# EFFICIENT GRADIENT-BASED ALGORITHM FOR TRAINING DEEP LEARNING MODELS WITH MANY NONLINEAR ACTIVATIONS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

This research paper presents a novel algorithm for training deep neural networks with many nonlinear layers (e.g., 30). The method is based on backpropagation of an approximated gradient, averaged over the range of a weight update. Unlike the gradient, the average gradient of a loss function is proven within this research to provide more accurate information on the change in loss caused by the associated parameter update of a model. Therefore, it may be utilized to improve learning. In our implementation, the efficiently approximated average gradient is paired with RMSProp and compared to the typical gradient-based approach. For the tested deep model with many nonlinear layers on MNIST and Fashion MNIST, the presented algorithm: (a) generalizes better, at least in a reasonable epoch count, (b) in the case of optimal implementation, learning would require less computation time than the gradient-based RMSProp, with the memory requirement of the Adam optimizer, (c) performs well on a broader range of learning rates, therefore it may bring time and energy savings from reduced hyperparameter searches, (d) improves sample efficiency about three times according to median training losses. However, in the case of the tested shallow model, the method performs approximately the same as the gradient-based RMSProp in terms of both training and test loss. The source code is provided at [...].

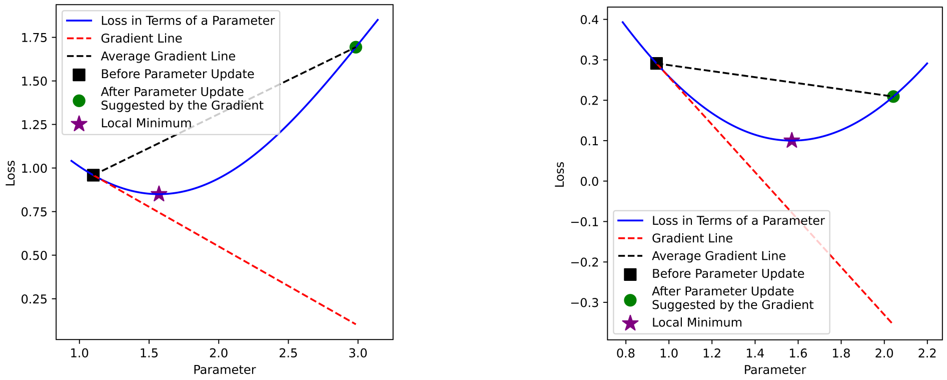
## 1 INTRODUCTION

### 1.1 BACKPROPAGATION OF ESTIMATED AVERAGE GRADIENT

This research paper presents an approach based on estimating the average gradient on the range of a potential parameter update. The average gradient, unlike the gradient, stores information about an accurate value of the loss delta (Figure 1). Therefore, the approach can be utilized to increase the precision of the computation of how loss is related to the parameters of a model. Consequently, it may contribute to more efficient loss minimization.

Our experiments show that deep learning with many nonlinear layers may be vastly enhanced by combining our method with RMSProp optimizer (Tieleman et al., 2012). In this research, RMSProp is paired with our method because the former does not incorporate momentum by default (i.e., in the most popular libraries like PyTorch or Keras). Therefore, comparing methods based on decreases in batch losses after each update is valid, assuming that the weight updates have similar magnitudes across compared methods. Nevertheless, the method can be paired with all other first-order optimizers, because it alters only the gradient. Importantly, our algorithm improves sample efficiency more than threefold (according to median training loss) for the tested deep model. Moreover, it may reduce energy consumption along with training time, performing better on unseen data. Furthermore, in the case of our method, additional electricity and time savings may be obtained due to faster hyperparameter searches. This is because it exhibits good performance over a wider range of learning rates compared to the gradient-based RMSProp for the deep model. However, in the case of the tested shallow model, the method performs approximately the same as the gradient-based approach in terms of both training and test loss.

In recent years, the sizes of deep learning models have significantly increased, vastly scaling up capabilities in different types of tasks. Moreover, due to the popularity of some nonlinear activation



(a) *Example 1.* The average gradient suggests a different direction for updating a particular parameter.

(b) *Example 2.* If the average gradient decreases in the same direction as the gradient, it additionally provides more information about the loss landscape.

Figure 1: *Comparison of Gradient and Average Gradient.* The latter accurately reflects the influence of a parameter update on loss (as described by Equation 3, under the assumptions that $f$ represents the visualized loss function, with $x$ and $x'$ denoting the parameter values before and after an update, respectively). The plots refer to a simple case with only one parameter of the model. Appendix F presents visualizations involving two parameters.

functions, also in models with huge numbers of layers, our approach may provide a perspective on improvements of practical deep learning. This especially regards areas, where sample efficiency is crucial, like reinforcement learning from human feedback (Kirk et al., 2023), which was used in popular chatbots, such as OpenAI's ChatGPT (OpenAI, 2023) or Anthropic's Claude (Kirk et al., 2023). Our method represents a step toward enabling very deep models to learn efficiently on the fly, akin to how people or some animals do.

### 1.2 GRADIENT OPTIMIZATION AND AVERAGING

Gradient optimization dominates deep learning with optimizers like Stochastic Gradient Descent (Liu et al., 2020), RMSProp (Tieleman et al., 2012), Adam (Kingma & Ba, 2014) or Nadam (Dozat, 2016). The leading algorithms for training do not change frequently over the years. However, our algorithm or its variants may be used along with first-order optimizers, as mentioned in Section 1.1.

Gradient averaging is commonly used in machine learning, but in a distinct scenario than in our approach. Momentum is the running average of gradients over subsequent batches (Liu et al., 2020; Kingma & Ba, 2014; Dozat, 2016). It prevents falling into local minimums and may accelerate learning. Similarly, averaging model parameters may improve convergence and learning speed (Ruppert, 1988; Polyak & Juditsky, 1992; Merity et al., 2017; Wei et al., 2023; Sun et al., 2010), though it requires a significant amount of memory. The technique can be described as averaging a function of gradients, as the averaged model parameters over subsequent updates depend on the gradient values.

Accumulating gradients over a batch is inherent in machine learning. In practice, it is equivalent to averaging gradients computed for multiple inputs. However, this approach alone does not take into account the information about a parameter update (Fig. 1), which remains unknown during its computation. Consequently, it does not guarantee the accuracy of computing the influence of the unknown parameter update on the loss. Nevertheless, batching remains fully compatible with our method and is employed in our implementation.

Our approach is more closely related to some second-order optimization methods (Tan & Lim, 2019) rather than momentum-based or parameter-averaging techniques. This is due to the utilization of information about the curvature of a loss function during each parameter update (Fig. 1). Recently, one of the most popular algorithms for second-order optimization of neural networks is L-BFGS (Berahas et al., 2016). However, the current methods in this field are impractical for training large models due to their computational inefficiency or substantial memory requirements.

The integrated gradient, closely related to the average gradient, is used in some neural-network explainability techniques (Sundararajan et al., 2017; Khorram et al., 2021; Sattarzadeh et al., 2021). However, the approximation algorithms for the integral of the gradient used in the literature are very inefficient to compute for every parameter update of a model due to the calculation of the Riemann sum (Hughes-Hallett et al., 2021).

## 2 METHODS

### 2.1 ALGORITHM

All of the best and most popular optimizers for training large neural networks rely on the gradient. Consequently, they explicitly ignore how loss function in terms of model parameters behaves in the range between before and after a potential weight update (Fig. 1). The definition of the gradient implies, that it reflects the accurate influence on loss only for learning rates approaching to zero, which does not hold in practice. Consequently, gradient-based optimizers do not calculate the accurate influence on loss of a potential weight update, which may significantly slow down the learning of very deep models with many nonlinear operators, as our experiments show. The average gradient solves the described problem. Our algorithm efficiently approximates the average gradient, providing more reliable information on the update direction that minimizes the loss. The average gradient (contrary to the gradient) is directly proportional to the loss delta (Fig. 1; Equation 14 in Appendix B), hence it accurately describes the influence on loss of a parameter delta.

In our algorithm, given a sequential model, the average gradient is approximated and propagated according to the assumption:

$$\underset{\theta_k}{\mathcal{AVG}}\,\nabla_{\theta_k}\ell \approx \underset{\theta_k}{\mathcal{AVG}}\,\frac{\partial \boldsymbol{x}_k}{\partial \theta_k} \cdot \underset{\boldsymbol{x}_k}{\mathcal{AVG}}\,\frac{\partial \boldsymbol{x}_{k+1}}{\partial \boldsymbol{x}_k} \cdot \ldots \cdot \underset{\boldsymbol{x}_{n-1}}{\mathcal{AVG}}\,\frac{\partial \boldsymbol{x}_n}{\partial \boldsymbol{x}_{n-1}} \cdot \underset{\boldsymbol{x}_n}{\mathcal{AVG}}\,\nabla_{\boldsymbol{x}_n}\ell \tag{1}$$

where $\ell$ is a loss function, $\theta_k$ are parameters of a layer no. $i$ and $(\boldsymbol{x}_k, \boldsymbol{x}_{k+1}, \ldots, \boldsymbol{x}_n)$ are inputs and outputs of subsequent layers of a neural network. The notation $\nabla_{\boldsymbol{x}} f$ refers to the gradient of some function $f$ for an argument $\boldsymbol{x}$, and $\frac{\partial f}{\partial \boldsymbol{x}}$ denotes the Jacobian. The average operator $\mathcal{AVG}$ of gradients or Jacobians is defined in Appendix A; however, it may be intuitive. The averages are aggregated with respect to the parameters of a model ($\theta_k$) or the outputs of subsequent layers ($\boldsymbol{x}_k, \boldsymbol{x}_{k+1}, \ldots, \boldsymbol{x}_n$). The average gradients are propagated in the same manner as the gradients in the standard backpropagation algorithm. The computation based on Equation 1 is fast and memory efficient because the procedure is similar to the standard backpropagation of gradients, which is done according to:

$$\nabla_{\theta_k}\ell = \tfrac{\partial \boldsymbol{x}_k}{\partial \theta_k} \cdot \tfrac{\partial \boldsymbol{x}_{k+1}}{\partial \boldsymbol{x}_k} \cdot \ldots \cdot \tfrac{\partial \boldsymbol{x}_n}{\partial \boldsymbol{x}_{n-1}} \cdot \nabla_{\boldsymbol{x}_n}\ell \tag{2}$$

The version of our algorithm that consists of two iterations (Algorithm 1) first performs the standard backpropagation (Equation 2) through layer outputs $\boldsymbol{x}$, and model parameters $\theta$ along with parameter update of an optimizer (in the experiments it is RMSProp) to new weight values $\theta'$. Then it is assumed that the absolute value of the parameter delta $|\theta - \theta'|$ of the RMSProp optimizer is good enough to retain it. The second backpropagation is performed for eventual negations of update directions only, where, conversely, the *average* gradient is propagated (Algorithm 2). Importantly, the range on which the gradient is averaged equals $[\theta, \theta']$ (between parameters before and after the estimated potential update; Algorithm 3). The average derivatives of each nonlinear activation are calculated as follows:

$$\underset{t\in[x,x']}{\mathcal{AVG}}\, f'(t) = \frac{\int_x^{x'} f'(t)\mathrm{d}t}{x'-x} = \frac{f(x')-f(x)}{x'-x} \tag{3}$$

where $f$ means an activation function (in the experiments it is either ELU or Tanh activation), $x$ means an input scalar assuming forward propagation using the $\theta$ weights, and $x'$ means the corresponding input number assuming forward pass for the $\theta'$. The equation is the one-dimensional analogy of the average gradient and the Jacobian, both of which are defined in Appendix A.

