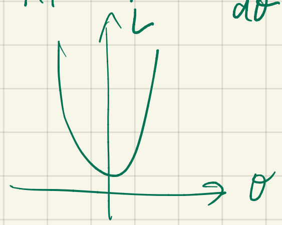
$$L = \sum_{i=1}^{n} (a_i - z_i)^2 \Rightarrow \frac{dL}{dz_i} = 2(a_i - z_i)$$

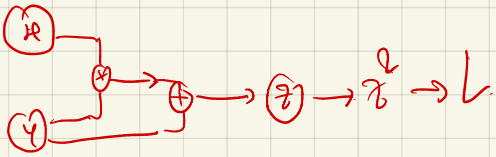
$$z_i = x_i\theta + \theta^2 \Rightarrow \frac{dz}{d\theta} = x + 2y \Rightarrow \frac{dL}{dy} = \frac{dL}{dz} \cdot \frac{dz}{dy}$$

(input x_i)

gradient optimization

$$\theta_{i+1} = \theta_i - r\frac{dL}{d\theta}$$





back pro (exact ∇f)

→ forward: ใช้การหาค่า แต่ละค่า

← backward: คำนวณ $\frac{dL}{dz} \to \frac{dz}{dy_i} \to \ldots$

---

purpose (Expect ∇f)

→ forward: ∇f, & ขนาด

} ประหยัด time ในการ optimize

use only forward

ใช้การหาเพื่อหา ไม่จำเป็นต้องใช้ ∇f โดยตรง

เพราะว่า

$$g(\theta) = (\nabla f \cdot v)v$$

$\nabla_v f$ → เป็นการ projection บนแกนของ v

$$\nabla_v f = \lim_{h \to 0} \frac{f(\theta + hv) - f(\theta)}{h}$$ Ar.

Gradients without Backpropagation

Atılım Güneş Baydin [1] **Barak A. Pearlmutter** [2] **Don Syme** [3] **Frank Wood** [4] **Philip Torr** [5]

Abstract

Using backpropagation to compute gradients of objective functions for optimization has remained a mainstay of machine learning. Backpropagation, or reverse-mode differentiation, is a special case within the general family of automatic differentiation algorithms that also includes the forward mode. We present a method to compute gradients based solely on the directional derivative that one can compute exactly and efficiently via the forward mode. We call this formulation the **forward gradient**, an unbiased estimate of the gradient that can be evaluated in a single forward run of the function, entirely eliminating the need for backpropagation in gradient descent. We demonstrate forward gradient descent in a range of problems, showing substantial savings in computation and enabling training up to twice as fast in some cases.

1. Introduction

Backpropagation (Linnainmaa, 1970; Rumelhart et al., 1985) and gradient-based optimization have been the core algorithms underlying many recent successes in machine learning (ML) (Goodfellow et al., 2016; Deisenroth et al., 2020). It is generally accepted that one of the factors contributing to the recent pace of advance in ML has been the ease with which differentiable ML code can be implemented via well engineered libraries such as PyTorch (Paszke et al., 2019) or TensorFlow (Abadi et al., 2016) with automatic differentiation (AD) capabilities (Griewank & Walther, 2008; Baydin et al., 2018). These frameworks provide the computational infrastructure on which our field is built.

Until recently, all major software frameworks for ML have been built around the reverse mode of AD, a technique to evaluate derivatives of numeric code using a two-phase forward–backward algorithm, of which backpropagation is a special case conventionally applied to neural networks. This is mainly due to the central role of scalar-valued objectives in ML, whose gradient with respect to a very large number of inputs can be evaluated exactly and efficiently with a single evaluation of the reverse mode.

Reverse mode is a member of a larger family of AD algorithms that also includes the forward mode (Wengert, 1964), which has the favorable characteristic of requiring only a single forward evaluation of a function (i.e., not involving any backpropagation) at a significantly lower computational cost. Crucially, forward and reverse modes of AD evaluate different quantities. Given a function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$, forward mode evaluates the Jacobian–vector product $\boldsymbol{J_f v}$, where $\boldsymbol{J_f} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{v} \in \mathbb{R}^n$; and revese mode evaluates the vector–Jacobian product $\boldsymbol{v}^\intercal \boldsymbol{J_f}$, where $\boldsymbol{v} \in \mathbb{R}^m$. For the case of $f : \mathbb{R}^n \to \mathbb{R}$ (e.g., an objective function in ML), forward mode gives us $\nabla f \cdot \boldsymbol{v} \in \mathbb{R}$, the directional derivative; and reverse mode gives us the full gradient $\nabla f \in \mathbb{R}^n$.[1]

From the perspective of AD applied to ML, a "holy grail" is whether the practical usefulness of gradient descent can be achieved using only the forward mode, eliminating the need for backpropagation. This could potentially change the computational complexity of typical ML training pipelines, reduce the time and energy costs of training, influence ML hardware design, and even have implications regarding the biological plausibility of backpropagation in the brain (Bengio et al., 2015; Lillicrap et al., 2020). In this work we present results that demonstrate stable gradient descent over a range of ML architectures using only forward mode AD.

Contributions

- We define the "forward gradient", an estimator of the gradient that we prove to be unbiased, based on forward mode AD without backpropagation.
- We implement a forward AD system from scratch in PyTorch, entirely independent of the reverse AD implementation already present in this library.
- We use forward gradients in stochastic gradient descent (SGD) optimization of a range of architectures, and show

[1]Department of Computer Science, University of Oxford [2]Department of Computer Science, National University of Ireland Maynooth [3]Microsoft [4]Computer Science Department, University of British Columbia [5]Department of Engineering Science, University of Oxford. Correspondence to: Atılım Güneş Baydin <gunes@robots.ox.ac.uk>.

Under Review.

[1]We represent ∇f as a column vector.

that a typical modern ML training pipeline can be constructed with only forward AD and no backpropagation.

- We compare the runtime and loss performance characteristics of forward gradients and backpropagation, and demonstrate speedups of up to twice as fast compared with backpropagation in some cases.

A note on naming: When naming the technique, it is tempting to adopt names like "forward propagation" or "forward-prop" to contrast it with backpropagation. We do not use this name as it is commonly used to refer to the forward evaluation phase of backpropagation, distinct from forward AD. We observe that the simple name "forward gradient" is currently not used in ML, and it also captures the aspect that we are presenting a drop-in replacement for the gradient.

2. Background

In order to introduce our method, we start by briefly reviewing the two main modes of automatic differentiation.

