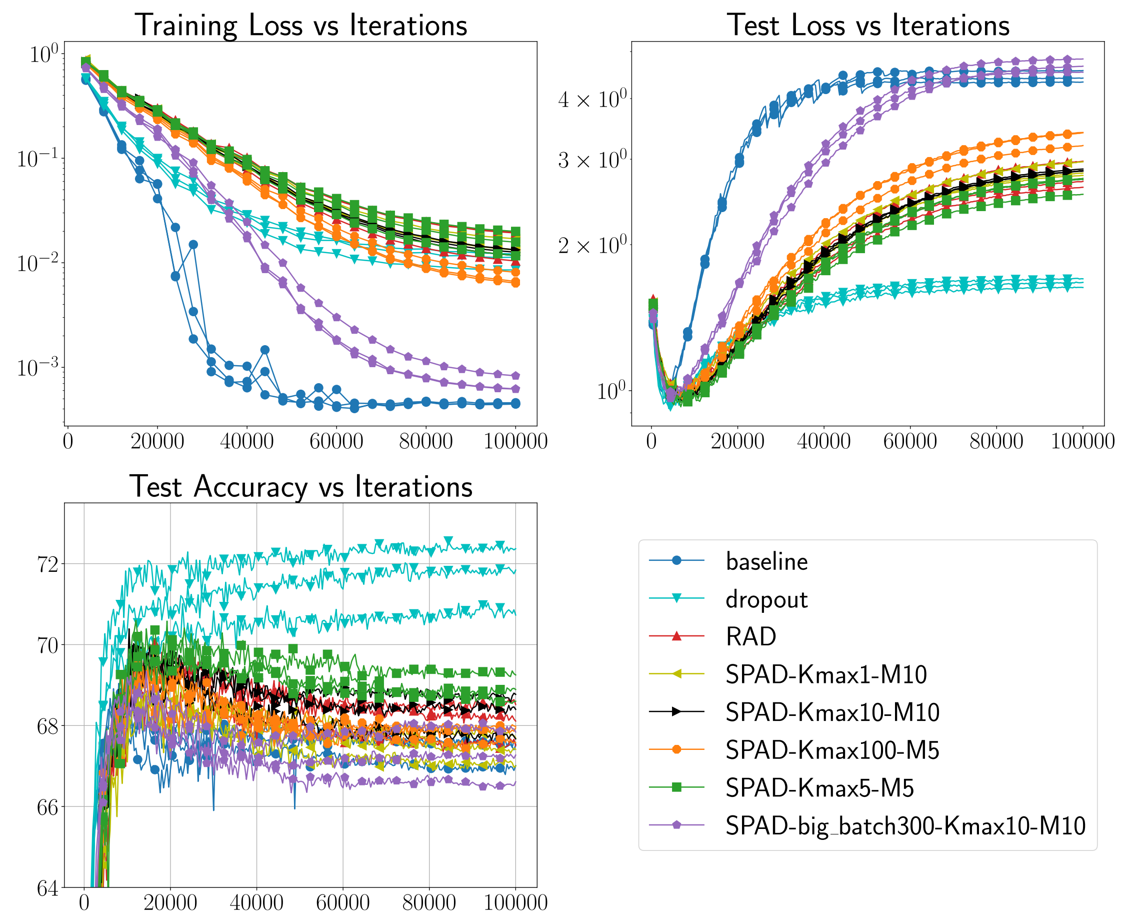


FIGURE 12. **Full train/test curves and memory consumption per iteration on CIFAR10.**

Data augmentation was not used, but the images were centered.

For each experiment reported in the main text, we tuned the initial learning rate and weight decay parameters for the feedforward networks. We generated 20 pairs of weight decay and learning rate values from specific distribution ranges.

To ensure consistent results, each experiment was trained five times using separate bootstrapped resamplings of the full training dataset (50,000 for CIFAR-10 and 60,000 for MNIST). The models were evaluated on the test dataset (10,000 for both). The repetition of these experiments were used to create the memory versus test accuracy plots.

On the MNIST dataset, the dropout backpropagation techniques perform slightly worse than the baseline and dropout. We do not observe any overfitting in this task, as shown in Figure 11, where the test accuracy does not decrease over the iterations even though this data is never used for updating the parameters. In contrast, on the CIFAR10 dataset (Figure 12), we observe that while training accuracies consistently increase, the test accuracies tend to decrease at some point for many techniques. This is especially true for the baseline, but not for the dropout technique. Dropout backpropagation techniques help mitigate overfitting, as notably highlighted by the evolution of the testing loss. Different versions of SPAD provide an interesting range between the baseline and dropout performance.

REFERENCES

[1] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," 2016, *arXiv:1604.06174*.

[2] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, "Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions," *ACM Comput. Surv.*, vol. 53, no. 4, pp. 1–37, Aug. 2020.

[3] D. Csiba and P. Richtárik, "Importance sampling for minibatches," *J. Mach. Learn. Res.*, vol. 19, no. 1, pp. 962–982, Feb. 2016.

[4] M. Cunneen, M. Mullins, and F. Murphy, "Autonomous vehicles and embedded artificial intelligence: The challenges of framing machine driving decisions," *Appl. Artif. Intell.*, vol. 33, no. 8, pp. 706–731, Jul. 2019.