In the case of applying an activation function $f : \mathbb{R} \to \mathbb{R}$, or $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^n$, to a layer output $\boldsymbol{x} = \langle x_1, x_2, ..., x_n\rangle$ (assuming parameters $\theta$), which changes to $\boldsymbol{x}' = \langle x_1', x_2', ..., x_n'\rangle$ during the forward pass with updated parameters $\theta'$:

$$\underset{\boldsymbol{t}\in[\boldsymbol{x},\boldsymbol{x}']}{\mathcal{AVG}}\,\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{t}} = \mathrm{diag}(\langle \underset{t_1\in[x_1,x_1']}{\mathcal{AVG}}\, f'(t_1), \underset{t_2\in[x_2,x_2']}{\mathcal{AVG}}\, f'(t_2), \ldots, \underset{t_n\in[x_n,x_n']}{\mathcal{AVG}}\, f'(t_n)\rangle) \tag{4}$$

where each term $\mathcal{AVG}_{(\cdot)}\, f'(\cdot)$ is defined in Equation 3.

Let us define a typical layer, denoted as $k$, which is parameterized by $\theta_k$. This layer could be a convolutional layer, a fully-connected layer, or another operator that is linear over all or most of its domain. Let us assume that the layer no. $k$ outputs $\boldsymbol{y}_k$, which is then passed to an activation $f_k$. Consequently each part of Equation 1 can be approximated as:

$$\begin{aligned} \mathcal{AVG}_{\boldsymbol{x}_k} \frac{\partial \boldsymbol{x}_{k+1}}{\partial \boldsymbol{x}_k} &= \mathcal{AVG}_{\boldsymbol{x}_k} \frac{\partial f_k}{\partial \boldsymbol{x}_k} \approx \mathcal{AVG}_{\boldsymbol{x}_k} \frac{\partial \boldsymbol{y}_k}{\partial \boldsymbol{x}_k} \cdot \mathcal{AVG}_{\boldsymbol{t}\in[\boldsymbol{y}_k,\boldsymbol{y}'_k]} \frac{\partial f}{\partial \boldsymbol{t}} \\ \mathcal{AVG}_{\theta_k} \frac{\partial \boldsymbol{x}_{k+1}}{\partial \theta_k} &= \mathcal{AVG}_{\theta_k} \frac{\partial f_k}{\partial \theta_k} \approx \mathcal{AVG}_{\theta_k} \frac{\partial \boldsymbol{y}_k}{\partial \theta_k} \cdot \mathcal{AVG}_{\boldsymbol{t}\in[\boldsymbol{y}_k,\boldsymbol{y}'_k]} \frac{\partial f}{\partial \boldsymbol{t}} \end{aligned} \tag{5}$$

where the approximation, instead of equality, is the consequence of chaining averages of Jacobians, which can be proven analogously to Equation 1 (see Appendix B). The average operator $\mathcal{AVG}$ of Jacobians is defined in Appendix A. $\mathcal{AVG}_{\boldsymbol{t}\in[\boldsymbol{y}_{k+1},\boldsymbol{y}'_{k+1}]} \frac{\partial f}{\partial \boldsymbol{t}}$ is defined in Equation 4. Generally, the vast majority of applied neural network operators are either nonlinear activations or linear functions in by far most of their domains (e.g., max pooling, convolution, fully connected, or ReLU). In the case of the nonlinear activations, equations no. 3 and 4 are used to compute the average Jacobians. For linear transformations, such as $\boldsymbol{y}_k(\boldsymbol{x}_k)$ and $\boldsymbol{y}_k(\theta_k)$, the average gradients and Jacobians are easy and fast to compute. However, for implementation simplicity and a slight speedup of computations, broader estimates of the average Jacobians from Equation 5 are applied:

$$\begin{aligned} \mathcal{AVG}_{\boldsymbol{x}_k} \frac{\partial \boldsymbol{x}_{k+1}}{\partial \boldsymbol{x}_k} &= \mathcal{AVG}_{\boldsymbol{x}_k} \frac{\partial f_k}{\partial \boldsymbol{x}_k} \approx \frac{\partial \boldsymbol{y}_k}{\partial \boldsymbol{x}_k} \cdot \mathcal{AVG}_{\boldsymbol{t}\in[\boldsymbol{y}_k,\boldsymbol{y}'_k]} \frac{\partial f}{\partial \boldsymbol{t}} \\ \mathcal{AVG}_{\theta_k} \frac{\partial \boldsymbol{x}_{k+1}}{\partial \theta_k} &= \mathcal{AVG}_{\theta_k} \frac{\partial f_k}{\partial \theta_k} \approx \frac{\partial \boldsymbol{y}_k}{\partial \theta_k} \cdot \mathcal{AVG}_{\boldsymbol{t}\in[\boldsymbol{y}_k,\boldsymbol{y}'_k]} \frac{\partial f}{\partial \boldsymbol{t}} \end{aligned} \tag{6}$$

which use the non-averaged Jacobian $\frac{\partial \boldsymbol{y}_k}{\partial \boldsymbol{x}_k}$. Therefore, intuitively, the broad estimation of $\mathcal{AVG}_{\boldsymbol{x}_k} \frac{\partial \boldsymbol{x}_{k+1}}{\partial \boldsymbol{x}_k}$ is approximately between $\frac{\partial \boldsymbol{x}_{k+1}}{\partial \boldsymbol{x}_k}$ and $\mathcal{AVG}_{\boldsymbol{x}_k} \frac{\partial \boldsymbol{x}_{k+1}}{\partial \boldsymbol{x}_k}$, and analogously for $\mathcal{AVG}_{\theta_k} \frac{\partial \boldsymbol{x}_{k+1}}{\partial \theta_k}$.

**Algorithm 1** *Simplified algorithm version for 2 iterations.* Back and forward propagation would be called two times in optimal implementation, where memory requirement would be the same as for Adam optimizer.

```
Input: $model$: Neural Network Model
       $dataset$: Training Dataset
       $lossFn$: Loss Function
       $optimizer$: Optimizer
for all $batch \in dataset$ do
    $modelOutput \leftarrow model(batch.\mathrm{x})$ {It is assumed that $model$'s layers' results are kept inside $model$}
    $modelLoss \leftarrow LossFn(modelOutput, batch.\mathrm{y})$
    $modelCopy \leftarrow model$ {Copy $model$}
    $modelCopyOutput \leftarrow modelCopy(batch.\mathrm{x})$ {This inference is redundant if $modelCopy$ gets also intermediate layers' results copied}
    $modelCopyLoss \leftarrow LossFn(modelCopyOutput, batch.\mathrm{y})$ {This computation is also redundant, since it is the same as $modelLoss$}
    $Backpropagate(modelCopyLoss)$ {Compute the gradients using the standard backpropagation procedure. Assume that the gradients are stored inside $modelCopy$}
    $optimizer.\mathrm{Step}(modelCopy)$ {Perform weight update on $modelCopy$ (using the gradients stored inside $modelCopy$)}
    $modelCopy(batch.\mathrm{x})$ {Execute inference to store new layer-wise results in $modelCopy$}
    AveragedBackpropagation$(model, modelCopy, modelLoss)$ {The procedure is described as Algorithm 2. The parameters of the $model$ are modified within}
end for
```

An algorithm version with $n$ backpropagation iterations computes $(n-1)$ times the approximated gradient average, each time based on the previous. The intuition behind this is that a better estimate of the averaged derivatives of nonlinear activations is computed after every iteration (Equation 3; backpropagated according to equations no. 4, 6 and 1). Consequently, each time a more precise estimate of the optimal parameter update $(\Delta\theta)^*$ is obtained (where optimality means that the average gradient is accurately estimated and the parameter update is compliant with it). Therefore, once again

**Algorithm 2** *Averaged Backpropagation Algorithm* calculates the approximation of the average gradient.

```
procedure AVERAGEDBACKPROPAGATION(
$model$: Neural Network Model
$modelAfterUpdate$: $model$ After Candidate Update of
    Parameters
$modelLoss$: $model$'s Loss)
    $Backpropagate(modelLoss,$
        $model$.Layers.Last().Output) {Compute the gradient of $modelLoss$ in terms of the last layer's output. Let us assume that the gradient is assigned to the $grad$ property of the output variable ($model$.Layers.Last().Output)}
    for $index \leftarrow (Count(model.\text{Layers}) - 1)$ to $0$ do
        $\theta = model.\text{Layers}[index].\theta$ {To simplify notation}
        if $IsNonlinear(model.\text{Layers}[index])$ then {Calculation of either the gradient or its average, which corresponds to the terms in Equation 6}
            BackpropagateThroughNonlinearLayer($model$.Layers[$index$].Output,
                $model$.Layers[$index$].Input, $modelAfterUpdate$.Layers[$index$].Output,
                $modelAfterUpdate$.Layers[$index$].Input) {Procedure described as Algorithm 3. Let us assume that activations are separate layers (like in equations no. 4 and 6)}
        else
            $Backpropagate(model.\text{Layers}[index].\text{Output}, model.\text{Layers}[index].\text{Input})$ {The typical backpropagation procedure. It propagates the gradient through a layer. Let us assume that the gradient is assigned to the $grad$ property of the input variable}
            $\theta.averagedGrad \leftarrow \theta.grad$ {In this case, for a linear layer, the gradient is treated as its average (compare equations no. 6 and 5)}
        end if
        $\theta' = modelAfterUpdate.\text{Layers}[index].\theta$ {Notation simplification}
        $\theta \leftarrow \theta + |\theta' - \theta| \cdot sgn(\theta.\text{averagedGrad})$ {Update by the absolute value of $optimizer$'s update from Algorithm 1: $|\theta' - \theta|$, but in the direction of the approximated gradient average $sgn(\theta.\text{averagedGrad})$}
    end for
end procedure
```

the average gradient can be refined to more precisely match the parameter update $(\Delta\theta)^*$, and so on (in a loop).

