Adapting Mono-Forward with Zeroth-Order Gradient Estimation for Automatic Differentiation-Free Training

Ateş Görpelioğlu

Supervisor(s): Stephanie Tan, Yaqi Guo

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

June 20, 2026

Name of the student: Ateş Görpelioğlu
Final project course: CSE3000 Research Project
Thesis committee: Stephanie Tan, Yaqi Guo, Inald Lagendijk

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Adapting Mono-Forward with Zeroth-Order Gradient Estimation for Automatic Differentiation-Free Training

Ateş Görpelioğlu
Delft University of Technology
a.gorpelioglu@student.tudelft.nl

Abstract

The aim of this paper is to explore the potential of adapting the Mono-Forward algorithm with Zeroth-Order Optimization for backpropagation (BP) and automatic-differentiation(AD)-free image classification, assessing its feasibility in scenarios where exact gradients are unavailable. The Mono-Forward method introduces a novel approach to training neural networks without the need for backpropagation or multiple forward passes typically required in forward-forward algorithms; however it still relies on AD for local training of model layers when implemented with modern deep learning frameworks. This work proposes *MF+DD*, which replaces AD in Mono-Forward with zeroth-order gradient estimation via directional derivatives, resulting in a training algorithm that is free of AD and global BP. This paper also introduces a random projection based modification to adress the limitation of Mono-Forward in architectures with large intermediate activation tensors, for increased computational efficiency. Experiments on MNIST, FashionMNIST, CIFAR-10, and CIFAR-100 with both MLP and CNN architectures show that MF+DD achieves comparable accuracy to MF with AD on simpler datasets, while the accuracy gap widens on more complex benchmarks, suggesting that the noise introduced by the directional derivative estimator becomes more impactful as task difficulty increases. Results further show that increasing the number of perturbation directions P improves both accuracy and training stability with a downside of increased computational cost.

1 Introduction

Deep learning (DL) has achieved remarkable success across a wide range of domains, largely driven by backpropagation (BP) [1]. BP can be considered as a specific implementation of the reverse-mode automatic differentiation algorithm applied to the computational graph of a neural network [2], which is also the main method for efficient gradient-based optimization in modern deep learning libraries such as PyTorch [3].

Despite its effectiveness and popularity in the deep learning field, BP presents several fundamental limitations. Due to its sequential nature, BP relies on a strict layer-wise dependency between forward and backward passes, as during the forward pass each layer's input relies on the output of the previous layer, and during the backward pass each layer must wait for error propagation from the succeeding layer. This leads to forward and backward locking, which limits parallelism and training throughput. From a memory perspective, BP requires storing intermediate activations during the forward pass, resulting in high memory usage [4]. Bartunov et al. [5] also note that relying on symmetric weights to propagate gradients is biologically implausible because "it

implies a mode of information propagation (error propagation) that does not influence neural activity, and that depends on an implausible network architecture (symmetric weight connectivity for feedforward and feedback directions, which is called the weight transport problem)."

Additionally, the implementation of automatic differentiation (AD) is a challenge in physical neural networks (PNNs). Backpropagation relies on the transpose of the weight matrix to propagate error signals backward through the layers. In digital systems, computing the matrix transpose is a simple software operation involving memory indexing. However, in PNNs, weights are embodied by physical hardware components, so reading and transposing these matrices are not inherently supported [6]. These constraints motivate the development of training paradigms that eliminate reliance on both backpropagation and automatic differentiation.

Alternative learning strategies have been proposed to address these limitations, including Direct Feedback Alignment (DFA) [7], equilibrium propagation [8], and the Forward-Forward (FF) algorithm [9]. Each method has its own weaknesses: DFA and FF lag behind BP in performance, and equilibrium propagation requires symmetric weights [7, 8, 10]. The algorithm that will be explored in more detail in this paper, Mono-Forward (MF) [10], improves on Forward-Forward by eliminating the need for negative samples and multiple forward passes for multi-class classification tasks, training each layer with a single forward pass and a locally-computed cross-entropy loss. In practice, MF still relies on automatic differentiation to compute layer-wise gradients when implemented in modern DL frameworks such as PyTorch, leaving it incompatible with hardware where AD is unavailable.

One approach to removing this remaining dependency is zeroth-order (ZO) optimization, which estimates gradients using only function evaluations [11]. A particular variant approximates the gradient via directional derivatives: parameters are perturbed in a random direction, and the resulting change in loss is used to construct a gradient estimate. A practical implementation, FFZero [12], demonstrated that this approach is viable in a local learning setting, showing that directional-derivative errors remain confined to the layer being trained and do not compound across the network, which is a property that makes ZO estimation particularly well-suited forward-only, layer-wise training paradigms like MF.

This work aims to adapt the MF algorithm to use ZO gradient estimation via directional derivatives. The resulting method, referred to as ***MF+DD*** onwards, is fully free of both global backpropagation and automatic differentiation: each layer can be trained independently using only two forward evaluations of its local loss function per parameter group and perturbation direction, while more perturbation directions can be used to reduce the noise of the gradient estimate. In addition to MF+DD, this work also introduces a random projection based modification to address the limitation of MF in architectures with large intermediate activations, which can lead to prohibitively large projection matrices, causing computational inefficiency. In the experiments, MF+DD is evaluated on standard image classification benchmarks then compared against BP and the original MF algorithm with AD.

This paper investigates two research questions. **RQ1:** What is the accuracy cost of replacing AD with DD based zeroth-order methods in the Mono-Forward framework, and **RQ2:** How does the number of perturbation directions P affect classification performance and model convergence?

2 Background and Related Work

This section gives an overview of the most relevant methods, including the Forward-Forward, Mono-Forward, and FFZero algorithms, as well as the theory behind zeroth-order optimization via directional derivatives.

2.1 Backpropagation-Free Local Learning

Forward-Forward algorithm. Hinton [9] proposed the Forward-Forward (FF) algorithm as a replacement for backpropagation in which the backward pass is eliminated entirely. Instead of propagating a global error

signal, FF trains each layer independently by contrasting two forward passes: one on real (positive) data and one on synthetically constructed negative data. Each layer maximizes a scalar "goodness" measure, such as the sum of squared neural activities, on positive data, and minimizes it on negative data.

Mono-Forward algorithm. The Mono-Forward (MF) algorithm [10] improves upon Forward-Forward by eliminating the need for negative samples and the associated second forward pass. Each layer maintains a projection matrix that maps activations to class-specific goodness scores, and is trained with a local cross-entropy loss. The authors argue that the performance gap between FF and backpropagation is primarily due to the inefficiency of the contrastive goodness objective used in FF. By replacing this with a multi-class classification signal, MF achieves superior accuracy while requiring only a single forward pass per training step.

In the MF algorithm, each layer l of a neural network contains a projection matrix $\boldsymbol{M}_l \in \mathbb{R}^{n \times m}$, where m is the number of output classes and n is the number of neurons in layer l. The goodness score for each layer is defined as:

$$\boldsymbol{G}_l \triangleq \boldsymbol{a}_l^\top \times \boldsymbol{M}_l \tag{1}$$

Where $\boldsymbol{a}_l$ denotes the activations of layer l and $\boldsymbol{M}_l$ is the projection matrix. This matrix can also be viewed as a layer-wise linear classifier. Each entry in $\boldsymbol{G}_l$ corresponds to the score for a specific class, which essentially acts as a intermediate prediction for that layer.

