Studies on Forward Mode Gradient Descent

Yuxing Yao
Department of Computer Science, EPFL, Switzerland

***Abstract*—In machine learning, backpropagation is the most common method for updating model parameters when training the model. It computes the exact gradient for each parameter through one forward pass followed by one backward path. A recent work [1] introduces an alternative way that implements a forward-mode automatic differentiation algorithm to compute unbiased estimates of the gradients of all parameters of the model. This work is based on the work [1]. First, we implement the traditional backpropagation method and the proposed forward-mode method from scratch using only tensor operations of PyTorch so that we can control the level of optimization to produce a meaningful comparison of their running times and memory usage. The result shows that the forward method can reduce memory consumption compared to backpropagation, and the reduction is greater as the batch size increases. But there is no significant difference between the two in terms of running time (This is not consistent with the conclusion of the original paper regarding the running time). Secondly, we note that the original method of forward-mode, while producing unbiased estimates, does not guarantee small variance, which leads to the fact that models trained using the forward-mode method can only be trained using smaller learning rates. To address this issue, we propose various methods to reduce the variance, which allow the model to use a larger learning rate or allow the model to achieve the performance that can only be achieved when the learning rate is large in traditional backpropagation even when the learning rate is very small.**

I. Introduction

Backpropagation-based gradient descent [2] is the dominant model training method used by the machine learning community today. This method is also known as the reverse-mode automatic differentiation. There are many mature platforms and libraries such as PyTorch [3] and TensorFlow [4] that can support this approach very well. However, The main method that this work is based on is forward-mode automatic differentiation [5]. The difference between these two modes is demonstrated as follows:

Reverse-mode autodiff The function that a ML model represents is $f : \mathbb{R}^n \to \mathbb{R}^1$, here we want to compute the gradients of all parameters instead of the inputs of the model, so n here is the number of parameters in the model instead of the number of input dimension. The set of parameters is $\theta \in \mathbb{R}^n$, and the vector of ad-joints is $v \in \mathbb{R}^1$, the reverse-mode computes $f(\theta)$ and $v^T J_f(\theta)$ which is also called the Vector-Jacobian Product, where $J_f \in \mathbb{R}^{1\times n}$ through one forward pass and one backward pass. Because v here is a scalar (The output dimension is one because of the loss function), this Vector-Jacobian Product is the true gradients of all n inputs $\nabla f(\theta) = [\frac{\partial f}{\partial \theta_1}, ..., \frac{\partial f}{\partial \theta_n}]$

Forward-mode autodiff The setting we use here are basically the same as those above, except that $v \in \mathbb{R}^n$, The forward mode computes $f(\theta)$ and $J_f(\theta)v$ which is also called the Jacobian-Vector Product, where $J_f \in \mathbb{R}^{1\times n}$ through one forward pass only. If we choose to set v to be 0 at all positions except for one position which is 1, then this forward pass produces the exact value of partial derivative for that specific parameter.

Among all the automatic differentiation modes, the reverse-mode is the most widely used in the field of machine learning because it can accurately compute the derivatives of all parameters of the model by performing one forward pass and one backward pass. The reason that the forward-mode method is not traditionally used is although it's computationally cheaper and easier to implement, it can only produce one scalar after each forward pass. If this scalar is the value of gradient for one specific parameter in a Neural Network, then n forward passes are needed to update all the parameters in a Network that has n different parameters, which is not a desired performance.

In paper [1], Baydin et al. proposed that if $v \in \mathbb{R}^n$ is a multi-variant random variable $v \sim p(v)$ such that all elements v_i in v are independent and have zero mean and one standard deviation, then we can multiply the scalar directional derivative $\nabla f(\theta)v$ which is produced by one forward pass with vector v to compute an unbiased estimation of the true gradients of the model's parameters that can be computed by backpropagation ("forward-mode method' is used below to refer to this method). However, this algorithm does not guarantee the variance to be small enough of this unbiased estimate, which leads to the models trained using this method can only use a small learning rate, limiting its usefulness in practical applications. This is the problem that this work focuses on.

II. Contributions of this work

The main content of this work can be divided into three areas.

1) Propose three methods to decrease the variance when using forward-mode method, so that we can use larger learning rate when the model is trained with forward-mode method.
2) Propose a 'momentum method' to enable the model trained using forward-mode method which has a small learning rate to converge faster, so that the forward-mode method can perform similarly to the backpropagation method, which can use a larger learning rate, even though the variance of the estimation is not reduced.
3) Implement both forward-mode method and reverse-mode method from scratch, using only PyTorch.tensor without any optimization from Autograd, which allows

us to control the degree of optimization for both methods to produce meaningful comparisons on time and memory consumption.

III. Method

A. Naive version Forward-mode method

Given a model which has a loss function that generates a scalar as a loss value, this model can be expressed as a function $f : \mathbb{R}^n \to \mathbb{R}^1$. Then we can define the "true gradient" $g : \mathbb{R}^n \to \mathbb{R}^n$ and the "forward gradient" $\hat{g} : \mathbb{R}^n \to \mathbb{R}^n$ to be:

$$g(\theta) = \nabla f(\theta) \tag{1}$$

$$\hat{g}(\theta) = (\nabla f(\theta) \cdot v)v \tag{2}$$

1) Unbiasedness: The content of this subsection is completely the same as the proof presented in paper [1]. The Jacobian-Vector product can be written out as follows:

$$\begin{aligned} \nabla f(\theta) \cdot v &= \sum_i \frac{\partial f}{\partial \theta_i} v_i \\ &= \frac{\partial f}{\partial \theta_1} v_1 + \frac{\partial f}{\partial \theta_2} v_2 + \cdots + \frac{\partial f}{\partial \theta_n} v_n \end{aligned} \tag{3}$$