The $n$-iteration version of the method is labeled as Algorithm 4 in Appendix G (for two iterations it is a little slower than Algorithm 1 due to additional model-state copies, moreover it is more complex). The optimal implementation of the $n$-iteration algorithm variant would be slightly more than $n$ times slower than the gradient-based RMSprop training, where $n$ is the number of iterations. There are exactly $n$ backward passes, optimally $n$ inferences, and some additional copy operations $C$ of a model (optimally $|C| \leq n + 1$, but Algorithm 4 in Appendix G is suboptimal in this respect). However, the copies are generally significantly faster than forward or backward passes, because it is just needed to copy blocks of data, that are not bigger than the memory used during forward or backward pass. Furthermore, creating the copies by saving results of weight updates directly into different memory addresses only slightly increases the execution time.

An interesting way of comparing the gradient-based RMSProp optimization with our algorithm is to examine the average loss deltas for all weight updates of both algorithms. The first iteration of our method is the gradient-based RMSProp procedure, hence the change in loss for RMSProp $\Delta_{RMSProp}$ is known for both the same model parameters and data as in the case of the loss delta of our method. Therefore, the sum of loss differences after the updates of both approaches can be easily and measurably compared relatively to the sum of loss deltas of RMSProp:

$$\mathcal{RD}_{AG,RMSProp} = \frac{\mathcal{AVG}_{b\in B}\,\Delta_{AG} - \mathcal{AVG}_{b\in B}\,\Delta_{RMSProp}}{|\,\mathcal{AVG}_{b\in B}\,\Delta_{RMSProp}|}$$

$$= \frac{\sum_{b\in B}\left(\ell_b(\theta'_{AG,b}) - \ell_b(\theta_b)\right)}{|\sum_{b\in B}\left(\ell_b(\theta'_{RMSProp,b}) - \ell_b(\theta_b)\right)|} - sgn(\sum_{b\in B}\left(\ell_b(\theta'_{RMSProp,b}) - \ell_b(\theta_b)\right)) \tag{7}$$

**Algorithm 3** *Backpropagation Through Nonlinear Layer.* It is assumed that each input number influences a corresponding single output scalar. This is because, in the experiments, the only operators assumed to be nonlinear during backpropagation of the gradient average are certain activation functions: $\mathbb{R} \to \mathbb{R}$).

```
procedure BackpropagateThroughNonlinearLayer($LayerOutput$: Tensor
$LayerInput$: Tensor
$LayerOutputAfterUpdate$: Tensor
$LayerInputAfterUpdate$: Tensor)
  $f \leftarrow$ Layer function
  for all $(outputNum, inputNum, outputNumUpdated, inputNumUpdated) \in Zip($
      $LayerOutput, LayerInput, LayerOutputAfterUpdate, LayerInputAfterUpdate)$ do
  {The commonly used $Zip$ function illustrates iterating through multiple tensors at once}
    if $|inputNumUpdated - inputNum| > \epsilon$ then {Check if the difference in the inputs is higher than a tiny constant $\epsilon$. The condition prevents division by zero. In the experiments $\epsilon \approx 1.19\text{e}{-7}$}
      $\mathcal{AVG}_{x \in [inputNum, inputNumUpdated]} f'(x) = \frac{outputNumUpdated - outputNum}{inputNumUpdated - inputNum}$ {Eq. 3 and 4}
      $inputNum$.averagedGrad $\leftarrow \mathcal{AVG}_x f'(x) \cdot outputNum$.averagedGrad
      {Propagate the average gradient backward using the chain rule. Equations 3 and 4 define the term $\mathcal{AVG}_{\boldsymbol{t} \in [\boldsymbol{y}_k, \boldsymbol{y}'_k]} \frac{\partial f}{\partial \boldsymbol{t}}$ in Equation 6, which is part of Equation 1}
    else
      $inputNum$.averagedGrad $\leftarrow f'(inputNum) \cdot outputNum$.averagedGrad {In this case $inputNum \approx inputNumUpdated$, thus $\mathcal{AVG}_{x \in [\dots]} f'(x) \approx f'(inputNum)$ for activation functions. The backpropagation towards input complies with the chain rule (equations no. 6 and 1)}
    end if
  end for
end procedure
```

$\mathcal{RD}_{AG,RMSProp}$ is the relative difference in avg. loss deltas of $RMSProp$ and the method based on the average gradient ($AG$). The $\mathcal{AVG}$ operator denotes the arithmetic average. $B$ is the set of all batches. $\mathit{\Delta}_{AG,b}$ is the loss delta assuming a batch $b$ after our algorithm's update of model parameters $\theta_b$ to new values $\theta'_{AG,b}$. Notation for RMSProp is analogous. $\ell_b$ is the loss, assuming data of a batch $b$. $sgn$ is the sign function. $\mathcal{RD}$ would not be as useful when using momentum because the metric compares the aggregated loss of a single batch per parameter update, whereas momentum contributes to a decrease in loss over many batches per a single parameter update. Without the momentum, $\mathcal{RD}$ significantly increases the statistical confidence in comparing training algorithms because, for *the same* model weights, the losses are compared for each weight update. Keeping the same parameter values for each loss delta reduces the variance of $\mathcal{RD}$, resulting in a decrease in errors when comparing methods.

### 2.2 MODELS AND TRAINING

Our algorithm was tested on two different models with nonlinear ELU (Clevert et al., 2015) and Tanh activations. Model A has a small number of layers (Table 1), and the second one, Model B, is much deeper, with 30 nonlinear layers (Table 2; not counting max pooling as nonlinear). It was assumed that Model A is trained for 15 epochs, while Model B – 500 in the case of the gradient-based RMSProp training, and 300 for our method. Grid search was used to find the optimal learning rates for the standard RMSProp training over the course of all 500 epochs, while our method was optimized only for 200 epochs (out of 300 during testing). The objective of the hyperparameter search was to minimize the loss that is the smallest over a training. The results of the search for optimal learning rates are shown in Table 3. The epoch counts are tailored to ensure that the training achieves minimal or near-minimal test loss values before the final epoch of the gradient-based RMSProp training. The only loss function used in this research is cross-entropy loss, and the batch size is set to 128 in all experiments.

Table 1: *Model A*

| Layers | Output Shape | Parameter Count |
|---|---|---|
| Convolution 2D $(3 \times 3)$ + ELU | $(8, 26, 26)$ | 80 |
| Convolution 2D $(3 \times 3)$ + ELU | $(8, 24, 24)$ | 584 |
| Convolution 2D $(5 \times 5)$ + ELU stride $= 2$ padding $= 2$ | $(16, 12, 12)$ | 3216 |
| Convolution 2D $(3 \times 3)$ + ELU | $(16, 10, 10)$ | 2320 |
| Convolution 2D $(3 \times 3)$ + ELU | $(16, 8, 8)$ | 2320 |
| Convolution 2D $(5 \times 5)$ + ELU stride $= 2$ padding $= 2$ | $(16, 4, 4)$ | 6416 |
| Flatten | 256 | |
| Linear + Softmax | 10 | 2570 |
| | | 17506 |

Table 2: *Model B* is designed to test the performance of our algorithm on deep neural networks to achieve a reasonable time of many trainings for statistical significance of the results. The practicality of the architecture is not prioritized.

| Layers | Output Shape | Parameter Count |
|---|---|---|
| Convolution 2D $(3 \times 3)$ + ELU | $(8, 26, 26)$ | 80 |
| Max Pooling 2D $(2 \times 2)$ | $(8, 13, 13)$ | |
| Convolution 2D $(3 \times 3)$ + ELU | $(16, 11, 11)$ | 1168 |
| Max Pooling 2D $(2 \times 2)$ | $(16, 5, 5)$ | |
| Flatten | 400 | |
| Linear + Tanh | 10 | 4010 |
| $\mathbf{26}\times$ Linear + Tanh | 10 | $\mathbf{26}\times 110$ |
| Linear + Softmax | 10 | 330 |
| | | 8228 |

The learning algorithms were tested on two popular image datasets: MNIST (LeCun & Cortes, 2010) and Fashion MNIST (Xiao et al., 2017). Both datasets have the same input size $(28 \times 28 \times 1)$, but their image characteristics are *significantly* different, allowing the performance of the training algorithm to be tested across various domains. Moreover, since the method does not have any hyperparameters apart from the learning rate, it is less likely to overfit to a specific experimental setup (model, dataset, and learning rate) and show good results on it, while experiencing significant performance drops on other setups.

## 3 RESULTS

For the shallow model A, all of the training algorithms are approximately equal (Fig. 2a, Fig. 2b). The relative difference in summed loss deltas (Equation 7) revealed that the algorithm based on the average gradient is only marginally better than the standard RMSprop according to $\mathcal{RD} = 1.20\text{e}{-3} \pm 2.7\text{e}{-4}$ ($0.12\%$ faster minimization of loss with $0.027\%$ of SEM error) on MNIST and $\mathcal{RD} = 5.86\text{e}{-3} \pm 2.79\text{e}{-3}$ on Fashion MNIST in the case of two iterations. For five iterations, $\mathcal{RD} = 6.47\text{e}{-4} \pm 9.8\text{e}{-5}$ on MNIST and $\mathcal{RD} = 2.37\text{e}{-3} \pm 4.5\text{e}{-4}$ on Fashion MNIST. $\mathcal{RD}$ of the first epoch has the highest influence on the scores above. Nevertheless, excluding the first epoch, the values of the metric remain positive.

The results of Model B are much more interesting. The version of the algorithm with two iterations is about three times faster at minimizing the median of training losses on both datasets (Fig. 2c; Fig. 2d). On the other hand, the mean training losses tend to fluctuate frequently in the plots, showing higher oscillations for our algorithm. However, this occurs due to the lower training count than in the case of the gradient-based RMSProp, hence an anomaly during a single training may significantly increase the average loss. Moreover, the mean losses still tend to be considerably lower than for the standard RMSProp training. Despite the minority of epochs with high oscillations, the method utilizing the average gradient is approximately two to three times faster in minimizing the mean loss, although this is not clearly visible in the plots. Furthermore, for both versions of our algorithm on both datasets, during from $49.3\%$ to $70\%$ of epochs, the average training loss was lower with statistical significance (SEM) than for the gradient-based RMSProp. Conversely, our algorithm was worse in that respect during from $0.667\%$ to $2.33\%$ of epochs with statistical significance. The average of minimal training losses on MNIST for the five iterations is $0.0393 \pm 0.0058$, which is significantly lower than $0.0883 \pm 0.0117$ for the standard RMSProp. Meanwhile, the two iterations are also perform better than the gradient-based RMSProp, but without statistical significance, achieving $0.0747 \pm 0.0188$. Even better averages of minimal training losses were obtained on Fashion MNIST, with the five-

Table 3: *Learning rates*. All hyperparameter searches of Model A consist of five trainings for each learning rate (LR), while in the case of Model B, it is one training, unless stated otherwise. For Model B, the losses do not directly predict the performance of the methods, because different epoch counts are used between the methods. The standard error of the mean is used as the confidence range for the losses, while for the LRs, the maximum distance to the next best LRs on both sides represents the errors. The LRs used in the experiments are listed in the ”Learning Rate” column.