The loss function used in training for each layer is defined as:

$$\mathcal{L}_l \triangleq -\sum_{c=1}^{m} y_c \log(\sigma(\boldsymbol{G}_l)_c) \tag{2}$$

Where y_c is the one-hot encoded label for class c, and σ is the softmax function. For a batch of B examples, this per-sample loss extends to the batched form used in practice:

$$\mathcal{L}_l = -\frac{1}{B} \sum_{b=1}^{B} \sum_{c=1}^{m} y_{bc} \log(\sigma(\boldsymbol{G}_l)_{bc}) \tag{3}$$

Where y_{bc} is a binary indicator of whether example b belongs to class c, and B is the batch size. The weights of both the main layer and the projection matrix are updated using gradient descent as follows:

$$\boldsymbol{W}_l \leftarrow \boldsymbol{W}_l - \eta \left(\frac{\partial \mathcal{L}_l}{\partial \boldsymbol{G}_l} \cdot \frac{\partial \boldsymbol{G}_l}{\partial \boldsymbol{a}_l} \cdot \frac{\partial \boldsymbol{a}_l}{\partial \mathbf{z}_l} \cdot \frac{\partial \mathbf{z}_l}{\partial \boldsymbol{W}_l} \right) \tag{4}$$

$$\boldsymbol{M}_l \leftarrow \boldsymbol{M}_l - \eta \left(\frac{\partial \mathcal{L}_l}{\partial \boldsymbol{G}_l} \cdot \frac{\partial \boldsymbol{G}_l}{\partial \boldsymbol{M}_l} \right) \tag{5}$$

Where η is the learning rate. The key point is that the loss $\mathcal{L}_l$ depends only on the parameters of layer l, so the gradients can be computed using only local information.

2.2 Zeroth-Order Optimization

In modern deep learning frameworks, the computation of gradients for the aforementioned local loss functions still relies on automatic differentiation. In PyTorch, for example, the `torch.autograd` module [13]

constructs a computational graph during the forward pass, and then traverses this graph in reverse to compute gradients during the backward pass. This means that even when using a local loss function that depends only on the parameters of a single layer, the gradients are still computed using AD, which is not compatible with hardware where AD may be implausible.

Gradient Estimation via Directional Derivatives ZO optimization methods estimate gradients using only function evaluations [11]. These methods are particularly useful in scenarios where gradient information is unavailable or expensive to compute, such as in black-box optimization problems or when training physical neural networks.

Let $\boldsymbol{\omega}$ denote the trainable parameters of a layer. The standard approach for optimizing a layer's parameters $\boldsymbol{\omega}$ using gradient descent is defined as:

$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \eta \frac{\partial \mathcal{L}(\boldsymbol{\omega})}{\partial \boldsymbol{\omega}} \tag{6}$$

To eliminate the dependence on automatic differentiation the exact gradient $\frac{\partial \mathcal{L}(\boldsymbol{\omega})}{\partial \boldsymbol{\omega}}$ is replaced by a DD approximation. Given a random perturbation vector $\mathbf{v} \sim \mathcal{N}(0, I)$, let $\hat{\mathbf{v}}$ denote the normalized perturbation vector, defined as $\hat{\mathbf{v}} = \frac{\mathbf{v}}{||\mathbf{v}||}$. The directional derivative of $\mathcal{L}$ in the direction of $\hat{\mathbf{v}}$ is defined as:

$$\nabla_{\mathbf{v}} \mathcal{L}(\boldsymbol{\omega}) = \lim_{\varepsilon \to 0} \frac{\mathcal{L}(\boldsymbol{\omega} + \varepsilon \hat{\mathbf{v}}) - \mathcal{L}(\boldsymbol{\omega} - \varepsilon \hat{\mathbf{v}})}{2\varepsilon}. \tag{7}$$

Where $\varepsilon \ll 1$ is a small perturbation magnitude. Here $\hat{\mathbf{v}}$ simultaneously perturbs all parameters in $\boldsymbol{\omega}$, and the normal distribution ensures that the perturbation is also uniformly distributed across all dimensions of the parameter space. Although this scalar quantity captures only the rate of change in a single direction, scaling it by $n\hat{\mathbf{v}}$, where $n = \dim(\boldsymbol{\omega})$ is the total number of trainable parameters in $\boldsymbol{\omega}$, produces an unbiased estimator of the full gradient vector. As shown in FFZero [12]:

$$\mathbb{E}_{\mathbf{v}}[n \nabla_{\mathbf{v}} \mathcal{L}(\boldsymbol{\omega}) \hat{\mathbf{v}}] = \frac{\partial \mathcal{L}(\boldsymbol{\omega})}{\partial \boldsymbol{\omega}} \tag{8}$$

Substituting this estimator into a standard gradient-descent step gives the parameter update:

$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \eta n \nabla_{\mathbf{v}} \mathcal{L}(\boldsymbol{\omega}) \hat{\mathbf{v}} \tag{9}$$

While any single directional update is a noisy estimate of the true gradient, the optimization trajectory converges in expectation to the same solution as exact gradient descent. To improve accuracy and accelerate convergence, the directional derivative can be averaged over $P > 1$ independent random perturbations $\{\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_P\}$ as follows:

$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} - \frac{\eta n}{P} \sum_{p=1}^{P} \nabla_{\mathbf{v}_p} \mathcal{L}(\boldsymbol{\omega}) \hat{\mathbf{v}}_p \tag{10}$$

Practical limitations of directional derivatives. A key insight is that directional derivatives are unstable when used to optimize a global loss across the full network, because the approximation error affects the entire chain of layers and can accumulate as depth increases. In contrast, when the same perturbative update is applied to local layer-wise objectives, the error is confined to the layer currently being trained, preventing it from compounding across the network [12].

While averaging over P directions reduces the noise of the estimate, a practical concern remains when applying this update to large layers. The scaling factor $n = \dim(\boldsymbol{\omega})$ can be extremely large: a single linear layer connecting 2000 activations to 2000 neurons has $n \approx 4{,}000{,}000$ parameters, meaning the noisy scalar estimate $\nabla_{\mathbf{v}}\mathcal{L}(\boldsymbol{\omega})$ is multiplied by a large value before being applied as an update. Therefore, even modest variance in the directional derivative estimate is amplified to an extent that can destabilize training entirely. This motivates partitioning the parameter vector $\boldsymbol{\omega}$ into smaller chunks and applying independent directional perturbations to each, a strategy discussed further in Section 4.4.

FFZero. An application of directional-derivative-based ZO optimization to local learning is the FFZero [12] framework, which combines Forward-Forward-style local learning with zeroth-order gradient estimation. Rather than using AD, FFZero estimates layer-wise gradients using directional derivatives: for a random unit perturbation $\hat{\mathbf{v}}$, the directional derivative is approximated via a centered finite difference, and the resulting scalar is used to construct a gradient estimate proportional to $\hat{\mathbf{v}}$.

The key observation is that directional-derivative approximation errors remain confined to the layer being trained in a local learning setting, preventing them from compounding across the network. FFZero demonstrates that fully AD-free training is feasible for image classification, though it relies on the an objective with prototype-based goodness scores rather than the more straightforward cross-entropy formulation of Mono-Forward.

In this paper, some implementation details are inspired by the FFZero framework, which are explained in more detail in Section 4.4.

3 Mono-Forward with Directional Derivatives

Let $\boldsymbol{W}_l$ and $\boldsymbol{M}_l$ denote the trainable parameters of layer l: the weight matrix and the projection matrix respectively. Recall Equation (2) , which defines the local loss for layer l in the Mono-Forward algorithm. The loss $\mathcal{L}_l$ is implicitly composed of both $\boldsymbol{W}_l$ and $\boldsymbol{M}_l$ through this chain of dependencies. Rather than differentiating through this chain using automatic differentiation, the directional derivative approach treats $\mathcal{L}_l$ as a black-box function and estimates its sensitivity to each parameter group purely from forward evaluations.