Combining the Eq. (2) and the Eq. (3), we can get the value of the i^{th} element of $\hat{g}(\theta)$ which is

$$\hat{g}_i(\theta) = \frac{\partial f}{\partial \theta_i} v_i^2 + \sum_{j \neq i} \frac{\partial f}{\partial \theta_j} v_i v_j \tag{4}$$

The expected value of Eq. (4) is

$$\begin{aligned} \mathbb{E}[\hat{g}_i(\theta)] &= \mathbb{E}[\frac{\partial f}{\partial \theta_i} v_i^2 + \sum_{j \neq i} \frac{\partial f}{\partial \theta_j} v_i v_j] \\ &= \frac{\partial f}{\partial \theta_i} \mathbb{E}[v_i^2] + \sum_{j \neq i} \frac{\partial f}{\partial \theta_j} \mathbb{E}[v_i v_j] \\ &= \frac{\partial f}{\partial \theta_i} = g_i(\theta) \end{aligned} \tag{5}$$

We get Eq. (5) based on the fact that $\mathbb{E}[v_i] = 0$, $\text{Var}[v_i] = 1, \forall i \in [0, n)$ and v_i, v_j are i.i.d. when $i \neq j$. This shows that $\hat{g}(\theta)$ is an unbiased estimate of $g(\theta)$.

2) Variance: The variance of the estimate is not considered in the original paper. But this is very important, because of the nature of the forward-mode method, the errors will keep passing cumulatively to the later iterations. If the errors are large, the directional derivatives obtained by this method will be very unreliable, and if such unreliable directional derivative is then multiplied with random vector v, the updated values of some parameters will be far from the true values, which will cause irreversible errors.

For example, if the task is about classification, and the loss function is a cross-entropy loss connected to a Softmax layer, then if after one update, one bias value of all the bias values of the output layer becomes extremely large can lead to a very large output for this very node of the output layer. Then after Softmax operation, there can be Nan value and 0 value produced for this very node and all other nodes respectively. This will cause the loss function to produce Nan value, this way the training and updating of the model can no longer continue. So we need to measure the level of variance of the estimates.

$$\begin{aligned} \text{Var}[\hat{g}_i(\theta)] &= \mathbb{E}[\hat{g}_i(\theta)] - \mathbb{E}^2[\hat{g}_i(\theta)] \\ &= \mathbb{E}[(v_i \sum_j g_j(\theta) v_j)^2] - g_i^2(\theta) \\ &= \mathbb{E}[(\sum_j g_j(\theta) v_j)^2] - g_i^2(\theta) \\ &= \sum_{j \neq i} g_j^2(\theta) \end{aligned} \tag{6}$$

The value of this sum is of degree $\mathcal{O}(n)$, where n is the number of parameters of the model.

B. Forward-mode with multiple directions

If we generate multiple, instead of one v, and calculate the corresponding multiple directional derivatives from these multiple v, and then update the parameters of the model by averaging them, this will effectively reduce the variance of the final unbiased estimate. Let's assume that the estimates of the gradient value of the i^{th} parameter by k^{th} random vector is $\hat{g}_i^k(\theta)$. There are K random vectors.

$$\begin{aligned} \text{Var}[\hat{g}_i^{avg}(\theta)] &= \text{Var}[\frac{\hat{g}_i^1(\theta) + \hat{g}_i^2(\theta) + \cdots + \hat{g}_i^K(\theta)}{K}] \\ &= \frac{1}{K^2} K \text{Var}[\hat{g}_i(\theta)] \\ &= \frac{\sum_{j \neq i} g_i^2(\theta)}{K} \end{aligned} \tag{7}$$

We can assume the expectation of $g_i(\theta) = \sigma, \forall i \in [0, n)$, then

$$\text{Var}[\hat{g}_i^{avg}(\theta)] = \frac{n-1}{K} \sigma^2 \tag{8}$$

The value of Eq. (8) is of degree $\mathcal{O}(n)$, where n is the number of parameters of the model. But compared to the naive method mentioned earlier, this can reduce the variance by a factor of K where K is the number of random vectors.

C. Forward-mode with direction partition

Instead of generating multiple random vectors independently, we can generate multiple vectors by partitioning one source random vector into multiple parts and padding with zeros. The gradient estimate is then obtained by the sum of estimates from different parts. Here is an example of the partition process:

$$\begin{aligned} v &= [v_1, v_2, v_3, v_4, v_5, \cdots, v_{n-2}, v_{n-1}, v_n] \\ v^1 &= [v_1, v_2, 0, 0, 0 \cdots, 0, 0, 0] \\ v^2 &= [0, 0, v_3, v_4, 0, \cdots, 0, 0, 0] \\ &\vdots \\ v^K &= [0, 0, 0, 0, 0, \cdots, 0, v_{n-1}, v_n] \end{aligned}$$

If a random vector of length n is partitioned into K different parts and each part is padded with zero, then each partition has

$\frac{n}{K}$ non-padded values. $v^{p(i)}$ is the vector that has non-padded value at position i.

$$\begin{aligned}\text{Var}[\hat{g}_i^{sum}(\theta)] &= \mathbb{E}[(\sum_j g_j(\theta)v_j^{p(i)})^2] - g_i^2(\theta) \\ &= \sum_j \mathbb{E}[g_j^2(\theta)(v_j^{p(i)})^2] - g_i^2(\theta) \\ &= \frac{n}{K}\sigma^2 - \sigma^2 \\ &= \frac{n-K}{K}\sigma^2\end{aligned} \tag{9}$$

The value of Eq. (9) is of degree $\mathcal{O}(n)$, where n is the number of parameters of the model. This method can reduce the variance even further than simply taking the average value of multiple directional derivatives. Note that if we set the number of partitions to n, then this method is equal to computing the exact gradients for all parameters via n passes of forward-mode auto-differentiation.