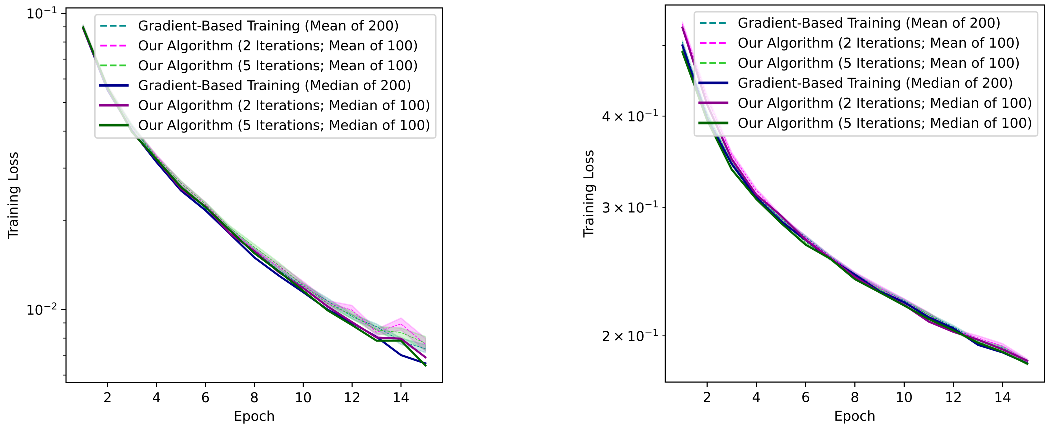
| Dataset | Model | Method | Learning Rate | The Most Important Hyperparameter Search Results [Learning Rate: Avg. of Min. Training Loss] |
|---|---|---|---|---|
| MNIST | Model A | RMSProp | **8e−4** | 6e−4 : 8.04e−3; 7e−4 : 6.48e−3; **8e−4 : 5.69e−3**<br>9e−4 : 5.94e−3; 10e−4 : 7.70e−3; 11e−4 : 7.39e−3 |
| | | 2 Iterations | **8e−4** | **8e−4 : 0.00555**; 9e−4 : 0.00692; 1e−3 : 0.00832 |
| | | 5 Iterations | **8e−4** | **8e−4 : 0.00514**; 9e−4 : 0.00580; 1e−3 : 0.00678 |
| | Model B | RMSProp | **2.5e−4** | 1.5e−4 : 0.194; 2e−4 : 0.0979; **2.5e−4 : 0.0651**<br>3e−4 : 0.0683; 3.5e−4 : 0.191; 4e−4 : 0.0759<br>The best learning rate of the search *after* the experiments (10 trainings per LR in $\{1.5\text{e}{-4}, 2\text{e}{-4}, \ldots, 5.5\text{e}{-4}\}$): $(3.5\text{e}{-4} \pm 1.5\text{e}{-4}) : (0.0856 \pm 0.0139)$, (matches the performance in our experiments in Section 3)<br>The loss for a high learning rate (10 trainings):<br>$1.5\text{e}{-3} : (2.09 \pm 0.05)$ |
| | | 2 Iterations | **7.5e−4** | The learning rate is guessed |
| | | 5 Iterations | **7.5e−4** | The learning rate is guessed |
| Fashion MNIST | Model A | RMSProp | **1.5e−3** | 1e−3 : 0.201; 1.25e−3 : 0.186; **1.5e−3 : 0.183**<br>1.75e−3 : 0.189; 2e−3 : 0.183; 2.25e−3 : 0.193 |
| | | 2 Iterations | **1.9e−3** | 1.8e−3 : 0.186; **1.9e−3 : 0.179**; 2e−3 : 0.180 |
| | | 5 Iterations | **1.5e−3** | **1.5e−3 : 0.178**; 1.6e−3 : 0.179; 1.7e−3 : 0.200 |
| | Model B | RMSProp | **3e−4** | 2e−4 : 0.356; 2.5e−4 : 0.331; **3e−4 : 0.285**<br>3.5e−4 : 0.349; 4e−4 : 0.487; 4.5e−4 : 0.459<br>The best learning rate of the search *after* the experiments (10 trainings per LR in $\{2\text{e}{-4}, 2.5\text{e}{-4}, \ldots, 6\text{e}{-4}\}$): $(4\text{e}{-4} \pm 1.5\text{e}{-4}) : (0.318 \pm 0.016)$, (matches the performance in our experiments in Section 3)<br>The loss for a high learning rate (10 trainings):<br>$9\text{e}{-4} : (0.641 \pm 0.168)$ |
| | | 2 Iterations | **9e−4** | 6e−4 : 0.330; **9e−4 : 0.242**; 1.2e−3 : 0.355 |
| | | 5 Iterations | **9e−4** | **9e−4 : 0.243**; 1.5e−3 : 0.276 |

iteration and two-iteration versions achieving $0.254 \pm 0.017$ and $0.257 \pm 0.014$ respectively, compared to $0.314 \pm 0.008$ by the gradient-based training.

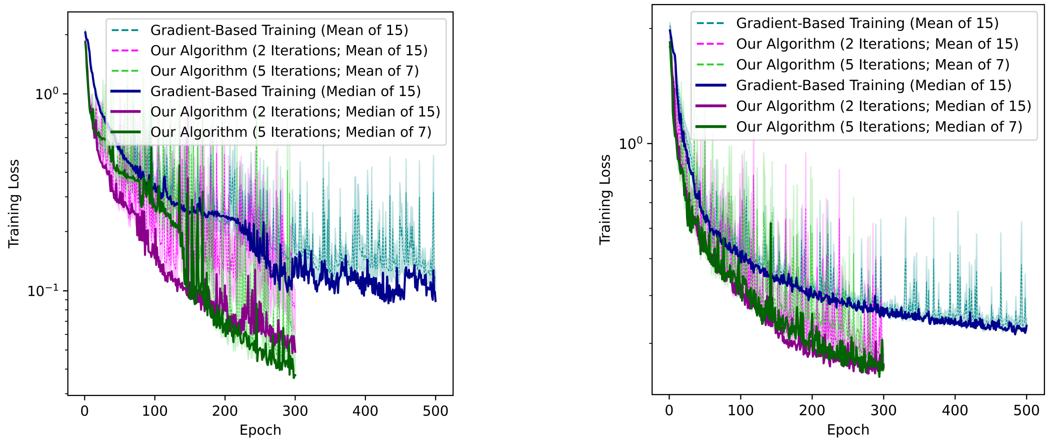
Plots of the test losses of Model B look very similar to the training losses (Appendix C), showing significant improvements in generalization, which correspond to the lower training losses. On MNIST, the average of best accuracies over training for five iterations is equal to $(97.87 \pm 0.09)\%$, which is significantly higher than $(96.80 \pm 0.78)\%$ and $(96.75 \pm 0.55)\%$ for the two-iteration version and gradient-based algorithm, respectively. On Fashion MNIST, the analogous results are $(88.09 \pm 0.35)\%$, $(87.54 \pm 0.55)\%$ and $(86.57 \pm 0.29)\%$, respectively. Appendix D presents the accuracy plots.

For Model B, the $\mathcal{RD}$ metric (Equation 7) provides a very high confidence of superiority of the average gradient for the high learning rates used for the trainings based on the average gradient (Table 3). On MNIST for two and five iterations, it equals $10.41 \pm 1.94$ and $1.43 \pm 0.29$, respectively. On Fashion MNIST, it is $0.58 \pm 0.14$ and $0.24 \pm 0.04$ for both variants, respectively.

Importantly, for Model B, the average-gradient algorithm dominated also for the learning rates that are optimal for the standard RMSProp training. Multiple metrics favored our algorithm with statistical significance , i.e., $\mathcal{RD} \in [0.0611 \pm 0.0004, 1.07 \pm 0.31]$, despite training counts equal to only two or three (for each of the four experiments).



(a) *Model A on MNIST* (b) *Model A on Fashion MNIST*



(c) *Model B on MNIST* (d) *Model B on Fashion MNIST*

Figure 2: *Training losses*. Only mean curves contain confidence ranges (SEM).

In the case of Model B, our implementation of the two-iteration variant of the algorithm based on the average gradient (Alg. 1) is nearly three times slower (per epoch) than the training based on the gradient, while the five iterations (Alg. 4 in Appendix G) are almost eight times slower. (The estimated runtime of optimal implementation is slightly more than two times longer for the two iterations per epoch when compared to the gradient-based RMSProp, and around six to seven times longer for the five iterations.)

## 4 DISCUSSION

### 4.1 CONCLUSIONS

The algorithm based on the average gradient offers significant benefits when compared to the standard RMSProp training for the deep model with many nonlinear activations: (a) About a threefold increase in sample efficiency in terms of median loss, and about two to three times faster mean loss reduction. This is reached by only two iterations, which optimally require a little more than double the time of computation per epoch in comparison with the gradient-based RMSProp training. Meanwhile, our suboptimal implementation of the two-iteration version of the algorithm needs nearly three times more runtime per epoch than the training based on the gradient. Therefore, the presented method is not only more sample-efficient, but it is also faster and saves energy. (b) Outstanding performance on higher learning rates, which may offer significant benefits in terms of both electricity and time spent on hyperparameter searches. (c) Considerably better generalization, at least in a reasonable epoch count.

The $\mathcal{RD}$ (Equation 7) confirms the outstanding results of the other measures. The score of $\mathcal{RD} = 10.41 \pm 1.94$, achieved by the two iterations on MNIST, corresponds to the average speed of batch-loss minimization that is $(1141 \pm 194)\%$ of the speed of the gradient-based RMSProp while using the same absolute values of weight updates. In the other cases, the average speed of batch-loss minimization ranges from $(124 \pm 3.6)\%$ to $(243 \pm 29)\%$. Therefore, even a relatively slight speedup in batch-loss minimization (such as $(24 \pm 3.6)\%$) may contribute to a significantly higher gain in sample efficiency, increasing it by two to three times. Moreover, it is crucial to note that the mentioned gain occurs at learning rates that are three times higher than the optimal rates for gradient-based training. Generally, high learning rate values may enable rapid learning because model parameters are adjusted faster.

Nevertheless, the average gradient is also superior in terms of the average speed of batch-loss minimization when using the optimal learning rates for gradient-based training. The algorithm's potential in handling very deep models with numerous nonlinear layers is further confirmed by its dominance during training with the optimal learning rate for gradient-based RMSProp. Despite the low sample sizes, multiple measures are statistically significant, i.e., $\mathcal{RD}$.

Surprisingly, the algorithm version with five iterations is worse than the two iterations according to $\mathcal{RD}$ with higher statistical confidence than for other measures. Across all experiments, the variant is computationally inefficient in terms of the resources required to reduce the loss to a certain level.

In the case of the shallow model with nonlinear ELU activations, the method is only marginally better (with statistical significance) than the standard gradient-based RMSProp training. This behavior is expected due to the scaling properties of the algorithm (Appendix E).

