Nesting Forward Automatic Differentiation for Memory-Efficient Deep Neural Network Training

Cong Guo$^{1,2}$, Yuxian Qiu$^{1,2}$, Jingwen Leng$^{1,2,*}$, Chen Zhang$^3$
Ying Cao$^4$, Quanlu Zhang$^4$, Yunxin Liu$^5$, Fan Yang$^4$, Minyi Guo$^{1,2,*}$
$^1$Shanghai Jiao Tong University, $^2$Shanghai Qi Zhi Institute, $^3$Alibaba Group
$^4$Microsoft Research, $^5$Institute for AI Industry Research (AIR), Tsinghua University

Abstract—An activation function is an element-wise mathematical function and plays a crucial role in deep neural networks (DNN). Many novel and sophisticated activation functions have been proposed to improve the DNN accuracy but also consume massive memory in the training process with back-propagation. In this study, we propose the nested forward automatic differentiation (Forward-AD), specifically for the element-wise activation function for memory-efficient DNN training. We deploy nested Forward-AD in two widely-used deep learning frameworks, TensorFlow and PyTorch, which support the static and dynamic computation graph, respectively. Our evaluation shows that nested Forward-AD reduces the memory footprint by up to 1.97× compared to the baseline model and outperforms the recomputation by 20% under the same memory reduction ratio.

I. INTRODUCTION

Deep neural network (DNN) models have gained tremendous success in many important domains. For example, ResNet [1] and BERT [2] (based on pre-trained Transformer network [3]) have shown impressive accuracy in the challenging area of image classification [4] and natural language processing [5] (NLP) tasks. Researchers have shown that activation functions are important elements in DNN models, and proposed many novel activation functions for the better accuracy. For most vision tasks, Mish [6], Swish [7] and GELU [8] surpass ReLU, specifically, about 1% accuracy improvement on ResNet [6], [7]. On the other hand, GELU [8] is the most widely used activation function in the NLP models and achieves the best accuracy among other candidates.

The activation functions mentioned above are characterized by their sophisticated architectures and massive memory consumption, especially when combined with straightforward implementations. In some cases, they become the top memory consumers, surpassing the intermediate variables of convolution or fully connected layers saved in the forward pass for gradient computation. The employment of novel activation functions exacerbates the model applicability when training with modern commodity accelerators, such as GPUs, with limited global memory capacity. For example, the activation-related variables of BERT-base and ResNet-50 occupy 22% and 52% memory footprint, respectively, as shown in Fig. 1.

Many memory optimization approaches [9]–[12] have been proposed, but none of them is appropriate for activation functions without introducing extra computing overhead. To reduce activation functions’ footprint usage during model training, recomputation [9], [12] reproduces the intermediate variables in the backward pass without saving them in memory. However, the recomputation inserts new operators breaking the original computation graph and brings extra computing overhead, dropping the speed of training.

We introduce forward mode automatic differentiation (Forward-AD, FAD) for gradient computation of activation functions to avoid the recomputation overhead and reduce the memory footprint. Automatic differentiation [13] (autodiff, AD) is a family of mathematics tools to automatically and accurately evaluate numeric function derivatives using computer programs. There are two modes of autodiff: backward mode (i.e., back-propagation, BP) and forward mode (FAD).

Back-propagation is the mainstay approach for DNN training. Today’s mainstream machine learning frameworks, such as PyTorch [14] and TensorFlow [15], implement BP using dynamic/static computational graphs and significantly improve model deployment efficiency. Compared to back-propagation AD, forward AD can reduce the stored intermediate variables and execute efficiently and straightforward in the specific numeric function $f : \mathbb{R}^N \rightarrow \mathbb{R}^M$, $(N \leq M)$. The activation functions are the typical element-wise functions with $f : \mathbb{R}^1 \rightarrow \mathbb{R}^1$, whose computation graphs can be optimized by FAD to reduce the DNN training’s memory consumption.