D. Forward-mode with random vector generated with input

According to the calculation above, the variance of the estimates is of degree $\mathcal{O}(n)$, Generally, in a network, the fully connected layer has the most parameters, so if the variance of the estimates of the gradients of the parameters in the fully connected layer can be reduced, then this will have an important impact on reducing the variance of the estimates of the gradients of all the parameters in the network.

For a fully connected layer with $input\_dim \times output\_dim$ weight parameters, when we use traditional gradient with backpropagation, the true gradients of these parameters can be computed by:

$$G = \frac{uv^t}{\beta} \tag{10}$$

Where $u \in \mathbb{R}^{input\_dim \times batch\_size}$ is the input matrix of this layer,$v \in \mathbb{R}^{output\_dim \times batch\_size}$ is the Jacobian matrix from the next layer during the process of backpropagation. $batch\_size = \beta$

Unlike all the methods mentioned above, we don't generate a random vector of the same size as the size of parameters. Instead, we generate a random vector $n \in \mathbb{R}^{output\_dim \times 1}$. Then we use xn^t to generate the random vector we desired, where $x \in \mathbb{R}^{Input\_dim \times 1}$. x is obtained by randomly select a column of the Q matrix which is from the QR decomposition of the input matrix u.

1) Unbiasedness: First let's prove that this method can generate an unbiased estimates of the gradients. Let u^k represents the k^{th} column of u and u_i^k represents the i^{th} element of the k^{th} column of u

$$\begin{aligned}\hat{G} &= xn^T[\frac{uv^T}{\beta} \cdot (xn^T)] \\ &= \frac{1}{\beta}\sum_k (xn^T\langle u^k, x\rangle\langle v^k, n\rangle)\end{aligned} \tag{11}$$

Then we can use QR decomposition:

$$u = qr \tag{12}$$

Combine Eq. (11) and Eq. (12)

$$\begin{aligned}\hat{G} &= \frac{1}{\beta}\sum_k (xn^T\langle qr^k, x\rangle\langle v^k, n\rangle) \\ &= \frac{1}{\beta}\sum_{k,i} (r_i^k xn^T\langle q^i, x\rangle\langle v^k, n\rangle)\end{aligned} \tag{13}$$

Because $n \in \mathbb{R}^{output\_dim \times 1}$ is a multi-variant random variable $n \sim p(n)$ such that all elements n_i in n are independent and have zero mean and one standard deviation, we have:

$$\mathbb{E}[n^T\langle v^k, n\rangle] = v^{kT} \tag{14}$$

Now we assume that $x = q^c$, where q^c is a random column of q. Combine Eq. (13) and Eq. (14)

$$\begin{aligned}\mathbb{E}[\hat{G}] &= \mathbb{E}[\frac{1}{\beta}\sum_{k,i}(r_i^k xv^{kT}\langle q^i, x\rangle)] \\ &= \mathbb{E}[\frac{1}{\beta}\sum_{k,i}(r_i^k xv^{kT}\cos(q^i, x))]\end{aligned} \tag{15}$$

Because the columns of q are orthogonal to each other

$$\mathbb{E}[r_i^k x\cos(q^i, x)] = \frac{1}{\beta}r_i^k q^i \tag{16}$$

From which we can get

$$\mathbb{E}[\sum_i r_i^k x\cos(q^i, x)] = \frac{1}{\beta}qr^k \tag{17}$$

Combine Eq. (15) and Eq. (17), we get

$$\mathbb{E}[\hat{G}] = \frac{1}{\beta^2}\sum_k qr^k v^{kT} = \frac{G}{\beta} \tag{18}$$

So according to Eq. (18), we can multiply the final result with $batch\_size$ of the training data to obtain the unbiased estimate.

2) Approximation: QR decomposition of a large matrix is time-consuming, so we use an approximation to make this method feasible in practice by directly letting x be any column of u and divide by the norm of this column vector, the reason we use QR decomposition is that the columns of q are orthogonal to each other so we can get rid of the cos value in Eq. (16). But in high-dimensional space, any two vectors are almost orthogonal, so we can use this approximation to produce a similar result.

3) Variance: In this subsection, we will present why this method can decrease the variance of the estimates. Now We've got $G, \hat{G} \in \mathbb{R}^{input\_dim \times output\_dim}$, and we can assume $g(ij), \hat{g}(ij)$ to be the value of element at the $(i,j)^{th}$ position in G and $\hat{G}$ respectively.

$$\begin{aligned}\text{Var}[\hat{g}(ij)] &= \mathbb{E}[\hat{g}^2(ij)] - \mathbb{E}^2[\hat{g}(ij)] \\ &= \mathbb{E}[\hat{g}^2(ij)] - \frac{g^2(ij)}{\beta^2}\end{aligned} \tag{19}$$

The second term in Eq. (19) is very small compared to the first term, so we can safely ignore it and only analyze the first term.

$$\mathbb{E}[\hat{g}^2(ij)] = \mathbb{E}[(\sum_{(a,b)} g(ab)n_a x_b n_i x_j)^2] \tag{20}$$