For the weight matrix $\boldsymbol{W}_l$, the directional derivative is estimated by perturbing $\boldsymbol{W}_l$ in a random direction $\mathbf{v} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ in the parameter space, running a forward pass through layer l to obtain the perturbed activations, computing the perturbed goodness scores via $\boldsymbol{M}_l$, and evaluating the resulting loss. The positive and negative perturbed losses are computed as follows:

$$\mathcal{L}_l^{+} = -\sum_{c=1}^{m} y_c \log\Big(\sigma\big((\boldsymbol{a}_l^{+})^\top \times \boldsymbol{M}_l\big)_c\Big), \quad \text{where } \boldsymbol{a}_l^{+} \text{ is computed using } \boldsymbol{W}_l + \varepsilon\,\hat{\mathbf{v}} \tag{11}$$

$$\mathcal{L}_l^{-} = -\sum_{c=1}^{m} y_c \log\Big(\sigma\big((\boldsymbol{a}_l^{-})^\top \times \boldsymbol{M}_l\big)_c\Big), \quad \text{where } \boldsymbol{a}_l^{-} \text{ is computed using } \boldsymbol{W}_l - \varepsilon\,\hat{\mathbf{v}} \tag{12}$$

Where $\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$ is the normalized perturbation direction. The directional derivative of $\mathcal{L}_l$ with respect to $\boldsymbol{W}_l$ is then estimated as:

$$\nabla_{\mathbf{v}}\mathcal{L}_l(\boldsymbol{W}_l) \approx \frac{\mathcal{L}_l^{+} - \mathcal{L}_l^{-}}{2\varepsilon} \tag{13}$$

and the weight update follows:

$$\boldsymbol{W}_l \leftarrow \boldsymbol{W}_l - \eta\, n_{\boldsymbol{W}}\, \nabla_{\mathbf{v}} \mathcal{L}_l(\boldsymbol{W}_l) \cdot \hat{\mathbf{v}} \tag{14}$$

An analogous procedure applies to $\boldsymbol{M}_l$. Here, the perturbation is applied directly to the projection matrix, while $\boldsymbol{W}_l$ and the activations $\boldsymbol{a}_l$ are held fixed. The perturbed goodness scores are computed using the perturbed projection matrix, and the resulting loss values are:

$$\mathcal{L}_l^+ = -\sum_{c=1}^{m} y_c \log\big(\sigma\big(\boldsymbol{a}_l^\top \times (\boldsymbol{M}_l + \varepsilon\, \hat{\mathbf{u}})\big)_c\big) \tag{15}$$

$$\mathcal{L}_l^- = -\sum_{c=1}^{m} y_c \log\big(\sigma\big(\boldsymbol{a}_l^\top \times (\boldsymbol{M}_l - \varepsilon\, \hat{\mathbf{u}})\big)_c\big) \tag{16}$$

Where $\mathbf{u} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ is an independently sampled perturbation direction and $\hat{\mathbf{u}} = \frac{\mathbf{u}}{\|\mathbf{u}\|}$ is its normalized form. Afterwards, an update rule similar to equation (14) is applied to $\boldsymbol{M}_l$.

In both cases, the entire update requires only two forward passes through layer l. No backward pass or automatic differentiation is needed at any point. Since $\mathcal{L}_l$ depends only on the parameters of layer l, the perturbation of $\boldsymbol{W}_l$ or $\boldsymbol{M}_l$ has no effect on any other layer.

Following the theoretical foundation, the algorithm still requires some practical considerations. As the MF algorithm maintains a projection matrix $\boldsymbol{M}_l$ for each layer, the number of trainable parameters can become extremely large for convolutional neural networks (CNNs) with with large channel counts and spatial dimensions (a channel with dimension $128 \times 32 \times 32$ with 100 output classes has $n = n_{\boldsymbol{W}} + n_{\boldsymbol{M}} \approx 13{,}000{,}000$ trainable parameters in a single layer compared to only $n_{\boldsymbol{W}} \approx 130{,}000$ for the classic CNN architecture). This makes training infeasible for both DD and AD based approaches, with the details discussed in more detail in Section 4.4. The complete algorithm procedure is outlined in Algorithm 1.

4 Methodology

4.1 Network Architectures

The architectures used in the experiments are standard multilayer perceptrons (MLPs) and CNNs. These architectures are widely used in the literature [12, 10, 9] and provide a good testbed for evaluating the performance of the proposed method.

MLP architecture. MLP hidden-layer width and depth are scaled with dataset complexity to ensure sufficient model capacity and prevent underfitting. Specifically, we use 2 hidden layers of width 50 for MNIST, 3 hidden layers of width 100 for FashionMNIST, and 4 hidden layers of width 200 for both CIFAR-10 and CIFAR-100.

The MLP architectures are deliberately kept simple to ensure that the model parameters remain small enough for the directional derivative updates to be stable without requiring excessive computational resources, with details about these decisions explained further in Section 4.4.

CNN architecture. Number of CNN channels in convolutional layers are similarly scaled with dataset complexity: 32 for MNIST, 64 for FashionMNIST, and 128 for both CIFAR datasets. The number of convolutional blocks is fixed at 2 across all datasets to ensure that spatial downsampling behaviour remains consistent. Each convolutional block follows a Convolution, BatchNorm, ReLU, MaxPool structure. Figure 1

illustrates the architecture of a single CNN block. CNN architectures are also modified to keep the trainable parameter count within a reasonable bound, with more details discussed in section 4.4

The complete architectural specifications, including kernel sizes and pooling parameters, are provided in Appendix A.

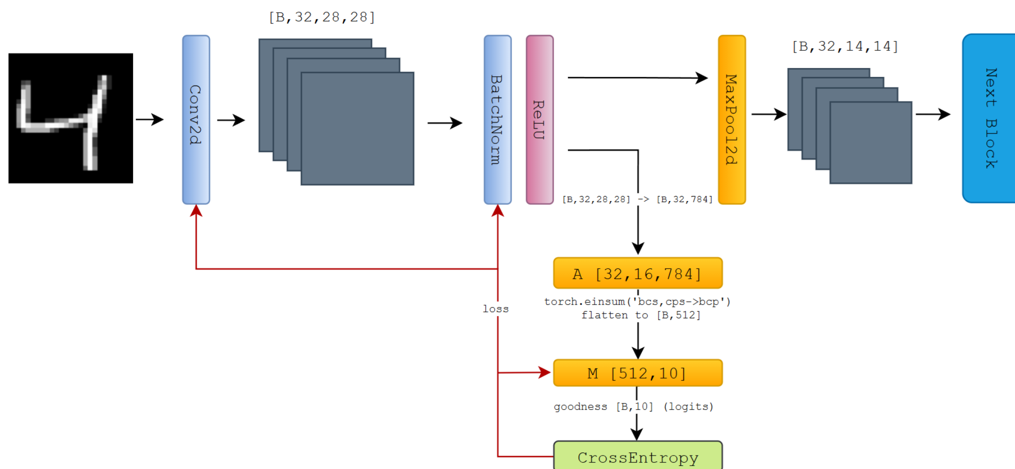

Figure 1: A single CNN block of the proposed architecture, with MNIST data. The spatial feature map is projected to a compact representation via a fixed random projection matrix $\mathbf{A} \in \mathbb{R}^{d \times HW}$ before being mapped to class scores by $\boldsymbol{M}_l$. The random projection matrices are fixed and non-trainable, so that only $\boldsymbol{M}_l$ and the convolutional weights are updated during training.

MF Prediction modes During inference, two prediction modes are evaluated for MF+AD and MF+DD. In ***Cumulative prediction mode***, the goodness score vectors from all layers are summed before the final prediction, making use of the intermediate projection matrices at every layer. In ***Final-layer prediction mode***, only the goodness scores from the final layer are used, similar to a standard classifier head. Both modes are reported to assess how much information the intermediate layers contribute relative to the last.

4.2 Baselines and Training Protocol

Three training conditions are compared under identical architectures: (1) standard backpropagation (BP), (2) the original Mono-Forward algorithm with automatic differentiation (MF+AD), and (3) the proposed Mono-Forward with Directional Derivatives (MF+DD).

1. **Backpropagation (BP):** Standard end-to-end global backpropagation, with cross entropy loss computed at the output layer and gradients propagated backward through the entire network using automatic differentiation. It is expected that BP serves as a strong reference baseline.
2. **Mono-Forward (MF+AD):** Each layer is trained independently using its local cross-entropy loss based on the goodness scores, and gradients are computed using automatic differentiation. Each block is assigned its own `torch.optim.SGD` instance operating only on that block's parameters. During training, the local cross-entropy losses for all blocks are computed in a single forward pass, with activations detached between blocks so that no gradient flows across blocks. For CNNs, uses the random-projection based architecture to reduce trainable parameter count for computational feasibility, which is described in more detail in Section 4.4.