2.1. Forward Mode AD

$$\begin{matrix}\boldsymbol{\theta} \\ \boldsymbol{v}\end{matrix} \xrightarrow{\quad\text{Forward}\quad} \begin{matrix}\boldsymbol{f}(\boldsymbol{\theta}) \\ \boldsymbol{J_f}(\boldsymbol{\theta})\,\boldsymbol{v}\end{matrix}$$

Given a function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$ and the values $\boldsymbol{\theta} \in \mathbb{R}^n$, $\boldsymbol{v} \in \mathbb{R}^n$, forward mode AD computes $\boldsymbol{f}(\boldsymbol{\theta})$ and the *Jacobian–vector product*[2] $\boldsymbol{J_f}(\boldsymbol{\theta})\,\boldsymbol{v}$, where $\boldsymbol{J_f}(\boldsymbol{\theta}) \in \mathbb{R}^{m\times n}$ is the Jacobian matrix of all partial derivatives of $\boldsymbol{f}$ evaluated at $\boldsymbol{\theta}$, and $\boldsymbol{v}$ is a vector of perturbations.[3] For the case of $f : \mathbb{R}^n \to \mathbb{R}$ the Jacobian–vector product corresponds to a directional derivative $\nabla f(\boldsymbol{\theta}) \cdot \boldsymbol{v}$, which is the projection of the gradient ∇f at $\boldsymbol{\theta}$ onto the direction vector $\boldsymbol{v}$, representing the rate of change along that direction.

It is important to note that the forward mode evaluates the function $\boldsymbol{f}$ and its Jacobian–vector product $\boldsymbol{J_f v}$ *simultaneously in a single forward run*. Also note that $\boldsymbol{J_f v}$ is obtained without having to compute the Jacobian $\boldsymbol{J_f}$, a feature sometimes referred to as a matrix-free computation.[4]

2.2. Reverse Mode AD

$$\begin{matrix}\boldsymbol{\theta} \\ \boldsymbol{v}^\intercal \boldsymbol{J_f}(\boldsymbol{\theta})\end{matrix} \underset{\text{Backward}}{\overset{\text{Forward}}{\rightleftarrows}} \begin{matrix}\boldsymbol{f}(\boldsymbol{\theta}) \\ \boldsymbol{v}\end{matrix}$$

Given a function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$ and the values $\boldsymbol{\theta} \in \mathbb{R}^n$, $\boldsymbol{v} \in \mathbb{R}^m$, reverse mode AD computes $\boldsymbol{f}(\boldsymbol{\theta})$ and the *vector–Jacobian product*[5] $\boldsymbol{v}^\intercal \boldsymbol{J_f}(\boldsymbol{\theta})$, where $\boldsymbol{J_f} \in \mathbb{R}^{m\times n}$ is the Jacobian matrix of all partial derivatives of $\boldsymbol{f}$ evaluated at $\boldsymbol{\theta}$, and $\boldsymbol{v} \in \mathbb{R}^m$ is a vector of adjoints. For the case of $f : \mathbb{R}^n \to \mathbb{R}$ and $v = 1$, reverse mode computes the gradient, i.e., the partial derivatives of f w.r.t. all n inputs $\nabla f(\boldsymbol{\theta}) = \left[\frac{\partial f}{\partial \theta_1}, \ldots, \frac{\partial f}{\partial \theta_n}\right]^\intercal$.

Note that $\boldsymbol{v}^\intercal \boldsymbol{J_f}$ is computed in a single forward–backward evaluation, without having to compute the Jacobian $\boldsymbol{J_f}$.[6]

2.3. Runtime Cost

The runtime costs of both modes of AD are bounded by a constant multiple of the time it takes to run the function $\boldsymbol{f}$ we are differentiating (Griewank & Walther, 2008). Reverse mode has a higher cost than forward mode, because it involves data-flow reversal and it needs to keep a record (a "tape", stack, or graph) of the results of operations encountered in the forward pass, because these are needed in the evaluation of derivatives in the backward pass that follows. The memory and computation cost characteristics ultimately depend on the features implemented by the AD system such as exploiting sparsity (Gebremedhin et al., 2005) or checkpointing (Siskind & Pearlmutter, 2018).

The cost can be analyzed by assuming computational complexities of elementary operations such as fetches, stores, additions, multiplications, and nonlinear operations (Griewank & Walther, 2008). Denoting the time it takes to evaluate the original function $\boldsymbol{f}$ as runtime($\boldsymbol{f}$), we can express the time taken by the forward and reverse modes as $R_f \times$ runtime($\boldsymbol{f}$) and $R_b \times$ runtime($\boldsymbol{f}$) respectively. In practice, R_f is typically between 1 and 3, and R_b is typically between 5 and 10 (Hascoët, 2014), but these are highly program dependent.

Note that in ML the original function corresponds to the execution of the ML code without any derivative computation or training, i.e., just evaluating a given model with input data.[7] We will call this "base runtime" in this paper.

3. Method

3.1. Forward Gradients

Definition 1. *Given a function $f : \mathbb{R}^n \to \mathbb{R}$, we define the "forward gradient" $\boldsymbol{g} : \mathbb{R}^n \to \mathbb{R}^n$ as*

$$\boldsymbol{g}(\boldsymbol{\theta}) = (\nabla f(\boldsymbol{\theta}) \cdot \boldsymbol{v})\,\boldsymbol{v}\,, \tag{1}$$

[2] Popularized recently as a `jvp` operation in tensor frameworks such as JAX (Bradbury et al., 2018).

[3] Also called "tangents".

[4] The full Jacobian $\boldsymbol{J}$ can be computed with forward AD using n forward evaluations of $\boldsymbol{J}\boldsymbol{e}_i$, $i = 1, \ldots n$ using standard basis vectors $\boldsymbol{e}_i$ so that each forward run gives us a single column of $\boldsymbol{J}$.

[5] Popularized recently as a `vjp` operation in tensor frameworks such as JAX (Bradbury et al., 2018).

[6] The full Jacobian $\boldsymbol{J}$ can be computed with reverse AD using m evaluations of $\boldsymbol{e}_i^\intercal \boldsymbol{J}$, $i = 1, \ldots m$ using standard basis vectors $\boldsymbol{e}_i$ so that each run gives us a single row of $\boldsymbol{J}$.