Like squaring any polynomial, this equation is obtained by summing the following combined terms:

1) If the combined term has different a, then this term is

$$c = 2\mathbb{E}[g(a_1b_1)g(a_2b_2)n(a_1)n(a_2)n^2(i)x(b_1)x(b_2)x^2(j)] = 0 \tag{21}$$

because $\mathbb{E}[n(a_1)n(a_2)n^2(i)] = 0$ when $a_1 \neq a_2$

2) If the combined term has same a and different b

$$c = 2\mathbb{E}[g(ab_1)g(ab_2)n^2(a)n^2(i)x(b_1)x(b_2)x^2(j)] = 0 \tag{22}$$

Because we can assume that $g(ab_1)$ and $g(ab_2)$ are independent and have zero mean according to Fig. (1)

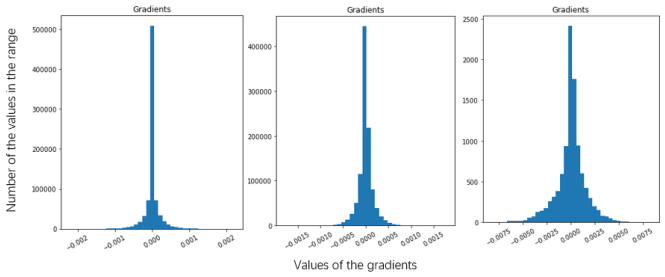

Fig. 1. The distribution of the true gradients in three consecutive fully connected layers; The mean of the values of true gradients is very close to 0.

3) If the combined term has same a and same b

$$c = 2\mathbb{E}[g^2(ab)n^2(a)n^2(i)x^(b)x^2(j)] = \begin{cases} \mathbb{E}[g^2(ab)x^2(b)x^2(j)] & a \neq i \\ 3\mathbb{E}[g^2(ab)x^2(b)x^2(j)] & a = i \end{cases} \tag{23}$$

There are $input\_size$ number of terms when $a = i$ and $(output\_size - 1) \times input\_size$ number of terms when $a \neq i$, so the number of terms is of degree $\mathcal{O}(input\_size \times output\_size)$. The value of each of these term is very small because $x = \frac{u^k}{||u^k||_2}$, and each element of x is a small value according to Fig. (2) So x_i^4 is very small.

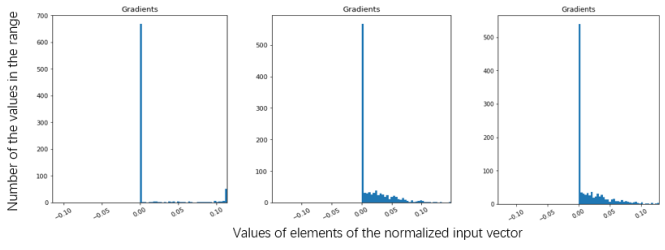

Fig. 2. The distribution of values in the vector $\frac{input}{||input||_2}$ for three consecutive fully connected layers when using ReLU as the Activation function.

So the overall expectation of the variance of $\hat{g}(ij)$ is very small compared to the value showed in Eq (.6)

E. The momentum Forward-mode method

In this method, we do not try to reduce the variance of the estimates, but rather generate random vectors in other ways to allow the model trained with the Forward-mode method to converge consistently and quickly even at small learning rates, allowing it to perform comparably to the method trained by backpropagation at larger learning rate. The idea of this method is inspired by [6], and based on the observation shown by Fig. (3).

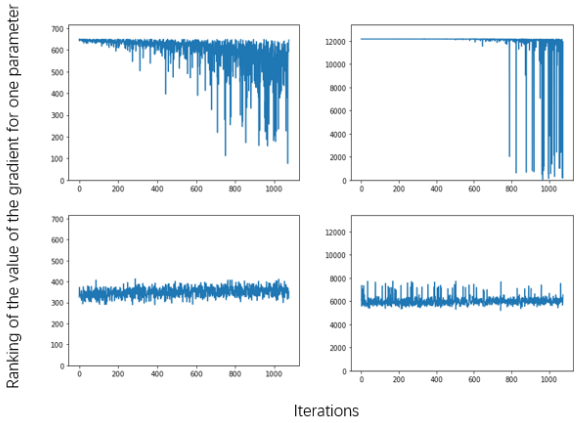

Fig. 3. Each image plots the change of the ranking of the value of gradient for a specific parameter among all the values of the gradients for all parameters along with the change of iterations. For example, If there are 100 parameters in a layer, then the parameter with the largest gradient in this layer is ranked 99 under this iteration, and the parameter with the smallest gradient is ranked 0 under this iteration. The first row of images shows the ranking change of the parameter with the largest gradient under the first iteration, and the second row of images shows the ranking change of the parameter with the middle gradient under the first iteration.