3. **Mono-Forward with Directional Derivatives (MF+DD):** The proposed method, where each layer is trained independently using the same local cross-entropy loss as MF+AD, but gradients are estimated using directional derivatives as described in Section 3. Uses the same architectural modification as MF+AD for CNNs to ensure a fair comparison between the two Mono-Forward variants. Both MF methods are evaluated with both the cumulative and final-layer prediction modes.

4.3 Datasets

The datasets used in the experiments are standard image classification benchmarks of increasing complexity MNIST [14], FashionMNIST [15], CIFAR-10 and CIFAR-100 [16]. These datasets were chosen because they are widely used in literature, representing a range of classification challenges from simple digit recognition to more complex object recognition tasks.

MNIST and FashionMNIST both consist of 28×28 grayscale images with 10 classes, containing 60,000 training and 10,000 test samples each. CIFAR-10 and CIFAR-100 consist of 32×32 RGB images with 10 and 100 classes respectively, each containing 50,000 training and 10,000 test samples. The CIFAR datasets are more complex due to their higher resolution, color channels, and greater number of classes, making them suitable for evaluating the scalability and performance of the proposed method under more challenging conditions.

Before training, all images were normalized using the per-channel mean and standard deviation computed from the training set to stabilize training. No random data augmentations were applied during training to ensure that performance differences between methods are not affected by the differences in data preprocessing.

4.4 Implementation Details and Training Stability

The implementation is done in the PyTorch [3] framework, with the Mono-Forward algorithm adapted to use directional derivative-based gradient estimation as described in Section 3. The entire MF+DD training loop is executed inside a `torch.no_grad()` context, ensuring that no computational graph is constructed at any point. Gradient signals are derived solely from the finite difference approximations described in Section 3.

Parameter chunking. As noted in Section 3, the gradient update in Equation 14 scales linearly with $n_{\boldsymbol{W}} = \dim(\boldsymbol{W}_l)$. For large layers this amplifies the noise in any single directional estimate to the point of destabilizing training. To mitigate this, the weight parameters of each block are partitioned into non-overlapping *chunks*, and an independent perturbation vector is drawn for each chunk. This keeps the effective dimensionality of each perturbation, and therefore the magnitude of the resulting update, bounded, regardless of the total parameter count of the layer.

The update for each chunk is applied immediately before moving to the next, making this a form of sequential block coordinate gradient descent [17] rather than a simultaneous block update. This means each subsequent chunk's gradient estimate is computed against already-updated weights, which further stabilises training by preventing the accumulation of noise across chunks.

For MLP layers, consecutive parameter indices are grouped into chunks of at most $C_{\max}$ parameters. For CNN layers, whole output channels are kept together to avoid splitting the correlated filter weights of a single channel across different perturbation directions: channels are accumulated into a chunk until the combined parameter count (conv weights + bias + BatchNorm scale + shift) would exceed $C_{\max}$, at which point a new chunk is started. The same chunking strategy is applied independently to the rows of the projection matrix $\boldsymbol{M}_l$ for both architecture types.

Algorithm 1 shows the resulting per-batch training procedure with chunking applied to both $\boldsymbol{W}_l$ and $\boldsymbol{M}_l$.

Algorithm 1 MF+DD Layer-Wise Training for One Batch (Chunked)

Require: $\mathbf{X}_{\text{batch}}, \mathbf{y}_{\text{batch}}$; perturbation magnitude ε; learning rate η; directions P; chunk partitions $\{\boldsymbol{W}_l^{(k)}\}$, $\{\boldsymbol{M}_l^{(j)}\}$

1: $\boldsymbol{a}_0 \leftarrow \mathbf{X}_{\text{batch}}$ ▷ Initialise activations with input batch
2: **for** $i \leftarrow 1$, num_layers **do**
3: **for** each chunk $\boldsymbol{W}_i^{(k)}$ of $\boldsymbol{W}_i$ **do**
4: $\mathbf{g}_{\boldsymbol{W}} \leftarrow \mathbf{0}$
5: **for** $p \leftarrow 1, P$ **do**
6: Sample $\mathbf{v} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, $\hat{\mathbf{v}} \leftarrow \mathbf{v}/\|\mathbf{v}\|$
7: $\boldsymbol{W}_i^{(k)} \mathrel{+}= \varepsilon\,\hat{\mathbf{v}}$; $\boldsymbol{a}^+ \leftarrow \mathrm{ReLU}(\boldsymbol{a}_{i-1}\boldsymbol{W}_i)$; $\mathcal{L}^+ \leftarrow \mathrm{CrossEntropy}(\mathbf{y}_{\text{batch}}, \boldsymbol{a}^+\boldsymbol{M}_i)$
8: $\boldsymbol{W}_i^{(k)} \mathrel{-}= 2\varepsilon\,\hat{\mathbf{v}}$; $\boldsymbol{a}^- \leftarrow \mathrm{ReLU}(\boldsymbol{a}_{i-1}\boldsymbol{W}_i)$; $\mathcal{L}^- \leftarrow \mathrm{CrossEntropy}(\mathbf{y}_{\text{batch}}, \boldsymbol{a}^-\boldsymbol{M}_i)$
9: $\boldsymbol{W}_i^{(k)} \mathrel{+}= \varepsilon\,\hat{\mathbf{v}}$ ▷ Restore weights
10: $\mathbf{g}_{\boldsymbol{W}} \mathrel{+}= \dfrac{\mathcal{L}^+ - \mathcal{L}^-}{2\varepsilon} \cdot \hat{\mathbf{v}}$
11: **end for**
12: $\boldsymbol{W}_i^{(k)} \leftarrow \boldsymbol{W}_i^{(k)} - \eta\,\dfrac{n_{\text{chunk}}^{(k)}}{P}\,\mathbf{g}_{\boldsymbol{W}}$ ▷ $n_{\text{chunk}}^{(k)} = \dim(\boldsymbol{W}_i^{(k)})$
13: **end for**
14: **for** each chunk $\boldsymbol{M}_i^{(j)}$ of $\boldsymbol{M}_i$ **do** ▷ Same process for $\boldsymbol{M}_i$
15: $\mathbf{g}_{\boldsymbol{M}} \leftarrow \mathbf{0}$
16: **for** $p \leftarrow 1, P$ **do**
17: Sample $\mathbf{v} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, $\hat{\mathbf{v}} \leftarrow \mathbf{v}/\|\mathbf{v}\|$
18: $\boldsymbol{M}_i^{(j)} \mathrel{+}= \varepsilon\,\hat{\mathbf{v}}$; $\mathcal{L}_{\boldsymbol{M}}^+ \leftarrow \mathrm{CrossEntropy}(\mathbf{y}_{\text{batch}}, \boldsymbol{a}_i\,\boldsymbol{M}_i)$
19: $\boldsymbol{M}_i^{(j)} \mathrel{-}= 2\varepsilon\,\hat{\mathbf{v}}$; $\mathcal{L}_{\boldsymbol{M}}^- \leftarrow \mathrm{CrossEntropy}(\mathbf{y}_{\text{batch}}, \boldsymbol{a}_i\,\boldsymbol{M}_i)$
20: $\boldsymbol{M}_i^{(j)} \mathrel{+}= \varepsilon\,\hat{\mathbf{v}}$ ▷ Restore matrix
21: $\mathbf{g}_{\boldsymbol{M}} \mathrel{+}= \dfrac{\mathcal{L}_{\boldsymbol{M}}^+ - \mathcal{L}_{\boldsymbol{M}}^-}{2\varepsilon} \cdot \hat{\mathbf{v}}$
22: **end for**
23: $\boldsymbol{M}_i^{(j)} \leftarrow \boldsymbol{M}_i^{(j)} - \eta\,\dfrac{n_{\text{chunk},M}^{(j)}}{P}\,\mathbf{g}_{\boldsymbol{M}}$
24: **end for**
25: $\mathbf{z}_i \leftarrow \boldsymbol{a}_{i-1}\,\boldsymbol{W}_i$; $\boldsymbol{a}_i \leftarrow \mathrm{ReLU}(\mathbf{z}_i)$ ▷ Forward pass for next layer
26: **end for**

Batch Normalization stability. Applying two perturbed forward passes per perturbation direction through a CNN block introduces a practical complication: even though the weight perturbation is reversed after each pair of passes, the running statistics (`running_mean` and `running_var`) maintained by the BatchNorm layers are updated as a side-effect of each forward pass. If left uncorrected, these updates accumulate across all perturbation directions within a single training step, corrupting the running statistics and causing training instability. To prevent this, the running statistics of every BatchNorm layer inside a block are snapshotted before the perturbation passes and restored afterwads, ensuring that only the clean, unperturbed forward pass contributes to the running statistics.