[7] Sometimes called "inference" by practitioners.

where $\boldsymbol{\theta} \in \mathbb{R}^n$ is the point at which we are evaluating the gradient, $\boldsymbol{v} \in \mathbb{R}^n$ is a perturbation vector taken as a multivariate random variable $\boldsymbol{v} \sim p(\boldsymbol{v})$ such that $\boldsymbol{v}$'s scalar components v_i are independent and have zero mean and unit variance for all i, and $\nabla f(\boldsymbol{\theta}) \cdot \boldsymbol{v} \in \mathbb{R}$ is the directional derivative of f at point $\boldsymbol{\theta}$ in direction $\boldsymbol{v}$.

We first talk briefly about the intuition that led to this definition, before showing that $\boldsymbol{g}(\boldsymbol{\theta})$ is an unbiased estimator of the gradient $\nabla f(\boldsymbol{\theta})$ in Section 3.2.

As explained in Section 2, forward mode gives us the directional derivative $\nabla f(\boldsymbol{\theta}) \cdot \boldsymbol{v} = \sum_i \frac{\partial f}{\partial \theta_i} v_i$ directly, without having to compute ∇f. Computing ∇f using only forward mode is possible by evaluating f forward n times with direction vectors taken as standard basis (or one-hot) vectors $\boldsymbol{e}_i \in \mathbb{R}^n, i = 1 \dots n$, where $\boldsymbol{e}_i$ denotes a vector with a 1 in the ith coordinate and 0s elsewhere. This allows the evaluation of the sensitivity of f w.r.t. each input $\frac{\partial f}{\partial \theta_i}$ separately, which when combined give us the gradient ∇f.

In order to have any chance of runtime advantage over backpropagation, we need to work with a single run of the forward mode per optimization iteration, not n runs.[8] In a single forward run, we can interpret the direction $\boldsymbol{v}$ as a weight vector in a weighted sum of sensitivities w.r.t. each input, that is $\sum_i \frac{\partial f}{\partial \theta_i} v_i$, albeit without the possibility of distinguishing the contribution of each θ_i in the final total. We therefore use the weight vector $\boldsymbol{v}$ to attribute the overall sensitivity back to each individual parameter θ_i, proportional to the weight v_i of each parameter θ_i (e.g., a parameter with a small weight had a small contribution and a large one had a large contribution in the total sensitivity).

In summary, each time the forward gradient is evaluated, we simply do the following:

- Sample a random perturbation vector $\boldsymbol{v} \sim p(\boldsymbol{v})$, which has the same size with f's argument $\boldsymbol{\theta}$.
- Run f via forward-mode AD, which evaluates $f(\boldsymbol{\theta})$ *and* $\nabla f(\boldsymbol{\theta}) \cdot \boldsymbol{v}$ *simultaneously* in the same single forward run, **without having to compute ∇f** at all in the process. The directional derivative obtained, $\nabla f(\boldsymbol{\theta}) \cdot \boldsymbol{v}$, is a scalar, and is computed exactly by AD (not an approximation).
- Multiply the scalar directional derivative $\nabla f(\boldsymbol{\theta}) \cdot \boldsymbol{v}$ with vector $\boldsymbol{v}$ and obtain $\boldsymbol{g}(\boldsymbol{\theta})$, the forward gradient.

Figure 1 illustrates the process showing several evaluations of the forward gradient for the Beale function. We see how perturbations v_k (orange) transform into forward gradients $(\nabla f \cdot v_k)v_k$ (blue) for $k \in [1, 5]$, sometimes reversing the sense to point towards the true gradient (red) while being constrained in orientation. The green arrow shows a Monte Carlo gradient estimate via averaged forward gradients, i.e., $\frac{1}{K}\sum_{k=1}^{K}(\nabla f \cdot v_k)v_k \approx \mathbb{E}[(\nabla f \cdot v)v]$.

[8] This requirement can be relaxed depending on the problem setting and we would expect the gradient estimation to get better with more forward runs per optimization iteration.

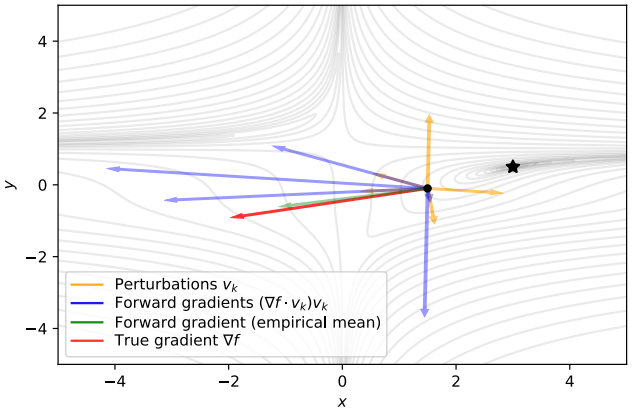


Figure 1. Five samples of forward gradient, the empirical mean of these five samples, and the true gradient for the Beale function (Section 5.1) at $x = 1.5, y = -0.1$. Star marks the global minimum.

3.2. Proof of Unbiasedness

Theorem 1. *The forward gradient $\boldsymbol{g}(\boldsymbol{\theta})$ is an unbiased estimator of the gradient $\nabla f(\boldsymbol{\theta})$.*

Proof. We start with the directional derivative of f evaluated at $\boldsymbol{\theta}$ in direction $\boldsymbol{v}$ written out as follows

$$d(\boldsymbol{\theta}, \boldsymbol{v}) = \nabla f(\boldsymbol{\theta}) \cdot \boldsymbol{v} = \sum_i \frac{\partial f}{\partial \theta_i} v_i$$
$$= \frac{\partial f}{\partial \theta_1} v_1 + \frac{\partial f}{\partial \theta_2} v_2 + \cdots + \frac{\partial f}{\partial \theta_n} v_n \,. \quad (2)$$

We then expand the forward gradient $\boldsymbol{g}$ in Eq. (1) as

$$\boldsymbol{g}(\boldsymbol{\theta}) = d(\boldsymbol{\theta}, \boldsymbol{v})\boldsymbol{v} = \begin{bmatrix} \frac{\partial f}{\partial \theta_1} v_1^2 + \frac{\partial f}{\partial \theta_2} v_1 v_2 + \cdots + \frac{\partial f}{\partial \theta_n} v_1 v_n \\ \frac{\partial f}{\partial \theta_1} v_1 v_2 + \frac{\partial f}{\partial \theta_2} v_2^2 + \cdots + \frac{\partial f}{\partial \theta_n} v_2 v_n \\ \vdots \\ \frac{\partial f}{\partial \theta_1} v_1 v_n + \frac{\partial f}{\partial \theta_2} v_2 v_n + \cdots + \frac{\partial f}{\partial \theta_n} v_n^2 \end{bmatrix}$$

and note that the components of $\boldsymbol{g}$ have the following form

$$g_i(\boldsymbol{\theta}) = \frac{\partial f}{\partial \theta_i} v_i^2 + \sum_{j \neq i} \frac{\partial f}{\partial \theta_j} v_i v_j \,. \quad (3)$$