Algorithm 1 Momentum Vector Generation

Output: $\hat{v}$: a random vector used in Forward-mode method
Input: fix_gap, $shuffle\_ratio$, $decay\_rate$, cur_itr, $decrease\_gap$

if $cur\_itr \mod fix\_gap == 0$ **then**
  perform backpropagation to compute the gradients
  sort the indexes according to the value of gradients
  store the sorted index list
end if
if $cur\_itr \mod decrease\_gap == 0$ and $cur\_itr \neq 0$ **then**
  $decay\_rate \leftarrow decay\_rate * *2$
end if
$v_1 \leftarrow \mathbb{N}(0, 1)$
shuffle the sorted index list according to shuffle_ratio
sort v_1 according to the shuffled index list
$v_2 \leftarrow \mathbb{N}(0, 1)$
$\gamma \leftarrow decay\_rate^{cur\_itr \mod fix\_gap}$
$\hat{v} \leftarrow v_1 \cdot \gamma + v_2 \cdot (1 - \gamma)$
output $\hat{v}$

We can find that the direction of gradient descent tends to be more stable within a certain interval, the absolute value of the gradient in some directions is always at a larger value, and the gradient in some directions is always around 0. This means that some directions are more important than others in the gradient descent process. However, if the Forward-mode method is used, the information on the difference in importance of these directions is lost because the v used is a random vector. If we can get this information, we can speed up the training process that uses the forward-mode method so that it can learn and converge faster even at a smaller learning rate. So we propose an algorithm to generate a modified version of random vector v to use in the Forward-mode method, which is shown in Algorithm 1.

The main ideas of this algorithm are:

1) After every fixed time interval, we calculate the exact values of gradients for all parameters via one time of backpropagation. Then we sort the indexes of all parameters in the model according to the value of the gradient and record this sorted index list. We use this information to calibrate the generation of the random vectors used by following iterations by sorting the generated random vectors in the order of the list so that the direction of the sorted vectors is closer to the true gradient direction.
2) We also use a hyper-parameter called "shuffle_ratio", which is used to simulate small fluctuations in the direction of the gradient. Because although the direction of the gradient is relatively stable over a certain time range, fluctuations in the small range are inevitable and significant. Therefore, before sorting the generated random vectors by sorted index list, we will break up the segments of the sorted index list, and the ratio of each segment to the total length is the value of "shuffle_ratio".
3) "decay_rate" and "decrease_gap" are two hyper-parameters used to balance the effect of foreknown direction and randomness. As the training process moves away from the previous checkpoint and approaches the next checkpoint, the reference value of the gradient direction computed at the previous checkpoint becomes smaller and smaller, and we use decay_rate to sum the sorted vectors and the completely random vectors as weights to gradually increase the proportion of completely random vectors. And as the training progresses, the gradient direction becomes more and more unstable, and then we have to gradually reduce the weight of the sorting vector. decrease_gap determines after how many iterations we have to reduce the decay_rate to reduce the weight of the sorting vector.

It is shown in our experiments that this method allows the direction of the gradient vector computed by the forward-mode method to be closer to the real gradient vector direction and speeds up the convergence of the model's training process when the learning rate is small.

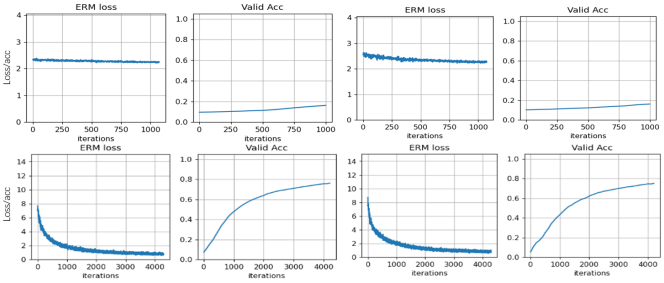

Fig. 4. (**First Row**) The learning process of MLP using learning rate 1×10^{-4}, The left image is of backpropagation method, the right image is of Forward-mode method. (**Second Row**) The learning process of CNN using learning rate 1×10^{-4}, The left image is of backpropagation method, the right image is of Forward-mode method.

IV. Experiments and Results

In this chapter, we show our experimental results for the above approach. We use the MNIST dataset. The networks used in our experiments are CNN and MLP. The architecture of MLP has three fully connected layers of size 1024, 1024, and 10, with ReLU activation function after the first two layers and Softmax + Cross-entropy loss as the loss function. The architecture of CNN has three convolutional layers followed by two fully connected layers. All of these layers use ReLU as the activation function and there is one MaxPooling layer after each convolutional layer. And no learning rate scheduler are used in the experiments. In all experiments, we run different methods for an equal number of iterations. We run the code with CUDA on an Nvidia GeForce RTX 3080 GPU.

A. Naive Forward-Mode method

First we try to compare the naive forward-mode method proposed by paper [1] with the traditional backpropagation. From the results shown in Fig. (4), we can see that we the learning rate is small, the performance of models trained using forward-mode and backpropagation method is essentially the same, and the pattern of loss decline is also the same.

However, during the experiment, as the learning rate increases, the training process of forward-mode method becomes more and more unstable, and even generates the loss value of Nan which makes the training impossible, as shown in Fig. (5). Therefore, it is crucial to reduce the variance of the estimates of the gradients calculated by the Forward-mode method.

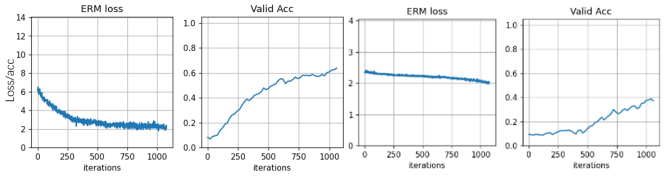

Fig. 5. (**left**) MLP trained with forward-mode method using lr = 3×10^{-4}; **right** CNN trained with backpropagation method using lr = 1×10^{-3}