Spatial dimensionality reduction using random projections (CNN). CNN blocks adapt the random projection architecture introduced in FFZero [12] with a small modification. After each forward pass through a block step, the spatial feature map of shape $[B, C, H, W]$ is projected to a compact representation before

being mapped to class scores by $\boldsymbol{M}_l$. The reason for this is because the spatial dimensions H and W can be large, and flattening the feature map would produce an extremely high-dimensional input to $\boldsymbol{M}_l$, making the projection matrix prohibitively large, which can cause DD training to be computationally infeasible.

A fixed random projection matrix $\mathbf{A}_c \in \mathbb{R}^{d \times HW}$ is maintained per output channel c and initialized as $\mathbf{A}_c \sim \mathcal{N}(0, 1/HW)$. The variance $1/HW$ is chosen so that the projected variance is independent of the spatial resolution HW, ensuring consistent input scale to $\boldsymbol{M}_l$ across layers (see Appendix B).

The activations of each channel are projected via $\mathbf{z}_c = \mathbf{A}_c\, \mathbf{u}_c$, where $\mathbf{u}_c \in \mathbb{R}^{HW}$ is the flattened spatial activation, yielding a compact vector $\mathbf{z}_c \in \mathbb{R}^d$. The per-channel projections are concatenated to form a flat representation $\mathbf{z} \in \mathbb{R}^{Cd}$, which is then multiplied by $\boldsymbol{M}_l \in \mathbb{R}^{Cd \times m}$ to produce the goodness scores. The matrices $\mathbf{A}_c$ are fixed and non-trainable, so that only $\boldsymbol{M}_l$ and the convolutional weights are updated during training. Keeping d small substantially reduces the size of $\boldsymbol{M}_l$, making the row-chunked perturbation updates more computationally feasible. The projection matrices are initialized lazily on the first forward pass once the spatial dimensions of the input are known.

A preliminary sweep over $d \in \{8, 16, 32, 64\}$ on FashionMNIST and CIFAR-10 (MF+DD only, 2 seeds) found broadly similar performance across all values, with no clear benefit from larger d; $d = 16$ was chosen as the default, balancing projection capacity against chunk size. Given the limited scope of this test, its effect on performance is not conclusively characterized (see Section 6.1).

4.5 Hyperparameters

All three training methods share a common set of hyperparameters to ensure a fair comparison. The learning rate is set to $\eta = 10^{-3}$ and the batch size to 256 for all datasets and architectures. Training runs for 200 epochs.

For MF+DD, two method-specific hyperparameters govern the quality of the gradient estimates. The perturbation magnitude, which is fixed at $\varepsilon = 10^{-3}$, and the number of perturbation directions per update step is P. The effect of P is the subject of RQ2 and is chosen over values $P \in \{1, 2, 4, 8\}$. The maximum chunk size is set to $C_{\max} = 50{,}000$ parameters per chunk, which bounds the effective n_{chunk} scaling factor in each gradient update and maintains training stability across all architectures evaluated.

The value of $C_{\max}$ was determined based on the number of parameters in the layers of the MLP architecture used for the MNIST dataset, which is the smallest and least complex setting evaluated. Based on preliminary trials, this architecture, with 50 neurons per hidden layer, showed stable training without chunking. Therefore the dimensionality of this model was chosen as a reference point for setting $C_{\max}$. With input dimensions of 784 and hidden layer widths of 50, the largest parameter count in a single layer is $n \approx 40{,}000$ parameters (for the first layer), therefore setting $C_{\max} = 50{,}000$ ensures that no layer is split into more than one chunk in this architecture, and also provides a buffer to accommodate the larger layers in the more complex architectures evaluated, while keeping the number of chunks per layer low enough to avoid excessive computational overhead from multiple perturbations.

For the CNN architecture, the channel-wise spatial projection dimension is set to $d = 16$. With C output channels, this yields a projection matrix $\boldsymbol{M}_l \in \mathbb{R}^{16C \times m}$, substantially smaller than a full spatial flattening would produce, making per-chunk perturbation updates feasible even for wider CNN blocks.

4.6 Hardware

All experiments were conducted using the DelftBlue supercomputer of Delft High Performance Computing Centre (DHPC) [18], using NVIDIA A100 GPU partitions. Batch jobs were split into individual SLURM tasks, each allocated a GPU instance with 10 GB of memory and 2 CPU cores, running on the `gpu-a100-small` partition with CUDA 12.9.

5 Experiments and Analysis

5.1 RQ1: Accuracy Cost of Eliminating Automatic Differentiation

Tables 1 and 2 report test accuracy for all three training methods across the four benchmark datasets for the MLP and CNN architectures respectively.[1] All results are averaged over 3 independent random seeds, with mean and standard deviation reported. For MF+AD and MF+DD, cumulative-prediction is the primary metric, with final-layer prediction mode shown in parentheses. MF+DD uses $P{=}4$ perturbation directions; the sensitivity to this choice is further examined in Section 5.2. The Δ column reports the accuracy gap between MF+AD and MF+DD, with cumulative-prediction as the primary value and final-layer prediction mode in parentheses.

Table 1: Test accuracy for MLP architectures, averaged over 3 seeds (mean $\pm$ std). Cumulative-prediction mode is the primary metric; final-layer prediction mode in parentheses. Δ = MF+DD $-$ MF+AD with cumulative-prediction, with final-layer prediction mode gaps in parentheses. Best cumulative-prediction results per dataset in **bold**.

Dataset	BP	MF+AD	MF+DD ($P{=}4$)	Δ
MNIST	**0.973** $\pm$ 0.001	0.950 $\pm$ 0.001 (0.948 $\pm$ 0.001)	0.933 $\pm$ 0.002 (0.932 $\pm$ 0.000)	-0.017 (-0.015)
FashionMNIST	**0.879** $\pm$ 0.004	0.860 $\pm$ 0.003 (0.858 $\pm$ 0.004)	0.837 $\pm$ 0.002 (0.821 $\pm$ 0.000)	-0.023 (-0.037)
CIFAR-10	0.486 $\pm$ 0.004	**0.526** $\pm$ 0.004 (0.523 $\pm$ 0.003)	0.376 $\pm$ 0.001 (0.300 $\pm$ 0.001)	-0.150 (-0.222)
CIFAR-100	**0.208** $\pm$ 0.004	0.189 $\pm$ 0.003 (0.149 $\pm$ 0.002)	0.109 $\pm$ 0.004 (0.069 $\pm$ 0.002)	-0.080 (-0.080)

For MLPs, BP achieves the highest accuracy on MNIST, FashionMNIST, and CIFAR-100, while MF+AD in cumulative-prediction performs best on CIFAR-10. MF+DD performs close to MF+AD on MNIST and FashionMNIST, with a delta Δ of -0.017 and -0.023 respectively, and a larger gap on CIFAR-10 and CIFAR-100, with Δ of -0.150 and -0.080 respectively. Similar trends are observed with final-layer mode, with MF+DD still lagging significantly behind MF+AD on the more complex datasets.

Table 2: Test accuracy for CNN architectures, averaged over 3 seeds (mean $\pm$ std). Cumulative-prediction mode is the primary metric; final-layer prediction mode in parentheses. Δ = MF+DD $-$ MF+AD with cumulative-prediction, with final-layer prediction mode gaps in parentheses. Best cumulative-prediction results per dataset in **bold**.