The method is promising and provides a perspective on multiple interesting further experiments. For some applications, the benefits of the method may surpass those for Model B, e.g., in the case of deeper models. The method may also contribute to the training of large models in the future, where sample efficiency is needed to learn new tasks on the fly, akin to how people or some animals do.

See Appendix B for mathematical guarantees of the average gradient. Refer to Appendix F for the limitations of our algorithm in estimating the average gradient.

### 4.2 FUTURE WORK

Interesting directions for further experiments include: (a) Computing the average gradients over a much larger range than that of a parameter update to capture the global trend of the loss landscape. (b) More accurate approximation of the average Jacobians using Equation 5 instead of Equation 6. This would enable computing the average Jacobians of linear operators. Therefore, the algorithm based on the average gradient may enhance trainings of deep models without nonlinear activations. Moreover, the usage of Equation 5 may further improve the performance in the case of many nonlinear activations because of the increased precision in approximating the average gradient. (c) Incorporation of the momentum into our algorithm. Preferably Nesterov momentum (Dozat, 2016) should be used. If not, the average gradient would also be calculated for the momentum part of the update step. This could often reverse the direction of the momentum for a model parameter, thereby impairing the effectiveness of the entire momentum procedure. (d) Development of similar algorithms, but with update steps, that, for a given model parameter, vary in size over the iterations of the average-gradient computation. By adjusting the step size of each model parameter to the absolute value of the average gradient, the learning process may be enhanced. (e) Tests of the method on large and very deep architectures, that are used in practice and contain many nonlinear layers. (f) More research on how the method scales up, also in relation to the number of neurons in layers of neural networks. (g) Experiments with learning without forgetting (Li & Hoiem, 2017) and online learning. Sample efficiency may be very beneficial there.

## 5 REPRODUCIBILITY STATEMENT

We put emphasis on providing detailed descriptions of all experiments. The algorithms (Alg. 4 in Appendix G and Alg. 1 in Section 2, with subprocedures labeled as Alg. 2 and Alg. 3) are described in detail in Section 2.1. The models (Tables 1 and 2), the learning rates (Table 3), and all other important experiment settings are described in Section 2.2. The code, along with environment settings, is available under [...]. Appendix B contains one of our most important theoretical results: the proof of Equation 1 and its superiority over the gradient in minimizing the batch loss by accurately indicating how each model parameter individually contributes to the change in the batch loss (Equation 14). The proven potential for batch-loss minimization is verified not only by the $\mathcal{RD}$ metric with high statistical significance but also by comparisons of training losses and other metrics (Section 3).

## REFERENCES

Albert S Berahas, Jorge Nocedal, and Martin Takác. A multi-batch l-bfgs method for machine learning. *Advances in Neural Information Processing Systems*, 29, 2016.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

Timothy Dozat. Incorporating nesterov momentum into adam. 2016.

Deborah Hughes-Hallett, Andrew M Gleason, Patti Frazer Lock, and Daniel E Flath. *Applied calculus*. John Wiley & Sons, 2021.

Nikhil Ketkar. Stochastic gradient descent. *Deep learning with Python: A hands-on introduction*, pp. 113–132, 2017.

Saeed Khorram, Tyler Lawson, and Li Fuxin. igos++ integrated gradient optimized saliency by bilateral perturbations. In *Proceedings of the Conference on Health, Inference, and Learning*, pp. 174–182, 2021.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

Robert Kirk, Ishita Mediratta, Christoforos Nalmpantis, Jelena Luketina, Eric Hambro, Edward Grefenstette, and Roberta Raileanu. Understanding the effects of rlhf on llm generalisation and diversity. *arXiv preprint arXiv:2310.06452*, 2023.

Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL `http://yann.lecun.com/exdb/mnist/`.

Zhizhong Li and Derek Hoiem. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947, 2017.

Yanli Liu, Yuan Gao, and Wotao Yin. An improved analysis of stochastic gradient descent with momentum. *Advances in Neural Information Processing Systems*, 33:18261–18271, 2020.

Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017.

R OpenAI. Gpt-4 technical report. *ArXiv*, 2303, 2023.

Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM journal on control and optimization*, 30(4):838–855, 1992.

David Ruppert. Efficient estimations from a slowly convergent robbins-monro process. Technical report, Cornell University Operations Research and Industrial Engineering, 1988.

Sam Sattarzadeh, Mahesh Sudhakar, Konstantinos N Plataniotis, Jongseong Jang, Yeonjeong Jeong, and Hyunwoo Kim. Integrated grad-cam: Sensitivity-aware visual explanation of deep convolutional networks via integrated gradient-based scoring. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1775–1779. IEEE, 2021.

Xu Sun, Hisashi Kashima, Takuya Matsuzaki, and Naonori Ueda. Averaged stochastic gradient descent with feedback: An accurate, robust, and fast training method. In *2010 IEEE international conference on data mining*, pp. 1067–1072. IEEE, 2010.

Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *International conference on machine learning*, pp. 3319–3328. PMLR, 2017.

Hong Hui Tan and King Hann Lim. Review of second-order optimization techniques in artificial neural networks backpropagation. In *IOP conference series: materials science and engineering*, volume 495, pp. 012003. IOP Publishing, 2019.

Tijmen Tieleman, Geoffrey Hinton, et al. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

Ziyang Wei, Wanrong Zhu, and Wei Biao Wu. Weighted averaged stochastic gradient descent: Asymptotic normality and optimality. *arXiv preprint arXiv:2307.06915*, 2023.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

# A DEFINITION OF AVERAGE GRADIENT/JACOBIAN

Let us define the average gradient of a function $f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}$ for some row vector $\boldsymbol{x} \in [\boldsymbol{a}, \boldsymbol{b}]$ (the formula is analogous to the one-dimensional case in Equation 3):

$$\underset{\boldsymbol{x}\in[\boldsymbol{a},\boldsymbol{b}]}{\mathcal{AVG}} \nabla_{\boldsymbol{x}} f = (\boldsymbol{b}-\boldsymbol{a})^{\circ -1} \circ \int_{\boldsymbol{a}}^{\boldsymbol{b}} \nabla_{\boldsymbol{x}} f \, d\boldsymbol{x} = (\boldsymbol{b}-\boldsymbol{a})^{\circ -1} \circ \int_{0}^{1} \nabla_{\boldsymbol{a}+t\cdot(\boldsymbol{b}-\boldsymbol{a})} f \, dt \tag{8}$$

where $\circ$ denotes the elementwise operation of either multiplication or inversion ($(\cdot)^{\circ -1}$). However, the cases of vector elements where division by zero occurs are handled differently, using the partial derivative $\frac{\partial f}{\partial x_i}$:

$$\forall i : b_i - a_i = 0 \implies \underset{x_i\in[a_i,b_i]}{\mathcal{AVG}} \frac{\partial f}{\partial x_i} = \frac{\partial f}{\partial a_i} \tag{9}$$

If $\boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}^m$, then using to Equation 8:

$$\begin{aligned}
\underset{\boldsymbol{x}\in[\boldsymbol{a},\boldsymbol{b}]}{\mathcal{AVG}} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} &= \begin{bmatrix} \mathcal{AVG}_{\boldsymbol{x}\in[\boldsymbol{a},\boldsymbol{b}]} \nabla_{\boldsymbol{x}} f_1 \\ \mathcal{AVG}_{\boldsymbol{x}\in[\boldsymbol{a},\boldsymbol{b}]} \nabla_{\boldsymbol{x}} f_2 \\ \cdots \\ \mathcal{AVG}_{\boldsymbol{x}\in[\boldsymbol{a},\boldsymbol{b}]} \nabla_{\boldsymbol{x}} f_m \end{bmatrix} = \begin{bmatrix} (\boldsymbol{b}-\boldsymbol{a})^{\circ -1} \circ \int_{\boldsymbol{a}}^{\boldsymbol{b}} \nabla_{\boldsymbol{x}} f_1 \, d\boldsymbol{x} \\ (\boldsymbol{b}-\boldsymbol{a})^{\circ -1} \circ \int_{\boldsymbol{a}}^{\boldsymbol{b}} \nabla_{\boldsymbol{x}} f_2 \, d\boldsymbol{x} \\ \cdots \\ (\boldsymbol{b}-\boldsymbol{a})^{\circ -1} \circ \int_{\boldsymbol{a}}^{\boldsymbol{b}} \nabla_{\boldsymbol{x}} f_m \, d\boldsymbol{x} \end{bmatrix} \\
&= \begin{bmatrix} (\boldsymbol{b}-\boldsymbol{a})^{\circ -1} \\ (\boldsymbol{b}-\boldsymbol{a})^{\circ -1} \\ \cdots \\ (\boldsymbol{b}-\boldsymbol{a})^{\circ -1} \end{bmatrix} \circ \begin{bmatrix} \int_0^1 \nabla_{\boldsymbol{a}+t\cdot(\boldsymbol{b}-\boldsymbol{a})} f_1 \, dt \\ \int_0^1 \nabla_{\boldsymbol{a}+t\cdot(\boldsymbol{b}-\boldsymbol{a})} f_2 \, dt \\ \cdots \\ \int_0^1 \nabla_{\boldsymbol{a}+t\cdot(\boldsymbol{b}-\boldsymbol{a})} f_m \, dt \end{bmatrix}
\end{aligned} \tag{10}$$

Again, the cases of vector elements where division by zero occurs are handled as follows:

$$\forall i : b_i - a_i = 0 \implies \underset{x_i\in[a_i,b_i]}{\mathcal{AVG}} \frac{\partial \boldsymbol{f}}{\partial x_i} = \frac{\partial \boldsymbol{f}}{\partial a_i} \tag{11}$$