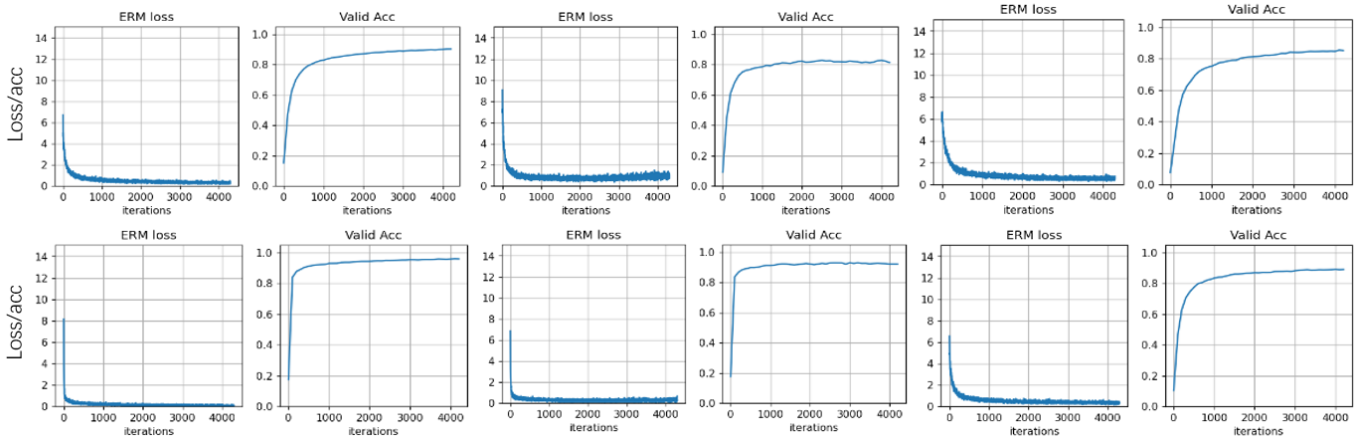

Fig. 6. (**top row**) the training process when using lr = 1×10^{-3}. Three columns from left to right are obtained by 1) Backpropagation method 2)Forward-mode + multi-direction with 3 directions 3)Forward-mode + partition with 2 partitions respectively. (**bottom row**) the training process when using lr = 1×10^{-2}. Three columns from left to right are obtained by 1) Backpropagation method 2)Forward-mode + multi-direction with 100 directions 3)Forward-mode + partition with 10 partitions respectively.

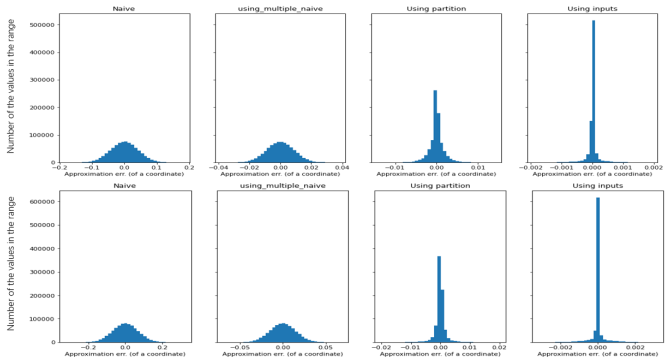

Fig. 7. Distribution of the difference between the true gradients of a model and the estimates of them using forward-mode method combined with methods that can decrease variance. The four graphs in one row corresponding to 1) Naive forward-mode method 2) Averaging between 3 different directions 3) sum of 3 partitions 4) using input to generate random vector from left to right respectively. Two rows correspond to two consecutive layers in MLP

B. Methods used to reduce variance

We propose three different methods to decrease the variance of the estimates for gradients when using the forward-mode method. Their effects are shown in Fig. (7). We can see that all these three methods can decrease the variance compared to the naive version. Among them, the method that uses input to generate a random vector has the greatest reduction in variance.

In practice, it is also possible to train with a larger learning rate using these methods The results of the first two methods are shown in Fig. (6). The maximum learning rate we can achieve without Nan loss value is 3×10^{-4} in the naive forward-mode method.

For the method that uses input to generate a random vector, we conduct two experiments, first is the one that uses QR decomposition and can produce truly unbiased estimates, and the other one is the approximation one, as shown in Fig. (8). We can see that they produce a very similar result, so we can use the second version for the sake of time consumption.

Overall, the method that uses inputs to generate random vectors has the highest degree of variance reduction, but the other two methods are more flexible in their applications because they can control the degree of variance reduction by changing the hyper-parameters (number of random directions, number of partitions), and in practical applications, we can use a mixture of these methods.

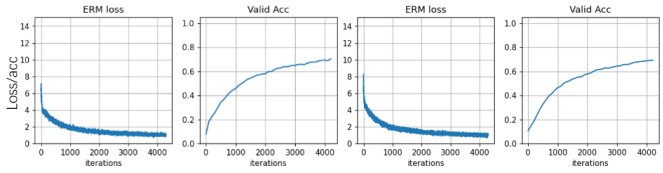

Fig. 8. The training process when using forward-mode method, and use input to generate random vector in order to decrease variance, trained at the lr=1×10^{-3}. (**left**): Use QR decomposition; (**right**): Use the approximation method.