The expected value of each component g_i is

$$\mathbb{E}\left[g_i(\boldsymbol{\theta})\right] = \mathbb{E}\left[\frac{\partial f}{\partial \theta_i} v_i^2 + \sum_{j \neq i} \frac{\partial f}{\partial \theta_j} v_i v_j\right]$$

$$= \mathbb{E}\left[\frac{\partial f}{\partial \theta_i} v_i^2\right] + \mathbb{E}\left[\sum_{j \neq i} \frac{\partial f}{\partial \theta_j} v_i v_j\right]$$
$$= \mathbb{E}\left[\frac{\partial f}{\partial \theta_i} v_i^2\right] + \sum_{j \neq i} \mathbb{E}\left[\frac{\partial f}{\partial \theta_j} v_i v_j\right]$$
$$= \frac{\partial f}{\partial \theta_i} \mathbb{E}\left[v_i^2\right] + \sum_{j \neq i} \frac{\partial f}{\partial \theta_j} \mathbb{E}\left[v_i v_j\right] \tag{4}$$

The first expected value in Eq. (4) is of a squared random variable and all expectations in the summation term are of two independent and identically distributed random variables multiplied.

Lemma 1. *The expected value of a random variable v squared $\mathbb{E}[v^2] = 1$ when $\mathbb{E}[v] = 0$ and $\mathrm{Var}[v] = 1$.*

Proof. Variance is $\mathrm{Var}[v] = \mathbb{E}\left[(v - \mathbb{E}[v])^2\right] = \mathbb{E}\left[v^2\right] - \mathbb{E}[v]^2$. Rearranging and substituting $\mathbb{E}[v] = 0$ and $\mathrm{Var}[v] = 1$, we get $\mathbb{E}\left[v^2\right] = \mathbb{E}[v]^2 + \mathrm{Var}[v] = 0 + 1 = 1$. □

Lemma 2. *The expected value of two i.i.d. random variables multiplied $\mathbb{E}[v_i v_j] = 0$ when $\mathbb{E}[v_i] = 0$ or $\mathbb{E}[v_j] = 0$.*

Proof. For i.i.d. v_i and v_j the expected value $\mathbb{E}[v_i v_j] = \mathbb{E}[v_i]\mathbb{E}[v_j] = 0$ when $\mathbb{E}[v_i] = 0$ or $\mathbb{E}[v_j] = 0$. □

Using Lemmas 1 and 2, Eq. (4) reduces to

$$\mathbb{E}\left[g_i(\boldsymbol{\theta})\right] = \frac{\partial f}{\partial \theta_i} \tag{5}$$

and therefore

$$\mathbb{E}\left[\boldsymbol{g}(\boldsymbol{\theta})\right] = \nabla f(\boldsymbol{\theta}) \ . \tag{6}$$

□

3.3. Forward Gradient Descent

We construct a forward gradient descent (FGD) algorithm by replacing the gradient ∇f in standard GD with the forward gradient $\boldsymbol{g}$ (Algorithm 1). In practice we use a mini-batch stochastic version of this where f_t changes per iteration as it depends on each mini-batch of data used during training. We note that the directional derivative d_t in Algorithm 1 can have positive or negative sign. When the sign is negative, the forward gradient $\boldsymbol{g}_t$ corresponds to backtracking from the direction of $\boldsymbol{v}_t$, or reversing the direction to point towards the true gradient in expectation. Figure 1 shows two v_k samples exemplifying this behavior.

In this paper we limit our scope to FGD to clearly study this fundamental algorithm and compare it to standard backpropagation, without confounding factors such as momentum or adaptive learning rate schemes. We believe that extensions of the method to other families of gradient-based optimization algorithms are possible.

Algorithm 1 Forward gradient descent (FGD)

Require: η: learning rate
Require: f: objective function
Require: $\boldsymbol{\theta}_0$: initial parameter vector
  $t \leftarrow 0$ ▷ Initialize
  while $\boldsymbol{\theta}_t$ not converged **do**
    $t \leftarrow t + 1$
    $\boldsymbol{v}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ ▷ Sample perturbation
    Note: *the following computes f_t and d_t simultaneously and without having to compute ∇f in the process*
    $f_t,\ d_t \leftarrow f(\boldsymbol{\theta}_t),\ \nabla f(\boldsymbol{\theta}_t) \cdot \boldsymbol{v}$ ▷ Forward AD (Section 3.1)
    $\boldsymbol{g}_t \leftarrow \boldsymbol{v}_t d_t$ ▷ Forward gradient
    $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta\, \boldsymbol{g}_t$ ▷ Parameter update
  end while
  return $\boldsymbol{\theta}_t$

3.4. Choice of Direction Distribution

As shown by the proof in Section 3.2, the multivariate distribution $p(\boldsymbol{v})$ from which direction vectors $\boldsymbol{v}$ are sampled must have two properties: (1) the components must be independent from each other (e.g., a diagonal Gaussian) and (2) the components must have zero mean and unit variance.

In our experiments we use the multivariate standard normal as the direction distribution $p(\boldsymbol{v})$ so that $\boldsymbol{v} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, that is, $v_i \sim \mathcal{N}(0, 1)$ are independent for all i. We leave exploring other admissible distributions for future work.

4. Related Work

The idea of performing optimization by the use of random perturbations, thus avoiding adjoint computations, is the intuition behind a variety of approaches, including simulated annealing (Kirkpatrick et al., 1983), stochastic approximation (Spall et al., 1992), stochastic convex optimization (Nesterov & Spokoiny, 2017; Dvurechensky et al., 2021), and correlation-based learning methods (Barto et al., 1983), which lend themselves to efficient hardware implementation (Alspector et al., 1988). Our work here falls in the general class of so-called weight perturbation methods; see Pearlmutter (1994, §4.4) for an overview along with a description of a method for efficiently gathering second-order information during the perturbative process, which suggests that accelerated second-order variants of the present method may be feasible. Note that our method is novel in avoiding the truncation error of previous weight perturbation approaches by using AD rather than small but finite perturbations, thus completely avoiding the method of divided differences and its associated numeric issues.