# B PROOF OF EQUATION 1 AND ITS LOSS-MINIMIZATION POTENTIAL

## B.1 DEFINITION AND PROPERTIES OF AVERAGE GRADIENT OF LOSS

Using Equation 2, the average gradient $\mathcal{AVG}_{\theta_k} \nabla_{\theta_k} \ell$ can be defined without the approximation given in Equation 1:

$$\underset{\theta_k}{\mathcal{AVG}} \nabla_{\theta_k} \ell = \underset{(\theta_k, \boldsymbol{x}_k, \boldsymbol{x}_{k+1}, \ldots, \boldsymbol{x}_n)}{\mathcal{AVG}} \left(\frac{\partial \boldsymbol{x}_k}{\partial \theta_k} \cdot \frac{\partial \boldsymbol{x}_{k+1}}{\partial \boldsymbol{x}_k} \cdot \ldots \cdot \frac{\partial \boldsymbol{x}_n}{\partial \boldsymbol{x}_{n-1}} \cdot \nabla_{\boldsymbol{x}_n} \ell\right) \tag{12}$$

where multiple variables are under the average operator $(\theta_k, \boldsymbol{x}_k, \boldsymbol{x}_{k+1}, \ldots, \boldsymbol{x}_n)$. There are numerous ways to define how $(\boldsymbol{x}_k, \boldsymbol{x}_{k+1}, \ldots, \boldsymbol{x}_n)$ depend on the weights and biases $\theta_k$, as they all change together during a parameter update. To compute the average (Equation 12), it can be assumed that the parameters of the layer no. $k$ and the outputs of the layers change linearly with respect to each other, as if they move from $\theta_k$ to $\theta'_k$ and from $(\boldsymbol{x}_k, \ldots, \boldsymbol{x}_n)$ to $(\boldsymbol{x}'_k, \ldots, \boldsymbol{x}'_n)$ after an update of the parameters of all layers. Under this assumption, the calculation is formulated as follows: while computing the average, the integral contains a function $f_{\theta,k}(t) = \theta_k + t \cdot (\theta'_k - \theta_k)$ for the variable under integration $t \in [0, 1]$ ($\theta_k$ and $\theta'_k$ denote model parameters before and after an update, respectively). Moreover, the integral involves each layer's output: $\boldsymbol{f}_{\boldsymbol{x},i}(t) = \boldsymbol{x}_i + t \cdot (\boldsymbol{x}'_i - \boldsymbol{x}_i)$. Finally, the average gradient (Equation 12) is equal to:

$$\begin{aligned}\underset{\theta_k}{\mathcal{AVG}}\,\nabla_{\theta_k}\ell &= \underset{f_{\theta,k}}{\mathcal{AVG}}\,\nabla_{f_{\theta,k}}\ell = \underset{t}{\mathcal{AVG}}\,(\frac{\partial \boldsymbol{f}_{\boldsymbol{x},k}(t)}{\partial f_{\theta,k}(t)} \cdot \frac{\partial \boldsymbol{f}_{\boldsymbol{x},k+1}(t)}{\partial \boldsymbol{f}_{\boldsymbol{x},k}(t)} \cdot \ldots \cdot \frac{\partial \boldsymbol{f}_{\boldsymbol{x},n}(t)}{\partial \boldsymbol{f}_{\boldsymbol{x},n-1}(t)} \cdot \nabla_{\boldsymbol{f}_{\boldsymbol{x},n}(t)}\ell(t)) \\ &= \int_0^1 \frac{\partial \boldsymbol{f}_{\boldsymbol{x},k}(t)}{\partial f_{\theta,k}(t)} \cdot \frac{\partial \boldsymbol{f}_{\boldsymbol{x},k+1}(t)}{\partial \boldsymbol{f}_{\boldsymbol{x},k}(t)} \cdot \ldots \cdot \frac{\partial \boldsymbol{f}_{\boldsymbol{x},n}(t)}{\partial \boldsymbol{f}_{\boldsymbol{x},n-1}(t)} \cdot \nabla_{\boldsymbol{f}_{\boldsymbol{x},n}(t)}\ell(t)\; dt\end{aligned} \tag{13}$$

which is more direct and easier to work with.

Importantly, unlike the gradient, the average gradient ($\mathcal{AVG}_{\theta_k}\,\nabla_{\theta_k}\ell$) is directly proportional to the loss-change impact of each model parameter separately $\boldsymbol{l}_{\theta',k} - \boldsymbol{l}_{\theta,k}$ (of the shape of $\theta_k$ and $\theta'_k$, unlike the scalar $\ell$):

$$\begin{aligned}\underset{\theta_k}{\mathcal{AVG}}\,\nabla_{\theta_k}\ell &= \underset{\theta_k}{\mathcal{AVG}}\,\nabla_{\theta_k}(\sum_{j=0}^{n}\sum_{i=0}^{|\theta_j|}\ell_{\theta,j,i}) = \underset{\theta_k}{\mathcal{AVG}}\,\nabla_{\theta_k}(\sum_{i=0}^{|\theta_k|}\ell_{\theta,k,i}) = \underset{\theta_k}{\mathcal{AVG}}\,(\mathrm{diag}(\frac{\partial \boldsymbol{l}_{\theta,k}}{\partial \theta_k})) = \\ &= \langle \underset{\theta_{k,1}}{\mathcal{AVG}}\,\ell'_{\theta,k,1}, \ldots, \underset{\theta_{k,n}}{\mathcal{AVG}}\,\ell'_{\theta,k,n}\rangle = \langle \frac{\int_{\theta_{k,1}}^{\theta'_{k,1}} \ell'_{\vartheta,k,1}\, d\vartheta}{\theta'_{k,1} - \theta_{k,1}}, \ldots, \frac{\int_{\theta_{k,n}}^{\theta'_{k,n}} \ell'_{\vartheta,k,n}\, d\vartheta}{\theta'_{k,n} - \theta_{k,n}}\rangle \\ &= \langle \frac{\ell_{\theta',k,1} - \ell_{\theta,k,1}}{\theta'_{k,1} - \theta_{k,1}}, \ldots, \frac{\ell_{\theta',k,n} - \ell_{\theta,k,n}}{\theta'_{k,n} - \theta_{k,n}}\rangle = (\theta'_k - \theta_k)^{\circ -1} \circ (\boldsymbol{l}_{\theta',k} - \boldsymbol{l}_{\theta,k}) \propto \boldsymbol{l}_{\theta',k} - \boldsymbol{l}_{\theta,k}\end{aligned} \tag{14}$$

where $\circ$ denotes the elementwise operation of either multiplication or inversion ($(\cdot)^{\circ -1}$). $\mathrm{diag}(\frac{\partial \boldsymbol{l}_{\theta,k}}{\partial \theta_k})$ denotes diagonal elements of the Jacobian matrix. $\ell_{\theta,k,i} \in \boldsymbol{l}_{\theta,k}$ represents the scalar loss contribution of a single model parameter ($\theta_{k,i}$), that can be defined as an integral of the gradient: $\ell_{\theta,k,i} = \int_{C_1}^{\theta_{k,i}} \nabla_{\vartheta}\ell_{\theta}\, d\vartheta + C_2$, for any constant scalars $C_1$ and $C_2$. (Note that in this case, $\ell_{\theta,k,i} \neq \ell_{\theta} + C_{\ell}$, for any constant $C_{\ell}$, because the loss $\ell$ also depends on other parameters than $\theta_{k,i}$.) Important properties: (a) $\ell_{\theta} = C + \sum_{k=0}^{n}\sum_{i=0}^{|\theta_k|}\ell_{\theta,k,i}$ for a constant $C$ that is invariant across updates of the model parameters $\theta$. (b) The elements of $\boldsymbol{l}$ are related to the difference in loss during parameter update: $\ell_{\theta'} - \ell_{\theta} = (\sum_{k=0}^{n}\sum_{i=0}^{|\theta'_k|}\ell_{\theta',k,i}) - (\sum_{k=0}^{n}\sum_{i=0}^{|\theta_k|}\ell_{\theta,k,i})$. (c) The following equation is satisfied: $\nabla_{\theta}\ell = \nabla_{\theta}(\sum_{k=0}^{n}\sum_{i=0}^{|\theta_k|}\ell_{\theta,k,i})$. The simple one-dimensional visualization of the proportionality from Equation 14 ($\mathcal{AVG}_{\theta_k}\,\nabla_{\theta_k}\ell \propto \boldsymbol{l}_{\theta',k} - \boldsymbol{l}_{\theta,k}$) is shown in Fig. 1. Note that the property of proportionality does not hold for the gradient updates (which are utilized by Adam (Kingma & Ba, 2014), RMSProp (Tieleman et al., 2012), and SGD (Ketkar, 2017; Liu et al., 2020)). In the gradient case, during the update step of $\theta$ weights, $\theta'$ is not used in the calculation of itself. Therefore, $\boldsymbol{l}_{\theta'} - \boldsymbol{l}_{\theta}$ cannot be computed yet, and the accurate influence on loss remains unknown, unlike for the average gradient (Equation 14). The cases of scalar parameters $\theta_{k,i} \in \theta_k$ and $\theta'_{k,i} \in \theta'_k$ where division by zero occurs are handled differently:

$$\forall i : \theta'_{k,i} - \theta_{k,i} = 0 \implies \underset{\vartheta_{k,i} \in [\theta_{k,i}, \theta'_{k,i}]}{\mathcal{AVG}}\,\frac{\partial \ell}{\partial \vartheta_{k,i}} = \frac{\partial \ell}{\partial \theta_{k,i}} \tag{15}$$

Assuming the functions $f_{\theta,k}$ and $\boldsymbol{f}_{\boldsymbol{x},i}$ from Equation 13 are any functions (but differentiable with respect to each other), Equation 14 remains valid. Therefore, the crucial property of direct proportionality to the loss values does not depend on our previous assumptions about $\theta_k$ and $\boldsymbol{x}_i$. The purpose of these assumptions is to provide a simple example, reduce reasoning abstraction, and simplify further proofs in Sections B.2 and B.3.

### B.2 PROOF OF OF EQUATION 1 WITHOUT SPECIFYING PRECISION OF APPROXIMATION

For some function $f$ and some constants $C_1, C_2, \dots, C_n$:

$$\int C_1 \cdot C_2 \cdot \ldots \cdot C_n \cdot f(x)\, dx \; = C_1 \cdot C_2 \cdot \ldots \cdot C_n \cdot \int f(x)\, dx \tag{16}$$

Similarly, let us denote approximately constant functions as $C_1'(x) \approx C_1, C_2'(x) \approx C_2, \dots, C_n'(x) \approx C_n$ for some $x \in [a, b]$, $a \neq b$. The constant that precisely approximates each function $C'(x)$, is its average: $C_1'(x) \approx \mathcal{AVG}\, C_1'(x) = C_1, C_2'(x) \approx \mathcal{AVG}\, C_2'(x) = C_2, \dots, C_n'(x) \approx \mathcal{AVG}\, C_n'(x) = C_n$. Therefore, similarly to Equation 16:

$$\begin{aligned}\int_a^b C_1'(x) \cdot \ldots \cdot C_n'(x) \cdot f(x)\, dx \;&\approx\; \underset{x \in [a,b]}{\mathcal{AVG}}\, C_1'(x) \cdot \ldots \cdot \underset{x \in [a,b]}{\mathcal{AVG}}\, C_n'(x) \cdot \int_a^b f(x)\, dx \\ \int_a^b C_1'(x) \cdot \ldots \cdot C_n'(x) \cdot f(x)\, dx \;&\approx\; \int_a^b \frac{C_1'(x)}{b-a} dx \cdot \ldots \cdot \int_a^b \frac{C_n'(x)}{b-a} dx \cdot \int_a^b f(x)\, dx\end{aligned} \tag{17}$$

which is also approximately equal to both sides of Equation 16. In Equation 17, both approximations are equivalent, because $\mathcal{AVG}\, C_i'(x) = \int_a^b C_i'(x)/(b-a)\; dx$. For functions $\mathbb{R}^n \to \mathbb{R}^m$, equations no. 16 and 17 are analogous. Note that, in the general case, the different approximations of the terms $C_i'(x) \approx C_i'(a)$ and $C_i'(x) \approx C_i(b)$ are worse than the average: $C_i'(x) \approx \mathcal{AVG}\, C_i'(x) = C_i$ (which is used further in Section B.3).