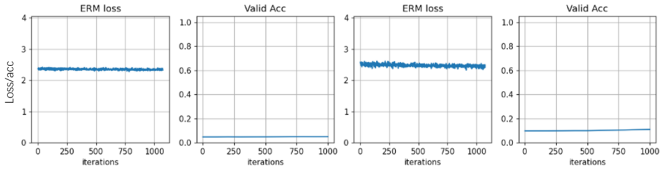

Fig. 9. train the network using lr = 1×10^{-6}. (**left**): using backpropagation method; (**right**): using naive forward-mode method

C. Momentum Forward-mode method

In this subsection, We will show the effect of the Momentum Forward-mode method. First, according to Fig. (9), we can see

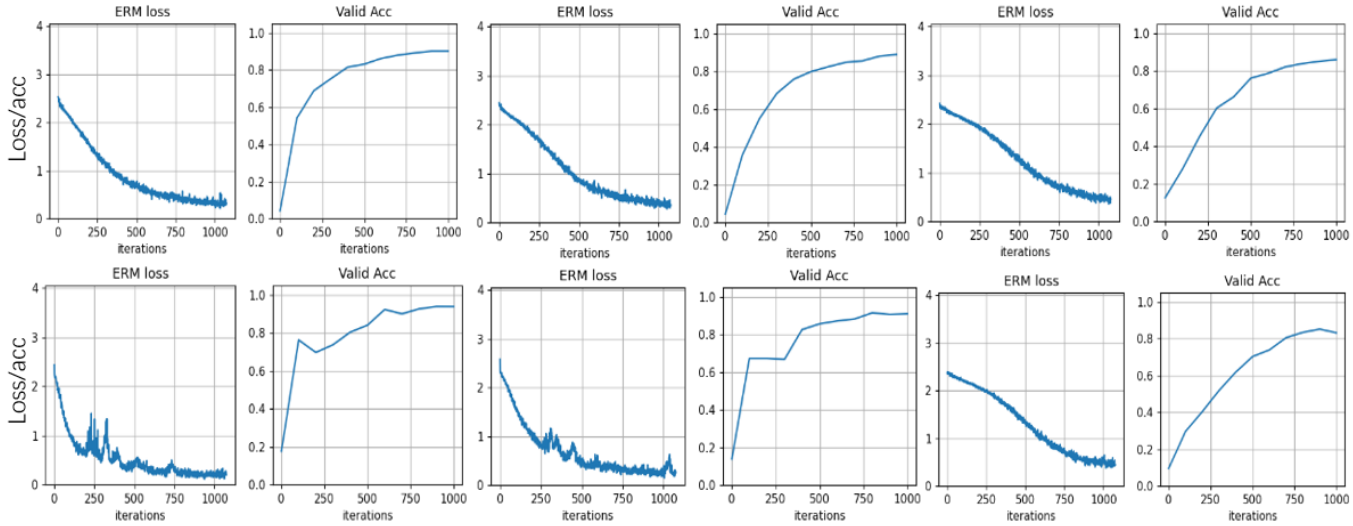

Fig. 10. The learning rate is 1×10^{-6}, fix_gap and decrease_gap are both set to be 10. (**top row**): shuffle_ratio is set to be 0.5, decay_rate is 0.9/0.8/0.7 respectively. (**bottom row**): decay_rate is set to be 0.9, shuffle_ratio is 0.2/0.4/0.5 respectively.

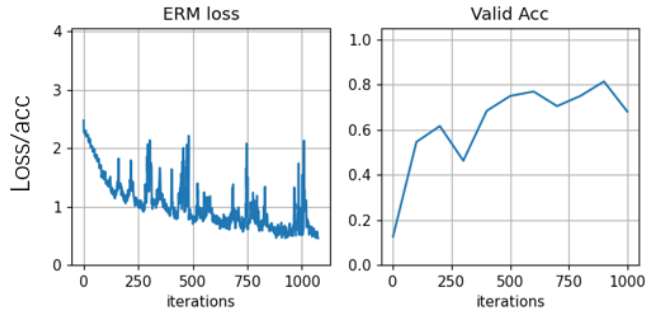

Fig. 11. The training process when lr=1×10^{-6} using momentum forward-mode method,fix_gap=10,decrease_gap=10,decay_rate=1.0,shuffle_ratio=0.0

that when the learning rate is small (1×10^{-6}), neither backpropagation nor forward-propagation can achieve satisfactory learning performance.

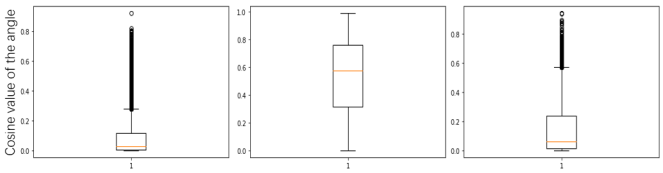

Fig. 12. The y-axis for all three images are the cosine value of the angle between the direction of estimates of gradient and the direction of true gradient. All plots are generated based on the data for more than 1000 iterations. (**left**):trained using naive forward-mode method; (**middle**):trained using momentum forward-mode method with decay_rate=1.0 and shuffle_ratio=0.0, both fix_gap and decrease_gap are set to be 10; (**right**):trained using momentum forward-mode method with decay_rate=0.9 and shuffle_ratio=0.5, both fix_gap and decrease_gap are set to be 10

Next, we try our new method, as shown in Fig. (11), which converges faster at the same learning rate but is very unstable. This is because we use a decay_rate that is 1.0 and a shuffle_ratio that is 0.0, which is an extreme case. Because in this case, the direction of gradient descent strictly follows the direction of the gradient obtained during the backpropagation session, without considering any fluctuations, which is obviously unreasonable, we next try some intermediate parameter settings. The results are shown in Fig. (10).

We can see that choosing the appropriate hyper-parameters can make the convergence stable and faster, similar to the learning process of the backpropagation method at large learning rates. In addition, Fig. (12) shows that the direction of the gradient vector estimated using the momentum forward-mode method is closer to the direction of the real gradient vector. It's shown that estimates using the momentum forward-mode method have a more similar direction to the true gradient.