[5] A. Defossez, L. Bottou, F. Bach, and N. Usunier, "On the convergence of adam and adagrad," Mar. 2020, *arXiv:2003.02395*.

[6] K. Goutam, S. Balasubramanian, D. Gera, and R. R. Sarma, "LayerOut: Freezing layers in deep neural networks," *Social Netw. Comput. Sci.*, vol. 1, no. 5, p. 295, Sep. 2020.

[7] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," Jul. 2012, *arXiv:1207.0580*.

[8] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," in *Proc. 5th Int. Conf. Learn. Represent. (ICLR)*, Toulon, France, Apr. 2017.

[9] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Represent.*, Dec. 2014.

[10] R. Liu, T. Wu, and B. Mozafari, "Adam with bandit sampling for deep learning," in *Proc. Annu. Conf. Neural Inf. Process. Syst. (NIPS)*, Dec. 2020.

[11] W. Moses and V. Churavy, "Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients," in *Advances in Neural Information Processing Systems*, vol. 33, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020, pp. 12472–12485.

[12] D. Oktay, N. McGreivy, J. Aduol, A. Beatson, and R. P. Adams, "Randomized automatic differentiation," 2020, *arXiv:2007.10412*.

[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, 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 *Proc. NIPS*, 2019, pp. 8024–8035.

[14] P. Peseux, "Differentiating relational queries," in *Proc. PhD@VLDB*, 2021.

[15] L. Ravaglia, M. Rusci, A. Capotondi, F. Conti, L. Pellegrini, V. Lomonaco, D. Maltoni, and L. Benini, "Memory-latency-accuracy trade-offs for continual learning on a RISC-V extreme-edge node," in *Proc. IEEE Workshop Signal Process. Syst. (SiPS)*, Oct. 2020, pp. 1–6.

[16] H. Robbins and S. Monro, "A stochastic approximation method," *Ann. Math. Statist.*, vol. 22, no. 3, pp. 400–407, 1951.

[17] S. Salman and X. Liu, "Overfitting mechanism and avoidance in deep neural networks," 2019, *arXiv:1901.06566*.

[18] P. Teikari, R. P. Najjar, L. Schmetterer, and D. Milea, "Embedded deep learning in ophthalmology: Making ophthalmic imaging smarter," *Therapeutic Adv. Ophthalmol.*, vol. 11, Jan. 2019.

[19] W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, vol. 25, nos. 3–4, p. 285, Dec. 1933.

[20] B. van Merrienboer, O. Breuleux, A. Bergeron, and P. Lamblin, "Automatic differentiation in ML: Where we are and where we should be going," in *Advances in Neural Information Processing Systems*, vol. 31, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2018.

[21] Y. M. Volin and G. M. Ostrovskii, "Automatic computation of derivatives with the use of the multilevel differentiating technique—1. Algorithmic basis," *Comput. Math. With Appl.*, vol. 11, no. 11, pp. 1099–1114, Nov. 1985.

[22] R. E. Wengert, "A simple automatic derivative evaluation program," *Commun. ACM*, vol. 7, no. 8, pp. 463–464, Aug. 1964.

PAUL PESEUX received the master's degree in mathematics and computer science from ENS Lyon and the degree from the Engineering School Centrale Lyon. He is currently pursuing the Ph.D. degree in collaboration with the Laboratoire d'Informatique, du Traitement de l'Information et des Systèmes (LITIS) EA4108 and Lokad, focusing on large-scale differentiable programming on relational data. He has experience in anomaly detection and game theory research. His research interests include machine learning, big data analytics, and applied mathematics.

MAXIME BERAR received the Ph.D. degree in signal processing from the Polytechnic Institute of Grenoble, in 2007. Since 2009, he has been an Assistant Professor with the Department of Physics, University of Rouen Normandie. His research interests include machine learning and signal processing with applications to brain–computer interfaces and audio applications.

THIERRY PAQUET received the master's degree in multimedia information processing with the University of Rouen Normandie, in 2012. In 2002, he was appointed as a Professor with the University of Rouen Normandie. He was also the Head of the Laboratoire d'Informatique, du Traitement de l'Information et des Systèmes (LITIS), the research laboratory in computer science, associated with the University of Rouen Normandie, Le Havre Normandy University, and the Rouen INSA School of Engineering. He has published more than 100 papers in international conferences and scientific journals. He contributed to many collaborative projects with academic or industrial partners. His research interests include machine learning, statistical pattern recognition, and deep learning, for sequence modeling, with application to document image analysis and handwriting recognition. He has supervised 20 Ph.D. students on these topics. From 2008 to 2016, he was a member of the governing board of the French Association for Pattern Recognition AFRIF. He was the President of the French Association Research Group on Document Analysis and Written Communication (GRCE), from 2002 to 2010. He is regularly invited as a reviewer in main international conferences and scientific journals.

VICTOR NICOLLET received the degree from École Normale Supérieure, Paris, in 2009, with a specialty in programming language semantics and static analysis. Since 2014, he has been the Chief Technical Officer of Lokad, where he leads all research initiatives related to programming languages and compiler design.

• • •