Rapid changes in the gradient over the range of an update indicate that the update step is too large, leading to instability and reduced training effectiveness due to excessively large steps in the loss landscape. We assume effective learning, where gradients do not change significantly[1] between updates, ensuring the learning rate is appropriately sized. In this case, the gradient $\nabla_{\theta_k} \ell$ does not change significantly[2] over the range of a weight update $[\theta, \theta']$. Therefore, using Equation 17 to approximate Equation 13:

$$\begin{aligned}\underset{\theta_k}{\mathcal{AVG}}\, \nabla_{\theta_k} \ell &= \int_0^1 \frac{\partial \boldsymbol{f}_{\boldsymbol{x},k}(t)}{\partial f_{\theta,k}(t)} \cdot \frac{\partial \boldsymbol{f}_{\boldsymbol{x},k+1}(t)}{\partial \boldsymbol{f}_{\boldsymbol{x},k}(t)} \cdot \ldots \cdot \frac{\partial \boldsymbol{f}_{\boldsymbol{x},n}(t)}{\partial \boldsymbol{f}_{\boldsymbol{x},n-1}(t)} \cdot \nabla_{\boldsymbol{f}_{\boldsymbol{x},n}(t)} \ell(t)\; dt \\ \approx \int_0^1 \frac{\partial \boldsymbol{f}_{\boldsymbol{x},k}(t)}{\partial f_{\theta,k}(t)}\; dt &\cdot \int_0^1 \frac{\partial \boldsymbol{f}_{\boldsymbol{x},k+1}(t)}{\partial \boldsymbol{f}_{\boldsymbol{x},k}(t)}\; dt \cdot \ldots \cdot \int_0^1 \frac{\partial \boldsymbol{f}_{\boldsymbol{x},n}(t)}{\partial \boldsymbol{f}_{\boldsymbol{x},n-1}(t)}\; dt \cdot \int_0^1 \nabla_{\boldsymbol{f}_{\boldsymbol{x},n}(t)} \ell(t)\; dt \\ &= \underset{\theta_k}{\mathcal{AVG}}\, \frac{\partial \boldsymbol{x}_k}{\partial \theta_k} \cdot \underset{\boldsymbol{x}_k}{\mathcal{AVG}}\, \frac{\partial \boldsymbol{x}_{k+1}}{\partial \boldsymbol{x}_k} \cdot \ldots \cdot \underset{\boldsymbol{x}_{n-1}}{\mathcal{AVG}}\, \frac{\partial \boldsymbol{x}_n}{\partial \boldsymbol{x}_{n-1}} \cdot \underset{\boldsymbol{x}_n}{\mathcal{AVG}}\, \nabla_{\boldsymbol{x}_n} \ell\end{aligned} \tag{18}$$

Applying the notation of Equation 12 to Equation 18, we get:

$$\begin{aligned}\underset{\theta_k}{\mathcal{AVG}}\, \nabla_{\theta_k} \ell &\approx \int_0^1 \frac{\partial \boldsymbol{f}_{\boldsymbol{x},k}(t)}{\partial f_{\theta,k}(t)}\; dt \int_0^1 \frac{\partial \boldsymbol{f}_{\boldsymbol{x},k+1}(t)}{\partial \boldsymbol{f}_{\boldsymbol{x},k}(t)}\; dt \cdot \ldots \cdot \int_0^1 \frac{\partial \boldsymbol{f}_{\boldsymbol{x},n}(t)}{\partial \boldsymbol{f}_{\boldsymbol{x},n-1}(t)}\; dt \int_0^1 \nabla_{\boldsymbol{f}_{\boldsymbol{x},n}(t)} \ell(t)\; dt \\ &= \int_{\theta_k}^{\theta_k'} \frac{\partial \boldsymbol{x}_k(\vartheta_k)}{\partial \vartheta_k}\; d\vartheta_k \int_{\boldsymbol{x}_k'}^{\boldsymbol{x}_k'} \frac{\partial \boldsymbol{x}_{k+1}(\boldsymbol{\chi}_k)}{\partial \boldsymbol{\chi}_k}\; d\boldsymbol{\chi}_k \cdot \ldots \cdot \int_{\boldsymbol{x}_{n-1}}^{\boldsymbol{x}_{n-1}'} \frac{\partial \boldsymbol{x}_n(\boldsymbol{\chi}_{n-1})}{\partial \boldsymbol{\chi}_{n-1}}\; d\boldsymbol{\chi}_{n-1} \int_{\boldsymbol{x}_n}^{\boldsymbol{x}_n'} \nabla_{\boldsymbol{\chi}_n} \ell\; d\boldsymbol{\chi}_n \\ &= \underset{\theta_k}{\mathcal{AVG}}\, \frac{\partial \boldsymbol{x}_k}{\partial \theta_k} \cdot \underset{\boldsymbol{x}_k}{\mathcal{AVG}}\, \frac{\partial \boldsymbol{x}_{k+1}}{\partial \boldsymbol{x}_k} \cdot \ldots \cdot \underset{\boldsymbol{x}_{n-1}}{\mathcal{AVG}}\, \frac{\partial \boldsymbol{x}_n}{\partial \boldsymbol{x}_{n-1}} \cdot \underset{\boldsymbol{x}_n}{\mathcal{AVG}}\, \nabla_{\boldsymbol{x}_n} \ell\end{aligned} \tag{19}$$

where $\theta_k, \boldsymbol{x}_k, \boldsymbol{x}_{k+1}, \dots, \boldsymbol{x}_n$ are all linear functions of $t$ (previously denoted as $f_{\theta,k}, f_{\boldsymbol{x},k}, f_{\boldsymbol{x},k+1}, \dots, f_{\boldsymbol{x},n}$). Therefore, the functions $\boldsymbol{x}_k(\theta_k), \boldsymbol{x}_{k+1}(\boldsymbol{x}_k), \dots, \boldsymbol{x}_n(\boldsymbol{x}_{n-1})$ are known. The edge cases of those scalars within $\theta_k, \boldsymbol{x}_k, \boldsymbol{x}_{k+1}, \dots, \boldsymbol{x}_n$ that do not depend on $t$ are handled analogously to Equation 15, as in these cases the average gradient equals the gradient.

Despite the provided arguments on why the approximation is applied, the precision of the estimation is not specified, although it *is crucial*. Therefore, the accuracy of the approximation is described in Section B.3. Otherwise, if the precision of the estimation is not important, then Equation 19 ultimately proves Equation 1. □

---

[1] The magnitude of the gradient change need not be specified, as it suffices that it contributes to the approximations with unspecified bounds in Equations 18 and 19. However, the accuracy of these approximations is proven in Section B.3.

[2] See footnote 1.

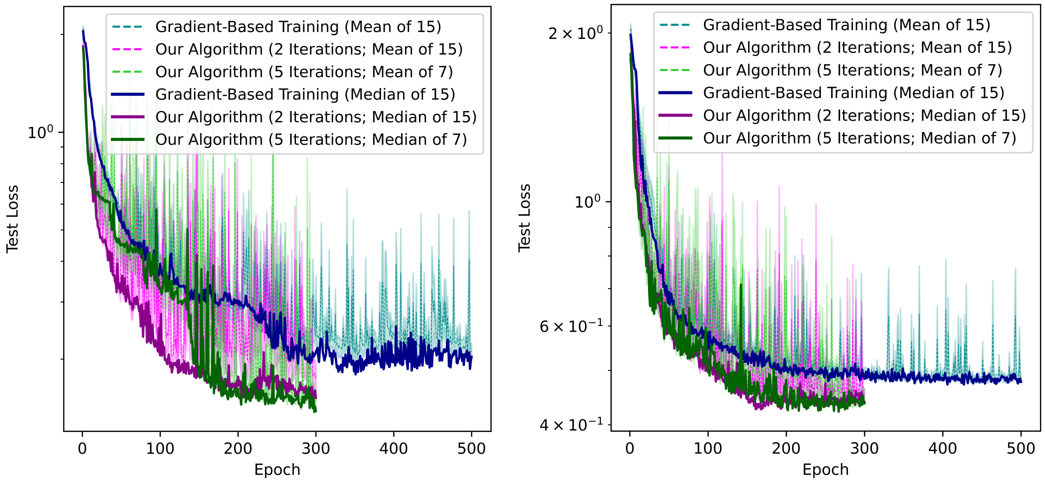
The analogous reasoning can be applied to prove Equation 5.

In the algorithm, it is also assumed that the average gradient of the loss with respect to the output of the last layer, denoted as $(\mathcal{AVG}_{\boldsymbol{x}_n} \nabla_{\boldsymbol{x}_n} \ell)$, is replaced by the gradient $(\nabla_{\boldsymbol{x}_n} \ell)$. Moreover, in our implementation, the gradients replace the average gradients of layers that are approximately linear (using Equation 6 instead of Equation 5), resulting in a broader approximation in Equation 1. However, the presented reasoning still applies, including the proof of approximation accuracy in Section B.3. See Appendix F for comments on the limitations of our implementation of Equation 1.

### B.3 PROOF OF SUFFICIENT PRECISION OF APPROXIMATION

Referring to the content of the paragraphs just before and after Equation 17, the approximation in Equation 17 is more precise in the case of $C_i'(x) \approx \mathcal{AVG}\, C_i'(x) = C_i$ than in the case of approximating $C_i'(x) \approx C_i'(a)$. The average Jacobian of each term in Equation 1 can be denoted as $\mathcal{AVG}\, C_i'(x)$, while the Jacobian of each term in Equation 2 can be denoted as $C_i'(a)$. For the average Jacobian $\mathcal{AVG}\, C_i'(x)$, a better estimation in Equation 17 is obtained, as stated in the text near the equation. Consequently, applying Equation 17 to approximate Equation 13 results in a higher precision in estimating Equation 1 when averaging each Jacobian term separately, compared to utilizing the Jacobians without averaging. Therefore, a better approximation of the accurate average gradient is obtained compared to using the gradient. ☐ The average gradient is proportional to the change in loss after the corresponding parameter update (Equation 14). Therefore, approximating the average gradient more precisely than current gradient-based methods can lead to more efficient minimization of batch loss, for example, by using Eq. 1. Therefore, learning can be enhanced compared to the potential of gradient-based methods.