Dataset	BP	MF+AD	MF+DD ($P{=}4$)	Δ
MNIST	**0.989** $\pm$ 0.001	0.965 $\pm$ 0.001 (0.971 $\pm$ 0.001)	0.949 $\pm$ 0.002 (0.956 $\pm$ 0.003)	-0.016 (-0.015)
FashionMNIST	**0.911** $\pm$ 0.001	0.883 $\pm$ 0.004 (0.888 $\pm$ 0.004)	0.842 $\pm$ 0.004 (0.827 $\pm$ 0.003)	-0.041 (-0.061)
CIFAR-10	**0.731** $\pm$ 0.003	0.552 $\pm$ 0.007 (0.540 $\pm$ 0.006)	0.418 $\pm$ 0.009 (0.387 $\pm$ 0.004)	-0.134 (-0.153)
CIFAR-100	**0.430** $\pm$ 0.002	0.274 $\pm$ 0.003 (0.268 $\pm$ 0.002)	0.176 $\pm$ 0.005 (0.151 $\pm$ 0.004)	-0.098 (-0.117)

For CNNs, BP is the upper bound for all datasets. The gap between MF+AD and MF+DD is similar to the trend in MLPs, with a small delta on simpler datasets and a larger gap on more complex datasets.

The delta values observed in both architectures suggest that while eliminating AD can be done with minimal accuracy loss on simpler datasets, it becomes more challenging as dataset complexity increases. This is consistent with the theoretical variance of the directional derivative estimator scaling with the parameter

[1] Absolute accuracy figures for MF+AD and BP are not directly comparable to those reported in Gong et al. [10] The architectures used here are deliberately smaller than those in the original paper, in order to keep DD perturbation chunks computationally feasible. The MF+AD architecture also uses the random projection based dimensionality reduction approach mentioned in 4.4. Gong et al. use MLP widths up to 1000 and a four-block CNN with up to 512 channels with a full hyperparameter grid search.

space dimension: as input and model complexity grow, more perturbation directions are needed to maintain a reliable gradient signal.

Notably, MF+AD outperforms BP in cumulative-prediction mode on CIFAR-10 with the MLP architecture (0.526 vs. 0.486); this can be a form of implicit regularization, particularly effective when the model is small relative to the data complexity. However, this advantage does not hold for the CNN architecture on the same dataset, where BP achieves higher accuracy (0.731 vs. 0.552). This suggests that the benefits of MF+AD over BP may be architecture-dependent, as CNNs with MF+AD use the spatial projection approach for computing goodness scores, which may have caused information loss that is more detrimental in the absence of AD with complex data.

Additionally, the gaps in CIFAR-100 is smaller than the gaps in CIFAR-10 with both architectures. This may be architecture dependent, as the architectures were deliberately kept small to keep the computational cost manageable. The architectures may be too small to capture the complexity of CIFAR-100, leading to underfitting and a smaller gap between MF+AD and MF+DD.

Overall, these results indicate that MF+DD is a viable AD-free alternative in low-complexity settings, but the gap widens substantially on harder benchmarks, pointing to the need for higher P or improved gradient estimation strategies for more complex tasks.

5.2 RQ2: Effect of Number of Perturbation Directions

To focus on the effect of P from dataset-specific factors, RQ2 is evaluated on FashionMNIST. Figure 2 shows test accuracy for MF+DD across $P \in \{1, 2, 4, 8\}$ for both the MLP and CNN architectures, averaged over 3 seeds. Higher P values were not evaluated due to computational constraints, but $P{=}4$ was found to provide a good balance between accuracy and training stability, while still being less resource-intensive than $P{=}8$.

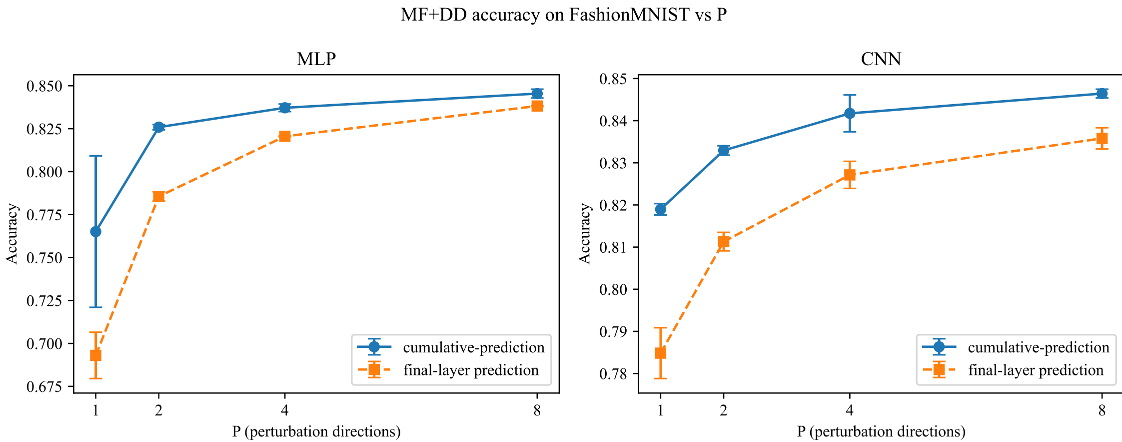

Figure 2: Test accuracy of MF+DD on FashionMNIST for varying P, averaged over 3 seeds (mean ± std), for MLP (left) and CNN (right).

Increasing P improves classification accuracy for both architectures. For the MLP architecture, it was also observed that $P{=}1$ led to unstable training, even in FashionMNIST, which is considered a moderately complex benchmark in the experiments. Larger P values produced stable convergence across all seeds on the same dataset. The unstable training behavior can be observed in Figure 3, which shows the cumulative-prediction

mode validation loss over training epochs for each value of P on FashionMNIST. Higher P leads to smoother training curves, consistent with the variance reduction expected from averaging more gradient directions.

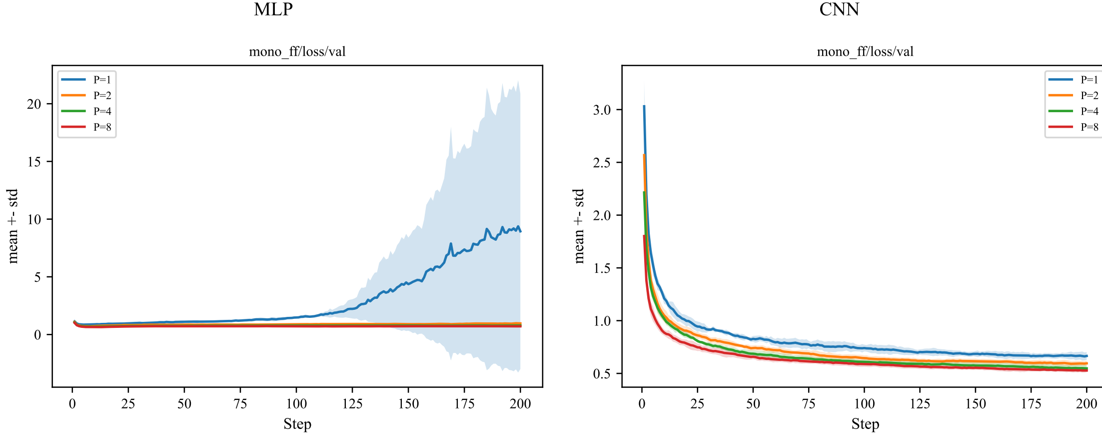

Figure 3: cumulative-prediction validation loss on FashionMNIST over training epochs for MF+DD with $P \in \{1, 2, 4, 8\}$, for MLP (left) and CNN (right). Final-layer prediction mode shows similar trends.

6 Conclusion

This paper investigated the feasibility of training neural networks without global backpropagation *or* automatic differentiation by combining the Mono-Forward algorithm with zeroth-order gradient estimation via directional derivatives. The resulting MF+DD algorithm trains each layer independently using only two forward evaluations of its local cross-entropy loss per perturbation direction, without the need for any form of automatic differentiation. Two research questions were studied.