In neural network literature, alternatives to backpropagation proposed include target propagation (LeCun, 1986; 1987; Bengio, 2014; 2020; Meulemans et al., 2020), a technique that propagates target values rather than gradients backwards between layers. For recurrent neural networks (RNNs), vari-

ous approaches to the online credit assignment problem have features in common with forward mode AD (Pearlmutter, 1995). An early example is the real-time recurrent learning (RTRL) algorithm (Williams & Zipser, 1989) which accumulates local sensitivities in an RNN during forward execution, in a manner similar to forward AD. A very recent example in the RTRL area is an anonymous submission we identified at the time of drafting this manuscript, where the authors are using directional derivatives to improve several gradient estimators, e.g., synthetic gradients (Jaderberg et al., 2017), first-order meta-learning (Nichol et al., 2018), as applied to RNNs (Anonymous, 2022).

Coordinate descent (CD) algorithms (Wright, 2015) have a structure where in each optimization iteration only a single component $\frac{\partial f}{\partial \theta_i}$ of the gradient ∇f is used compute an update. Nesterov (2012) provides an extension of CD called random coordinate descent (RCD), based on coordinate directional derivatives, where the directions are constrained to randomly chosen coordinate axes in the function's domain as opposed to arbitrary directions we use in our method. A recent use of RCD is by Ding & Li (2021) in Langevin Monte Carlo sampling, where the authors report no computational gain as the RCD needs to be run multiple times per iteration in order to achieve a preset error tolerance. The SEGA (SkEtched GrAdient) method by Hanzely et al. (2018) is based on gradient estimation via random linear transformations of the gradient that is called a "sketch" computed using finite differences. Jacobian sketching by Gower et al. (2018b) is designed to provide good estimates of the Jacobian, in a manner similar to how quasi-Newton methods update Hessian estimates (Gower et al., 2018a).

Lastly there are other, and more distantly related, approaches concerning gradient estimation such as synthetic gradients (Jaderberg et al., 2017), motivated by a need to break the sequential forward–backward structure of backpropagation, and Monte Carlo gradient estimation (Mohamed et al., 2020), where the gradient of an expectation of a function is computed with respect to the parameters defining the distribution that is integrated.

For a review of the origins of reverse mode AD and backpropagation, we refer the interested readers to Schmidhuber (2020) and Griewank (2012).

5. Experiments

We implement forward AD in PyTorch to perform the experiments (details given in Section 6). In all experiments, except in Section 5.1, we use learning rate decay with $\eta_i = \eta_0\, e^{-i\,k}$, where η_i is the learning rate at iteration i, η_0 is the initial learning rate, and $k = 10^{-4}$. In all experiments we run forward gradients and backpropagation for an *equal number of iterations*. We run the code with CUDA on a Nvidia Titan XP GPU and use a minibatch size of 64.

First we look at test functions for optimization, and compare the behavior of forward gradient and backpropagation in the $\mathbb{R}^2$ space where we can plot and follow optimization trajectories. We then share results of experiments with training ML architectures of increasing complexity. We measured no practical difference in memory usage between the two methods (less than 0.1% difference in each experiment).

5.1. Optimization Trajectories of Test Functions

In Figure 2 we show the results of experiments with

- the Beale function, $f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$
- and the Rosenbrock function, $f(x, y) = (a-x)^2 + b(y - x^2)^2$, where $a = 1, b = 100$.

Note that forward gradient and backpropagation have roughly similar time complexity in these cases, forward gradient being slightly faster per iteration. Crucially, we see that forward gradient steps behave the same way as backpropagation in expectation, as seen in loss per iteration (leftmost) and optimization trajectory (rightmost) plots.

5.2. Empirical Measures of Complexity

In order to compare the two algorithms applied to ML problems in the rest of this section, we use several measures.

For runtime comparison we use the R_f and R_b factors defined in Section 2.3. In order to compute these factors, we measure runtime(f) as the time it takes to run a given architecture with a sample minibatch of data and compute the loss, without performing any derivative computation and parameter update. Note that in the measurements of R_f and R_b, the time taken by gradient descent (parameter updates) are included, in addition to the time spent in computing the derivatives. We also introduce the ratio R_f/R_b as a measure of the runtime cost of the forward gradient relative to the cost of backpropagation in a given architecture.

In order to compare loss performance, we define T_b as the time at which the lowest validation loss is achieved by backpropagation (averaged over runs). T_f is the time the same validation loss is achieved by the forward gradient for the same architecture. The T_f/T_b ratio gives us a measure of the time it takes for the forward mode to achieve the minimum validation loss relative to the time taken by backpropagation.

5.3. Logistic Regression

Figure 3 gives the results of several runs of multinomial logistic regression for MNIST digit classification. We observe that the runtime cost of the forward gradient and backpropagation relative to the base runtime are $R_f = 2.435$ and

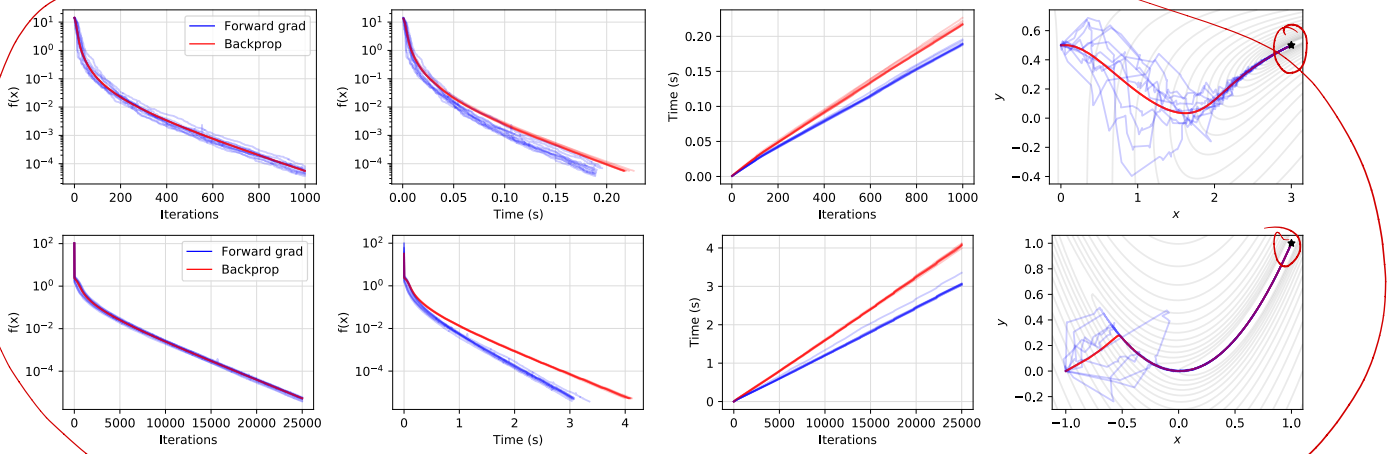


Figure 2. Comparison of forward gradient and backpropagation in test functions, showing ten independent runs. Top row: Beale function, learning rate 0.01. Bottom row: Rosenbrock function. Learning rate 5×10^{-4}. Rightmost column: Optimization trajectories in each function's domain, shown over contour plots of the functions. Star symbol marks the global minimum in the contour plots.