D. Time and Memory comparison

We implemented the backpropagation method and the various methods mentioned in this paper from scratch. In the implementation, we divide the large program into different component modules, such as the convolution module, random number generation module, mask generation module, etc. We use these modules to implement all the different methods to ensure that the different methods have the same level of optimization so that meaningful comparisons of time performance and memory performance can be generated.

1) Time consumption comparison: The time consumption when using different methods to train MLP (except for the last two rows) is shown in the table. (I) . We can find that there is no significant difference between method 1 and method 2 in terms of running time, which is not consistent with the conclusion in the original paper [1]. The time consumption for reducing the variance by taking multiple random directions and partition methods grows linearly with the number of directions and partitions, and the latter grows faster than the former

because the segmentation process requires us to use loops. The method using inputs to generate random vectors increases the required time slightly, and the momentum forward-mode method increases the time a little bit, but the increase is still quite noticeable on MLP networks, but the increase in runtime is negligible on networks like CNNs, which are inherently time-consuming.

TABLE I
TIME CONSUMPTION FOR DIFFERENT METHODS ON MLP FOR ONE ITERATION WHEN BATCH_SIZE = 256(EXCEPT FOR THE LAST THREE ROWS ARE MEASURED ON CNN)

method	Time
Backpropagation	0.0009s - 0.0011s
Naive forward-mode	0.0009s - 0.0011s
Forward-mode + multi-dir	shown in Fig. (13)
Forward-mode + partition	shown in Fig. (13)
Forward-mode + using input	0.0010s - 0.0012s
Momentum forward mode(Except check point)	0.0012s - 0.0017s
Backpropagation(CNN)	0.025s - 0.03s
Naive forward-mode(CNN)	0.025s - 0.03s
Momentum forward mode(Except check point, CNN)	0.027s - 0.03s

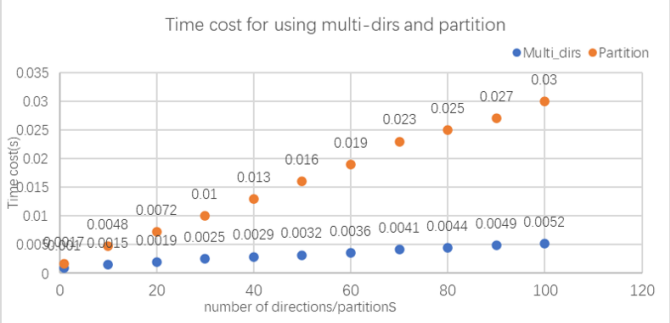

Fig. 13. The change in the amount of time consumption when the number of directions/partitions is changed

2) Memory consumption comparison: We find that the main advantage of the forward-mode method over the backpropagation method is the storage space-saving because the forward-mode method only needs to remember the seeds used to generate random numbers, and even with other improvements, it only stores one more set of gradients that equals to the size of the model's parameters or one more data point in a batch. This increase is much smaller than the storage space required by the backpropagation method. And this saving becomes more and more obvious as the input data batch size increases, as shown in Fig. (14).

V. CONCLUSION

The forward-mode method can greatly save memory space during training, allowing the model to use a larger batch size. while it is more friendly to parallel operation, in the backpropagation method, the calculation of gradients must wait until the end of the forward pass. However, in the forward-mode method, the forward pass to compute results and the forward pass to compute direction derivatives can be performed

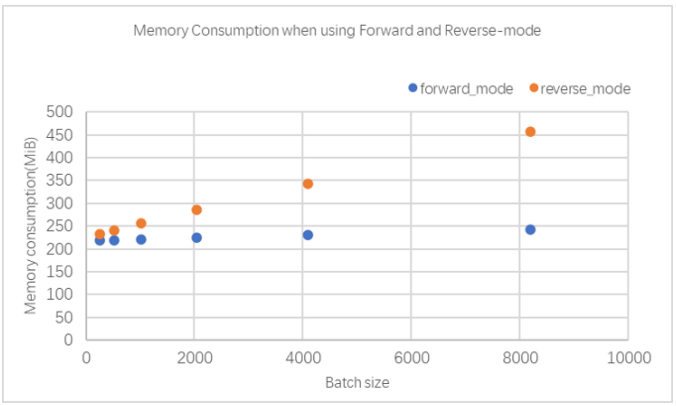

Fig. 14. The change in the amount of memory consumption when the batch size is changed

simultaneously, and the forward passes to compute multiple direction derivatives can also be performed simultaneously, and some intermediate computational results can be shared to speed up the operation. Several methods proposed in this paper can effectively reduce the variance of the gradient estimates, enable the forward-mode method to use a larger learning rate during training, or use the momentum forward-mode method to make the model converge faster even at a smaller learning rate, all of which solve the previous problems of the forward-mode method to some extent.

However, we do not observe a runtime improvement in the forward-mode method compared to the backpropagation method as stated in paper [1]. It is also doubtful whether the proposed methods in this paper, including the forward-mode method itself, can maintain the same performance on larger networks and data, and we believe this should be our next research step.

REFERENCES

[1] A. G. Baydin, B. A. Pearlmutter, D. Syme, F. Wood, and P. Torr, "Gradients without backpropagation," *arXiv preprint arXiv:2202.08587*, 2022.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "{TensorFlow}: a system for {Large-Scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[5] R. E. Wengert, "A simple automatic derivative evaluation program," *Communications of the ACM*, vol. 7, no. 8, pp. 463–464, 1964.

[6] G. Gur-Ari, D. A. Roberts, and E. Dyer, "Gradient descent happens in a tiny subspace," *arXiv preprint arXiv:1812.04754*, 2018.