RQ1: Accuracy cost of replacing AD with directional derivatives. MF+DD stays close to the AD-based Mono-Forward baseline on MNIST and FashionMNIST, with Δ ranging from -0.016 to -0.041 across both architectures. The gap is larger on CIFAR, but not monotonically so: $\Delta = -0.150$ (MLP) / -0.134 (CNN) on CIFAR-10, versus -0.080 (MLP) / -0.098 (CNN) on CIFAR-100, smaller than CIFAR-10 despite the harder task, most likely due to the small architectures' limited capacity for the harder task (see Section 6.1). BP remains the clear upper bound throughout, except on CIFAR-10 with the MLP architecture, where MF+AD outperforms BP. These results suggest that with easier tasks, MF+DD can achieve performance close to AD-based methods, while on harder tasks, the gap widens, indicating the need for further improvements in gradient quality or model capacity.

RQ2: Effect of the number of perturbation directions P. Increasing P consistently improves both final accuracy and training stability for MF+DD. At P=1, training is unstable even on the moderately complex benchmark, and variance across seeds is high. Already at P=4, convergence becomes reliable and accuracy approaches the P=8 ceiling. The improvement from P=4 to P=8 is smaller than from P=1 to P=4, with a significant increase in training time (See Appendix C), indicating diminishing returns. Given that each additional direction doubles the number of forward passes required per layer update, P=4 offers a practical balance between gradient quality and computational cost.

6.1 Limitations

The primary limitation of MF+DD is computational cost. The $2P$ forward passes per chunk per layer make training significantly slower than AD-based methods. The need to keep chunk sizes small for gradient stability, also limits the scale of architectures that can be trained practically. The architectures evaluated in this work are deliberately small; as a result, the degree to which the findings generalise to deeper or wider networks remains an open question. This capacity constraint likely also explains why the MF+DD$-$MF+AD gap on CIFAR-100 is smaller than on CIFAR-10 (Section 5.1): with insufficient capacity to fit the harder 100-class task even under AD, BP and MF+AD accuracy are themselves depressed on CIFAR-100, which narrows the apparent gap rather than indicating that directional derivatives scale better to harder tasks.

Second, all experiments are restricted to image classification on standard benchmarks; the behaviour of MF+DD on other task types is not studied. Additionally, the MF+DD training procedure introduces several method-specific hyperparameters (ε, P, and, for CNNs, the spatial projection dimension d) whose optimal values may depend on architecture and dataset, and no hyperparameter search was performed beyond the P-sweep in RQ2 due to computational constraints. The choice of d for the CNN spatial projection was evaluated only with a limited number of seeds and datasets, so its effect on performance is not conclusively characterised.

Lastly, this work is also purely theoretical and algorithmic in nature, as no physical hardware experiments were conducted. Therefore, the claims regarding the suitability of MF+DD for hardware without AD support remain at the algorithmic level and have not been validated in practice.

6.2 Future Work

The most direct next step is to evaluate the MF+DD method on physical hardware platforms for which AD is genuinely unavailable, which would provide direct evidence for the practical claims motivating the method. On the algorithmic side, future work could address the computational cost by exploring adaptive perturbation strategies that allocate more directions to high-loss layers, which could provide a balance between gradient quality and computational cost.

Finally, in order to reduce the effect of the noise inherent in DD based gradient estimation, the usage of optimizers such as Adam [19] can be explored. Adam is particularly well-suited for very noisy gradients, as it maintains a running average of both the first and second moments of the gradients, normalizing each update by the gradient's historical variance. This adaptive scaling could help suppress the high variance introduced by single-direction perturbation estimates, potentially improving convergence stability without requiring a large P.

7 Responsible Research

This section is included here to provide transparency about the research process.

Use of AI. AI-based tools were used in the preparation of this paper. Specifically, a large language model was used to improve grammar and phrasing in written sections, and to generate boilerplate LaTeX code for tables and figures. Additionally, AI was used to assist with code formatting and debugging during the implementation of the MF+DD algorithm. All AI-generated content was thoroughly reviewed and edited to ensure accuracy and clarity.

Reproducibility. All experiments were conducted using publicly available benchmark datasets: MNIST [14], FashionMNIST [15], CIFAR-10, and CIFAR-100 [16]. The proposed MF+DD algorithm, network architectures, hyperparameters, and training protocols are described in full in Sections 3, 4.5 and Appendix A.

All results are averaged over 3 independent random seeds, to account for variance introduced by random weight initialisation and the stochastic perturbation directions used in directional derivative estimation. The implementation uses PyTorch [3] with CUDA 12.9 , making the code portable across standard GPU hardware. The code for the training pipeline and the analysis can be found at `https://github.com/gorpeliates/ad-free-monofwd`.

Computational resources. The MF+DD method is computationally more expensive than both BP and MF+AD at training time. For $P{=}8$, this results in substantially higher wall-clock training times (See Figure C). Experiments involving the full P-sweep and cross-dataset comparisons would require a non-trivial computational budget. Training cost is further amplified on the CIFAR datasets, which use larger MLP and CNN architectures than MNIST/FashionMNIST, leading to longer training times across all methods (see Figure C). To make sure the computational resources were used efficiently and they were not blocked for a long period of time for other users, all training runs were split into different SLURM jobs on `gpu-a100-small` partitions, which do not use the full GPU capacity.

References

[1] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. doi: 10.1038/323533a0.

[2] Atılım Güneş Baydin, Barak A. Pearlmutter, Don Syme, Frank Wood, and Philip Torr. Gradients without backpropagation, 2022. URL `https://arxiv.org/abs/2202.08587`.

[3] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. URL `https://arxiv.org/abs/1912.01703`.

[4] Rongguang Ye, Chenhao Ye, Chao Huang, Ming Tang, and Yunhao Liu. Beyond-backpropagation training: Methods, applications, and perspectives. *TechRxiv*, 2026(0103), 2026. doi: 10.36227/techrxiv.176740426.63642005/v1. URL `https://www.techrxiv.org/doi/abs/10.36227/techrxiv.176740426.63642005/v1`.

[5] Sergey Bartunov, Adam Santoro, Blake A. Richards, Luke Marris, Geoffrey E. Hinton, and Timothy Lillicrap. Assessing the scalability of biologically-motivated deep learning algorithms and architectures, 2018. URL `https://arxiv.org/abs/1807.04587`.

[6] Ali Momeni, Babak Rahmani, Benjamin Scellier, et al. Training of physical neural networks. *Nature*, 645:53–61, 2025. doi: 10.1038/s41586-025-09384-2.

[7] Arild Nøkland. Direct feedback alignment provides learning in deep neural networks, 2016. URL `https://arxiv.org/abs/1609.01596`.

[8] Benjamin Scellier and Yoshua Bengio. Equilibrium propagation: Bridging the gap between energy-based models and backpropagation, 2017. URL `https://arxiv.org/abs/1602.05179`.

[9] Geoffrey Hinton. The forward-forward algorithm: Some preliminary investigations, 2022. URL `https://arxiv.org/abs/2212.13345`.

[10] James Gong, Bruce Li, and Waleed Abdulla. Mono-forward: Revisiting forward-forward through objective-locality decomposition, 2026. URL `https://arxiv.org/abs/2501.09238`.

[11] Sijia Liu, Pin-Yu Chen, Bhavya Kailkhura, Gaoyuan Zhang, Alfred O. Hero III, and Pramod K. Varshney. A primer on zeroth-order optimization in signal processing and machine learning: Principals, recent advances, and applications. *IEEE Signal Processing Magazine*, 37(5):43–54, 2020. doi: 10.1109/MSP.2020.3003837.

[12] Yaqi Guo, Fabian Braun, Bastiaan Ketelaar, Stephanie Tan, Richard Norte, and Siddhant Kumar. Local learning for stable backpropagation-free neural network training towards physical learning, 2026. URL https://arxiv.org/abs/2603.24790.

[13] PyTorch. A gentle introduction to torch.autograd. https://docs.pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html, 2025. Accessed: 2026-06-19.

[14] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

[15] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017. URL https://arxiv.org/abs/1708.07747.

[16] Alex Krizhevsky. Learning multiple layers of features from tiny images. https://cave.cs.toronto.edu/kriz/learning-features-2009-TR.pdf, 2009.

[17] Stephen J. Wright. Coordinate descent algorithms, 2015. URL https://arxiv.org/abs/1502.04759.