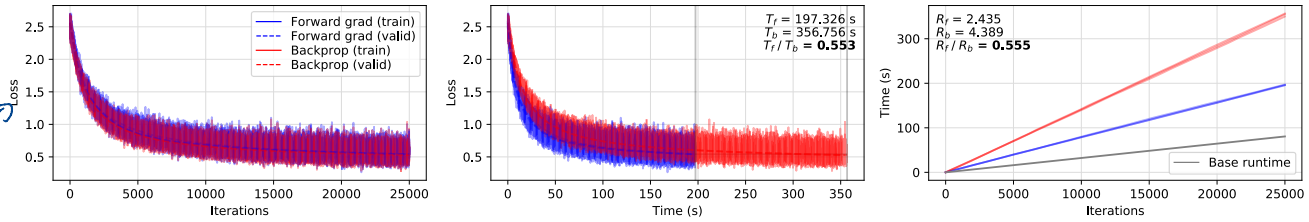


Figure 3. Comparison of forward gradient and backpropagation in logistic regression, showing five independent runs. Learning rate 10^{-4}.

$R_b = 4.389$, which are compatible with what one would expect from a typical AD system (Section 2.3). The ratios $R_f/R_b = 0.555$ and $T_f/T_b = 0.553$ indicate that the forward gradient is roughly twice as fast as backpropagation in both runtime and loss performance. In this simple problem these ratios coincide as both techniques have nearly identical behavior in the loss per iteration space, meaning that the runtime benefit is reflected almost directly in the loss per time space. In more complex models in the following subsections we will see that the relative loss and runtime ratios can be different in practice.

5.4. Multi-Layer Neural Networks

Figure 4 shows two experiments with a multi-layer neural network (NN) for MNIST classification with different learning rates. The architecture we use has three fully-connected layers of size 1024, 1024, 10, with ReLU activation after the first two layers. In this model architecture, we observe the runtime costs of the forward gradient and backpropagation relative to the base runtime as $R_f = 2.468$ and $R_b = 4.165$, and the relative measure $R_f/R_b = 0.592$ on average. These are roughly the same with the logistic regression case.

The top row (learning rate 2×10^{-5}) shows a result where forward gradient and backpropagation behave nearly identical in loss per iteration (leftmost plot), resulting in a T_f/T_b ratio close to R_f/R_b. We show this result to communicate an example where the behavior is similar to the one we observed for logistic regression, where the loss per iteration behavior between the techniques are roughly the same and the runtime benefit is the main contributing factor in the loss per time behavior (second plot from the left).

Interestingly, in the second experiment (learning rate 2×10^{-4}) we see that forward gradient achieves faster descent in the loss per iteration plot. We believe that this behavior is due to the different nature of stochasticity between the regular SGD (backpropagation) and the forward SGD algorithms, and we speculate that the noise introduced by forward gradients might be beneficial in exploring the loss surface. When we look at the loss per time plot, which also incorporates the favorable runtime of the forward mode, we see a loss performance metric T_f/T_b value of 0.211, representing a case that is more than four times as fast as backpropagation in achieving the reference validation loss.

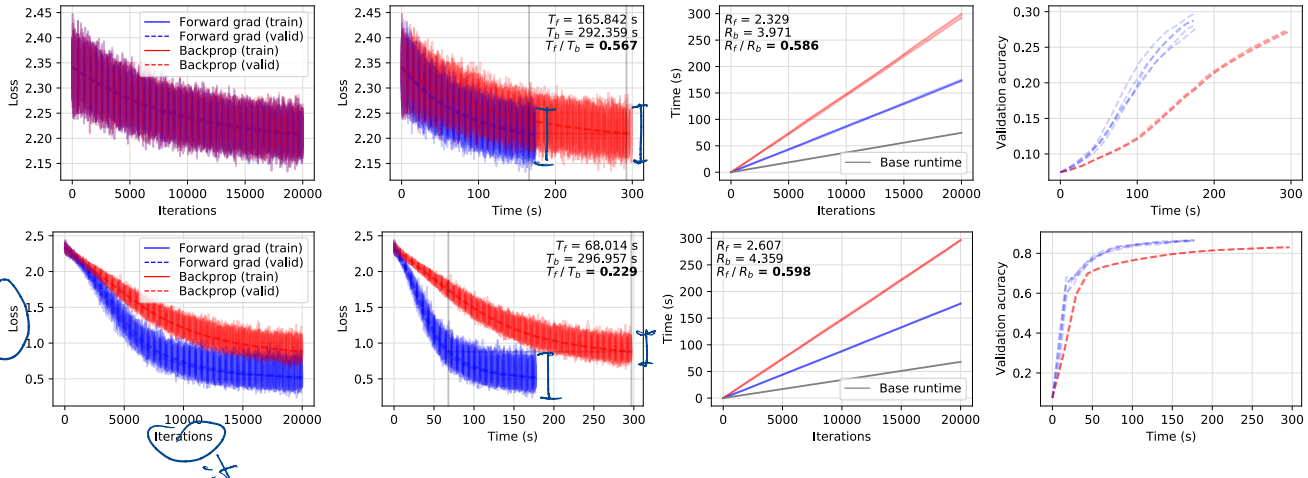


Figure 4. Comparison of forward gradient and backpropagation for the multi-layer NN, showing two learning rates. Top row: learning rate 2×10^{-5}. Bottom row: learning rate 2×10^{-4}. Showing five independent runs per experiment.