## C TEST LOSS CURVES OF MODEL B

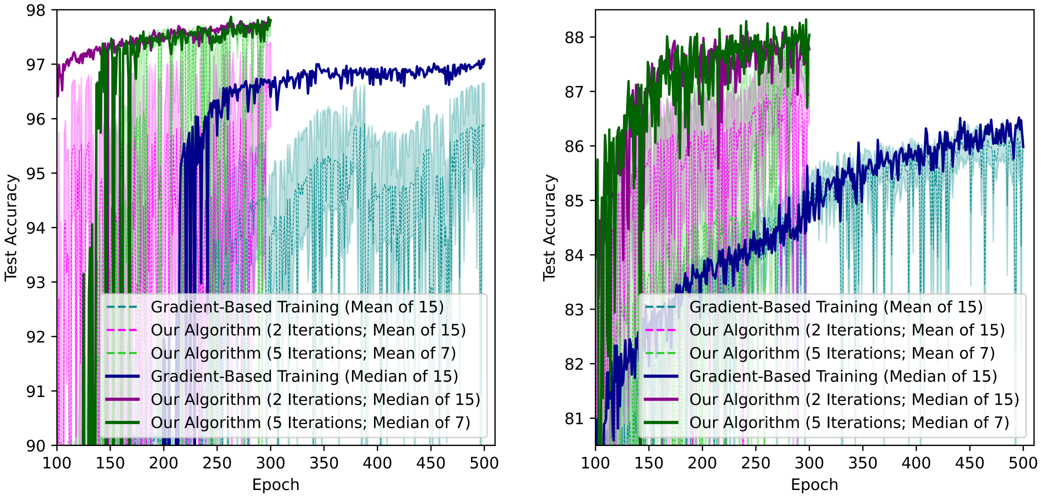


(a) *MNIST*

(b) *Fashion MNIST*

Figure 3: *Test losses of Model B*. Only mean curves contain confidence ranges (SEM).

## D TEST ACCURACY CURVES OF MODEL B



(a) *MNIST* (b) *Fashion MNIST*

Figure 4: *Test accuracy of Model B*. Only mean curves contain confidence ranges (SEM).

## E SCALING IN TERMS OF MODEL DEPTH

The algorithm based on the average gradient aims to reduce errors of the predicted influence on loss of a parameter update. In the case of the gradient-based approach, the errors arise from the impaired prediction of how inputs to subsequent layers influence their outputs (Fig. 1). Let us model the errors as multiplicative, because each time a fraction of output may be influenced by the error. Therefore, when compared to the gradient-based algorithm as a baseline, the multiplicative errors are reduced after backpropagation through each nonlinear layer (by computing the average Jacobian of the layer). Consequently, the incorporation of the average gradient exponentially reduces the error in terms of a count of nonlinear layers (that are involved in the backpropagation process). This explains the huge performance-improvement gap between the models for the method based on the average gradient, which emerges from the difference in models' depths. However, the gap is also increased due to the linearity of the ELU activation function in most of its domain, where the gradient equals its average. In this case, our algorithm produces results similar to those of gradient-based optimization.

If the errors (of the predicted influence on loss of a parameter update) are enormous, then the learning is impossible. Therefore, the learning performance tends to decrease after the error reaches a certain value for a given model, learning rate, and other parameters. From that point onward, our algorithm more efficiently reduces the batch loss compared to the gradient-based approach by minimizing the error in the loss-influence prediction. Importantly, the improvements tend to increase with both the number of nonlinear layers in a model and the learning rate.

## F THREE-DIMENSIONAL COMPARISON OF THE GRADIENT AND THE AVERAGE GRADIENT

In our experiments, during a parameter update, in terms of the average reduction of loss for a batch, our algorithm lies between the gradient (red arrows in Figure 5) and the lowest average gradient (black arrows in Figure 5). Our algorithm does not always find a locally optimal solution (the best in the range of a single parameter update) because:

a The average gradient is approximated (by using Equation 1 instead of Equation 3, Equation 6 as a substitute of Equation 5, and the non-averaged gradient of the loss with respect to the last layer output).

b The optimal parameter update may be inaccurately estimated before the average gradient for this parameter update is calculated. Moreover, even after many iterations of Algorithm 4

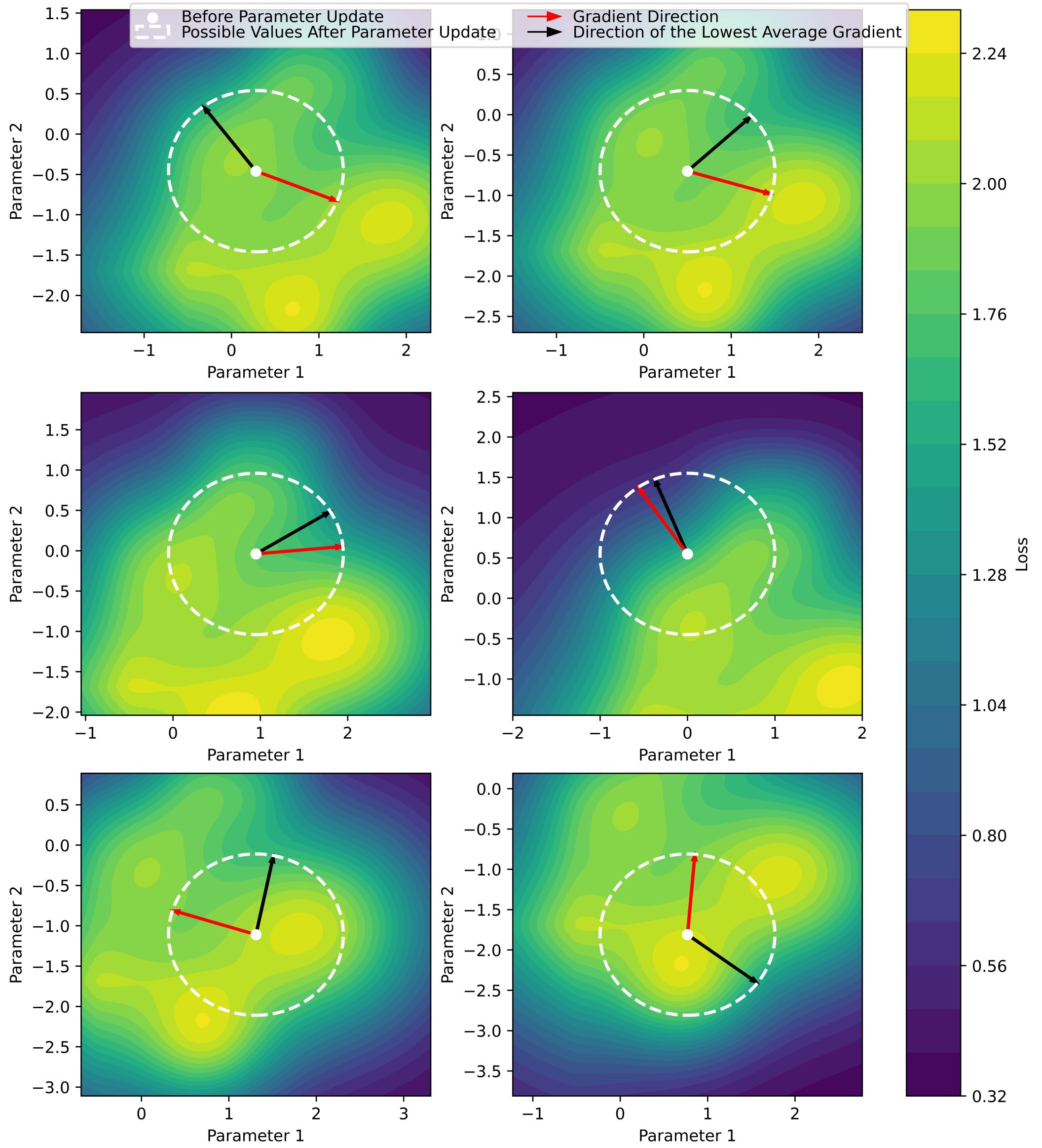


Figure 5: *Three-dimensional comparison of the gradient and the lowest average gradient in a few example scenarios.* The latter accurately reflects the influence on the loss of a parameter update. Furthermore, it accurately shows how each model parameter individually contributes to the change in the batch loss (Equation 14), which is utilized by our algorithm. Each plot illustrates the loss in terms of two example model parameters, assuming a specific magnitude for each parameter update (represented by the radius of each white circle). The arrows point to the loss values after an update based on the gradient and the average gradient. The average gradient is calculated for the update that minimizes it. Therefore, it points to the minimum loss on each white circle, although this minimum is not always achieved by the approximated average gradient computed by our algorithm.

(Appendix G), the update step may not converge to a locally optimal solution (black vectors in Figure 5).

c After the first iteration of our algorithm, only the negations of the directions of changes in each parameter are possible. Thus, the search for locally optimal updates is bounded by $2^{|\Theta|}$ combinations, where $|\Theta|$ is the count of trainable parameters.

Nevertheless, the $\mathcal{RD}$ metric (defined in Equation 7) indicates our algorithm minimizes the batch loss more efficiently on average compared to the gradient-based approach.

## G ALGORITHM VERSION WITH PARAMETERIZED NUMBER OF ITERATIONS

**Algorithm 4** *Algorithm Version with Parameterized Number of Iterations* (two or more). The number of iterations is equal to the optimal number of backpropagation calls and inferences. The memory requirement of the ideal implementation would be higher than that of Adam by only an additional scalar size per parameter of the model.

```
Input: $model$: Neural Network Model to Train
       $dataset$: Training Dataset
       $lossFn$: Loss Function
       $optimizer$: Optimizer
       $iterCount$: Number of Backpropagation Iterations
for all $batch \in dataset$ do
    $modelInitial \leftarrow model$
    $modelCopy \leftarrow model$
    $initialOutput \leftarrow modelCopy(batch.\mathrm{x})$ {It is assumed that $modelCopy$'s layers' results are kept inside $modelCopy$}
    $initialLoss \leftarrow LossFn(initialOutput, batch.\mathrm{y})$
    $Backpropagate(initialLoss)$ {Compute the gradients using the standard backpropagation procedure. Assume that the gradients are stored inside $modelCopy$}
    $optimizer.\mathrm{Step}(modelCopy)$ {Parameter update}
    $modelOutputAfterUpdate \leftarrow modelCopy(batch.\mathrm{x})$
    $modelLossAfterUpdate \leftarrow LossFn(modelOutputAfterUpdate, batch.\mathrm{y})$
    for $iter = 1, ..., iterCount - 1$ do {Loop $(iterCount - 1)$ times, because one backward propagation is done}
        if $iter \neq 1$ then
            $modelCopy \leftarrow model$
            $modelCopy(batch.\mathrm{x})$ {For each layer, compute its output, and store it inside $modelCopy$}
            $model \leftarrow modelInitial$
        end if
        $initialOutput \leftarrow model(batch.\mathrm{x})$ {This computation is redundant if layer outputs are copied from $modelInitial$}
        $initialLoss \leftarrow LossFn(initialOutput, batch.\mathrm{y})$ {Analogously, this computation is also redundant}
        $\mathbf{AveragedBackpropagation}(model, modelCopy, initialLoss)$ {The procedure is described as Algorithm 2. The parameters of the $model$ are modified within}
    end for
end for
```