[18] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2, 2024.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

A Model Architectures

Tables 3 and 4 give the complete architectural specifications for the MLP and CNN models used in all experiments.

MLP block structure. Each MLP block consists of a fully connected layer followed by ReLU activation. In MF+AD and MF+DD, each block additionally maintains a trainable projection matrix $\mathbf{M}_l \in \mathbb{R}^{n_{\text{out}} \times C}$ used to compute local goodness scores, where C is the number of classes. In BP, a single linear classifier head $\mathbb{R}^{n_{\text{out}} \to C}$ is appended after the final block.

CNN block structure. Each CNN block follows the *Conv2d(kernel_size=3, stride=1, padding=1)*, *BatchNorm2d*, *ReLU*, *MaxPool2d(kernel_size=2, stride=2)* structure.

In MF+AD and MF+DD, each block additionally uses a fixed random projection matrix $\mathbf{A} \in \mathbb{R}^{C_{\text{out}} \times d \times HW}$ (non-trainable buffer) to compress the spatial activations per channel, followed by a trainable projection matrix $\mathbf{M}_l \in \mathbb{R}^{C_{\text{out}} d \times C}$. In BP, the spatial feature map is flattened and passed to a linear classifier $\mathbb{R}^{C_{\text{out}} HW \to C}$ appended after the final block, where H, W is the feature map size *after* the final block's MaxPool.

Table 3: MLP architecture specifications per dataset. All blocks within a dataset share the same width. Parameter counts include the projection matrices $\mathbf{M}_l$ for MF+AD/MF+DD, and the final classification head for BP.

Dataset	Input dim	Depth	Width	Classes	MF params	BP params
MNIST	784	2	50	10	42,800	42,310
FashionMNIST	784	3	100	10	101,700	99,710
CIFAR-10	3,072	4	200	10	743,200	737,210
CIFAR-100	3,072	4	200	100	815,200	755,300

Table 4: CNN architecture specifications per dataset. Both architectures use 2 convolutional blocks with the same Conv, BN, ReLU, MaxPool(2×2) structure. *Channels* lists the output channel count for each block; *Spatial (B1 / B2)* lists the feature map size *before* MaxPool at each block, which is the size used when computing the per-channel random projection for goodness scores; the activations passed to the next block are halved by the subsequent MaxPool. The random projection dimension is d=16 for all MF+AD/MF+DD models; the corresponding $\mathbf{A}$ matrices are non-trainable buffers and are excluded from the MF parameter counts.

Dataset	In ch.	Channels	Spatial (B1 / B2)	d	Classes	MF params	BP params
MNIST	1	[32, 32]	28×28 / 14×14	16	10	19,936	25,386
FashionMNIST	1	[64, 64]	28×28 / 14×14	16	10	58,304	69,194
CIFAR-10	3	[128, 128]	32×32 / 16×16	16	10	192,640	233,610
CIFAR-100	3	[128, 128]	32×32 / 16×16	16	100	561,280	970,980

B Variance of the CNN Spatial Random Projection

This section derives the variance of the projected activation entries and show why the $\mathcal{N}(0, 1/HW)$ initialization is necessary for resolution-stable training.

Let $\mathbf{u}_c \in \mathbb{R}^{HW}$ denote the flattened spatial activation map for channel c after a Conv–BatchNorm–ReLU block. Because ReLU is applied after BatchNorm, the post-ReLU activations u_j are not Gaussian: negative

pre-activations are clipped to zero, so $\mathbb{E}[u_j]$ and $\mathrm{Var}(u_j)$ differ from the BatchNorm parameters β and γ^2. The resulting value only relies on $\mathbb{E}[u_j^2]$, which does not depend on the spatial index j or on HW.

The random projection matrix $\mathbf{A}_c \in \mathbb{R}^{d\times HW}$ has entries drawn i.i.d. as $A_{ij} \sim \mathcal{N}(0, \sigma^2)$ for some variance σ^2 to be determined. Crucially, $\mathbf{A}_c$ is fixed prior to training and independent of $\mathbf{u}_c$. Each entry of the projected vector $\mathbf{z}_c = \mathbf{A}_c\mathbf{u}_c$ is:

$$z_i = \sum_{j=1}^{HW} A_{ij}\, u_j \tag{17}$$

Expand the variance of z_i:

$$\mathrm{Var}(z_i) = \sum_{j=1}^{HW} \mathrm{Var}(A_{ij}u_j) \tag{18}$$

For each term, since A_{ij} and u_j are independent and $\mathbb{E}[A_{ij}] = 0$:

$$\mathrm{Var}(A_{ij}u_j) = \mathrm{Var}(A_{ij})\,\mathbb{E}[u_j^2] = \sigma^2\,\mathbb{E}[u_j^2] \tag{19}$$

Summing over all HW spatial locations:

$$\mathrm{Var}(z_i) = \sigma^2 \cdot HW \cdot \mathbb{E}[u_j^2] \tag{20}$$

We want $\mathrm{Var}(z_i)$ to be independent of HW, so that $\mathbf{M}_l$ receives inputs at a consistent scale regardless of which layer the projection is attached to. Setting $\sigma^2 = 1/HW$ cancels the spatial dimension exactly:

$$\boxed{\mathrm{Var}(z_i) = \mathbb{E}[u_j^2]} \tag{21}$$

The resulting variance depends on the post-ReLU activations, and is not dependent on the spatial dimension HW.

C Training Times for Experiments

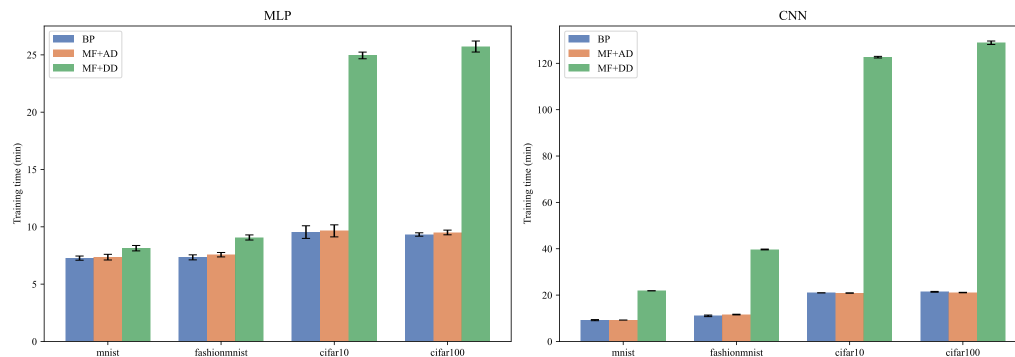

Figure 4: Training time per method and dataset (mean $\pm$ std across seeds)

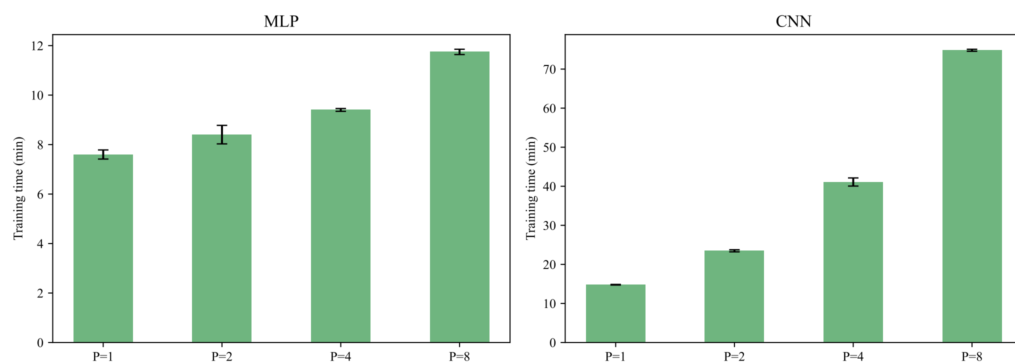

Figure 5: Training time per for $P \in 1, 2, 4, 8$ for the FashionMNIST dataset using the CNN and MLP architectures described in Appendix A (mean $\pm$ std across seeds)