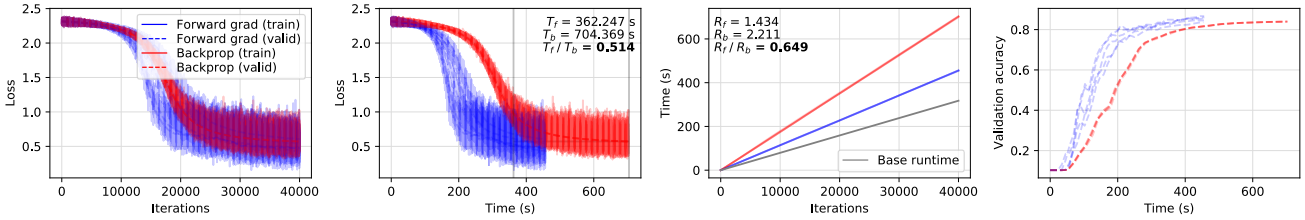


Figure 5. Comparison of forward gradient and backpropagation for the CNN. Learning rate 2×10^{-4}. Showing five independent runs.

5.5. Convolutional Neural Networks

In Figure 5 we show a comparison between the forward gradient and backpropagation for a convolutional neural network (CNN) for the same MNIST classification task. The CNN has four convolutional layers with 3×3 kernels and 64 channels, followed by two linear layers of sizes 1024 and 10. All convolutions and the first linear layer are followed by ReLU activation and there are two max-pooling layers with 2×2 kernel after the second and fourth convolutions.

In this architecture we observe the best forward AD performance with respect to the base runtime, where the forward mode has $R_f = 1.434$ representing an overhead of only 43% on top of the base runtime. Backpropagation with $R_b = 2.211$ is very close to the ideal case one can expect from a reverse AD system, taking roughly double the time. $R_f/R_b = 0.649$ represents a significant benefit for the forward AD runtime with respect to backpropagation. In loss space, we get a ratio $T_f/T_b = 0.514$ which shows that forward gradients are close to twice as fast as backpropagation in achieving the reference level of validation loss.

5.6. Scalability

The results in the previous subsections demonstrate that

- training without backpropagation can feasibly work within a typical ML training pipeline and do so in a computationally competitive way, and
- forward AD can even beat backpropagation in loss decrease per training time for the same choice of hyperparameters (learning rate and learning rate decay).

In order to investigate whether these results will scale to larger NNs with more layers, we measure runtime cost and memory usage as a function of NN size. In Figure 6 we show the results for the MLP architecture (Section 5.4), where we run experiments with an increasing number of layers in the range $[1, 100]$. The linear layers are of size 1,024, with no bias. We use a mini-batch size 64 as before.

Looking at the cost relative to the base runtime, which also changes as a function of the number of layers, we see that backpropagation remains within $R_b \in [4, 5]$ and forward gradient remains within $R_f \in [3, 4]$ for a large proportion of the experiments. We also observe that forward gradients remain favorable for the whole range of layer sizes considered, with the R_f/R_b ratio staying below 0.6 up to ten layers and going slightly over 0.8 at 100 layers. Importantly, there is virtually no difference in memory consumption between the two methods.

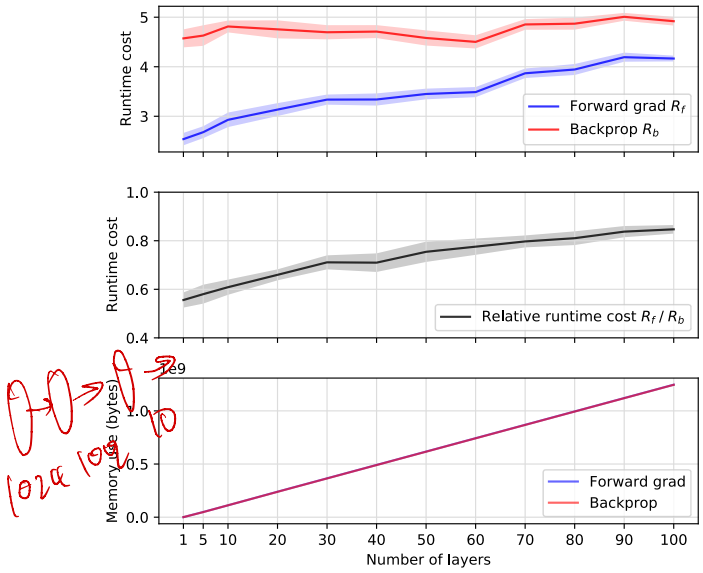


Figure 6. Comparison of how the runtime cost and memory usage of forward gradients and backpropagation scale as a function NN depth for the MLP architecture where each layer is of size 1024. Showing mean and standard deviation over ten independent runs.

6. Implementation

We implement a forward-mode AD system in Python and base this on PyTorch tensors in order to enable a fair comparison with a typical backpropagation pipeline in PyTorch, which is widely used by the ML community.[9] We release our implementation publicly.[10]

Our forward-mode AD engine is implemented from scratch using operator overloading and non-differentiable PyTorch tensors (`requires_grad=False`) as a building block. This means that our forward AD implementation does not use PyTorch's reverse-mode implementation (called "autograd") and computation graph. We produce the backpropagation results in experiments using PyTorch's existing reverse-mode code (`requires_grad=True` and `.backward()`) as usual.

Note that empirical comparisons of the relative runtimes of forward- and reverse-mode AD are highly dependent on the implementation details in a given system and would show differences across different code bases. When implementing the forward mode of tensor operations common in ML (e.g., matrix multiplication, convolutions), we identified opportunities to the make forward AD operations even more efficient (e.g., stacking channels of primal and derivative parts of tensors in a convolution). Note that the implementation we use in this paper does not currently have these. We expect the forward gradient performance to improve even further as high-quality forward-mode implementations find their way into mainstream ML libraries and get tightly integrated into tensor code.

Another implementation approach that can enable a straightforward application of forward gradients to existing code can be based on the complex-step method (Martins et al., 2003), a technique that can approximate directional derivatives with nothing but basic support for complex numbers.

[9] We also experimented with the forward mode implementation in JAX (Bradbury et al., 2018) but decided to base our implementation on PyTorch due to its maturity and simplicity allowing us to perform a clear comparison.

[10] To be shared in the upcoming revision.

7. Conclusions

We have shown that a typical ML training pipeline can be constructed without backpropagation, using only forward AD, while still being computationally competitive. We expect this contribution to find use in distributed ML training, which is outside the scope of this paper. Furthermore, the runtime results we obtained with our forward AD prototype in PyTorch are encouraging and we are cautiously optimistic that they might be the first step towards significantly decreasing the time taken to train ML architectures, or alternatively, enabling the training of more complex architectures with a given compute budget. We are excited to have the results confirmed and studied further by the research community.

The work presented here is the basis for several directions that we would like to follow. In particular, we are interested in working on gradient descent algorithms other than SGD, such as SGD with momentum, and adaptive learning rate algorithms such as Adam (Kingma & Ba, 2015). In this paper we deliberately excluded these to focus on the most isolated and clear case of SGD, in order to establish the technique and a baseline. We are also interested in experimenting with other ML architectures. The components used in our experiments (i.e., linear and convolutional layers, pooling, ReLU nonlinearity) are representative of the building blocks of many current architectures in practice, and we expect the results to apply to these as well.

Lastly, in the longer term we are interested in seeing whether the forward gradient algorithm can contribute to the mathematical understanding of the biological learning mechanisms in the brain, as backpropagation has been historically viewed as biologically implausible as it requires a precise backward connectivity (Bengio et al., 2015; Lillicrap et al., 2016; 2020). In this context, one way to look at the role of the directional derivative in forward gradients is to interpret it as the feedback of a single global scalar quantity that is identical for all the computation nodes in the network.

We believe that forward AD has computational characteristics that are ripe for exploration by the ML community, and we expect that its addition to the conventional ML infrastructure will lead to major breakthroughs and new approaches.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

Alspector, J., Allen, R., Hu, V., and Satyanarayana, S. Stochastic learning networks and their electronic implementation. In Anderson, D. (ed.), *Neural Information Processing Systems*. American Institute of Physics, 1988. URL https://proceedings.neurips.cc/paper/1987/file/f033ab37c30201f73f142449d037028d-Paper.pdf.

Anonymous. Learning by directional gradient descent. In *Submitted to The Tenth International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=5i7lJLuhTm. Under Review.

Barto, A. G., Sutton, R. S., and Anderson, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. doi: 10.1109/TSMC.1983.6313077.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research (JMLR)*, 18(153):1–43, 2018. URL http://jmlr.org/papers/v18/17-468.html.

Bengio, Y. How auto-encoders could provide credit assignment in deep networks via target propagation. *arXiv preprint arXiv:1407.7906*, 2014.

Bengio, Y. Deriving differential target propagation from iterating approximate inverses. *arXiv preprint arXiv:2007.15139*, 2020.

Bengio, Y., Lee, D.-H., Bornschein, J., Mesnard, T., and Lin, Z. Towards biologically plausible deep learning. *arXiv preprint arXiv:1502.04156*, 2015.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Deisenroth, M. P., Faisal, A. A., and Ong, C. S. *Mathematics for Machine Learning*. Cambridge University Press, 2020.

Ding, Z. and Li, Q. Langevin Monte Carlo: random coordinate descent and variance reduction. *Journal of Machine Learning Research*, 22(205):1–51, 2021.

Dvurechensky, P., Gorbunov, E., and Gasnikov, A. An accelerated directional derivative method for smooth stochastic convex optimization. *European Journal of Operational Research*, 290(2):601–621, 2021.

Gebremedhin, A. H., Manne, F., and Pothen, A. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705, 2005.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Gower, R., Hanzely, F., Richtarik, P., and Stich, S. U. Accelerated stochastic matrix inversion: General theory and speeding up bfgs rules for faster second-order optimization. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018a.

Gower, R. M., Richtárik, P., and Bach, F. Stochastic quasi-gradient methods: Variance reduction via Jacobian sketching. *arXiv preprint arXiv:1805.02632*, 2018b.

Griewank, A. Who invented the reverse mode of differentiation? *Documenta Mathematica*, Extra Volume ISMP: 389–400, 2012.

Griewank, A. and Walther, A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.

Hanzely, F., Mishchenko, K., and Richtárik, P. SEGA: Variance reduction via gradient sketching. *arXiv preprint arXiv:1809.03054*, 2018.

Hascoët, L. Adjoints by automatic differentiation. *Advanced Data Assimilation for Geosciences: Lecture Notes of the Les Houches School of Physics: Special Issue, June 2012*, (2012):349, 2014.

Jaderberg, M., Czarnecki, W. M., Osindero, S., Vinyals, O., Graves, A., Silver, D., and Kavukcuoglu, K. Decoupled neural interfaces using synthetic gradients. In *International Conference on Machine Learning*, pp. 1627–1635. PMLR, 2017.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. Optimization by simulated annealing. *Science*, 220(4598): 671–680, 1983.

LeCun, Y. Learning process in an asymmetric threshold network. In *Disordered Systems and Biological Organization*, pp. 233–240. Springer, 1986.

LeCun, Y. *PhD thesis: Modeles connexionnistes de l'apprentissage (connectionist learning models)*. Universite P. et M. Curie (Paris 6), June 1987.

Lillicrap, T. P., Cownden, D., Tweed, D. B., and Akerman, C. J. Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, 7(1):1–10, 2016.

Lillicrap, T. P., Santoro, A., Marris, L., Akerman, C. J., and Hinton, G. Backpropagation and the brain. *Nature Reviews Neuroscience*, 21(6):335–346, 2020.

Linnainmaa, S. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, pp. 6–7, 1970.

Martins, J. R., Sturdza, P., and Alonso, J. J. The complex-step derivative approximation. *ACM Transactions on Mathematical Software (TOMS)*, 29(3):245–262, 2003.

Meulemans, A., Carzaniga, F., Suykens, J., Sacramento, J., and Grewe, B. F. A theoretical framework for target propagation. *Advances in Neural Information Processing Systems*, 33:20024–20036, 2020.

Mohamed, S., Rosca, M., Figurnov, M., and Mnih, A. Monte Carlo gradient estimation in machine learning. *Journal of Machine Learning Research*, 21(132):1–62, 2020.

Nesterov, Y. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22(2):341–362, 2012.

Nesterov, Y. and Spokoiny, V. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, 17(2):527–566, 2017.

Nichol, A., Achiam, J., and Schulman, J. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*, 2018.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8026–8037, 2019.

Pearlmutter, B. A. Fast exact multiplication by the Hessian. *Neural Computation*, 6(1):147–60, 1994. doi: 10.1162/neco.1994.6.1.147.

Pearlmutter, B. A. Gradient calculations for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural Networks*, 6(5):1212–1228, 1995.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

Schmidhuber, J. Who invented backpropagation? 2020. URL https://people.idsia.ch/~juergen/who-invented-backpropagation.html.

Siskind, J. M. and Pearlmutter, B. A. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software*, 33(4-6):1288–1330, 2018.

Spall, J. C. et al. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE Transactions on Automatic Control*, 37(3):332–341, 1992.

Wengert, R. E. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

Williams, R. J. and Zipser, D. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

Wright, S. J. Coordinate descent algorithms. *Mathematical Programming*, 151(1):3–34, 2015.