We propose an element-wise specific computation graph optimization, which substitutes the original BP execution of subgraphs with nested FAD by recognizing the original graph’s specific function pattern $(\mathbb{R}^1 \rightarrow \mathbb{R}^M)$ without influencing the remaining graph. This approach can automatically optimize both static and dynamic computation graphs in the popular deep learning frameworks, especially in the imperative mode in PyTorch, and achieves memory reduction for DNN training. The contribution of our work is as follows:

- We propose a memory optimization approach with the nested FAD inside BP in the deep learning framework.
- The nested FAD can automatically be executed in the popular frameworks with the two execution modes: dynamic and static computation graphs.
- We evaluate FAD in the state-of-the-art models achieving a higher memory reduction ratio (as high as 1.97× and 1.34× on average) than the original end-to-end model and an average of 1.78× speedup than recomputation for the activation function on BERT.

* Jingwen Leng and Minyi Guo are corresponding authors of this paper.
We organize the paper as follows. Sec. II introduces the background of the automatic differentiation and the overview of the nested FAD algorithm. We explain nested FAD implementation with static and dynamic computation graph in Sec. III and Sec. IV, respectively. We evaluate the nested FAD with DNN models in Sec. V, introduce the related work in Sec. VI, and conclude in Sec. VII.

II. ACTIVATION AND AUTODIFF

This section first introduces the relevant background on the activation function and analyzes the memory efficiency of the activation function in DNN models. Then, we compare the differences among BP, recomputation, and FAD with a specific example. Finally, we define the nested FAD in the BP algorithm of the DNN models.

A. Activation functions

DNN models have recently achieved state-of-the-art results in many important domains, e.g., convolution neural network [16] (CNN) in the computer vision domain, and BERT [2] in the natural language processing domain. Recently, many activation functions, e.g., Mish [6], Swish [7], and GELU [8], have been proposed to optimize ResNet-50 [1] and BERT [3], showing higher accuracy than ReLU. Their formulas are shown in Tbl. I.

| Activation | Forward pass | Backward pass (derivative) |
|------------|--------------|-----------------------------|
| GELU       | $0.5 \cdot x \cdot \{1 + \tanh[\sqrt{2/\pi} \cdot (x + 0.044715 \cdot x^3)]\}$ | $g \cdot v_2 \cdot [\sigma(x)(1 - \sigma(x))] = g \cdot v_2 \cdot [v_1(1 - v_1)]$, (1) |
| Mish       | $x \cdot \tanh[\ln(1 + e^x)] = x \cdot \tanh[\text{softplus}(x)]$ | based on the $\sigma(x)$ derivative, $\sigma(x)' = \sigma(x) \cdot (1 - \sigma(x))$. Node $v_2'$ calculates the gradient $v_2' = g \cdot v_1 \cdot 1$ for $x'$. Finally, Node $x'$ has the gradient $x' = v_1' + v_2' = g \cdot v_2 \cdot [v_1(1 - v_1)] + g \cdot v_1$ (2) |
| Swish      | $x/(1 + e^{-x}) = x \cdot \text{sigmoid}(x)$ | from the two branches $v_1'$ and $v_2'$. |

| TABLE I: Activation functions. |

We collect the memory footprint usage in ResNet-50 and BERT with activation functions GELU and Swish in PyTorch. The two activations have different intermediate variables usage due to their different derivative functions. The memory breakdown results in Fig. 1 show great potential for memory optimization utilizing FAD, which, without the burden of saving intermediate variables, has much less memory footprint than back-propagation. It is noteworthy that the proportion of IM-ACT (and IM-Other) will increase as the batch size increase. That will strengthen the performance of Forward-AD.

C. Forward-AD, BP and recomputation

We explain the difference between recomputation and FAD using the Swish activation function as an example. The formula of Swish is shown in Tbl. II. Swish function is a simple activation that has two operations: Mul and Sigmoid. Fig. 2 depicts the three approaches: BP, recomputation, and FAD with an input value $x$.

Original BP (1) algorithm first executes the forward pass with three nodes:

$$v_1 = \sigma(x), \quad v_2 = x, \quad v_3 = v_1 \cdot v_2.$$  

In backward pass, Node $v_3'$ receives its gradient $g$ from upstream and calculates gradient for $v_1'$ and $v_2'$. Obviously, backpropagation needs to save intermediate variables $v_1$ and $v_2$ for computing Swish gradient. Node $v_3'$ will deliver downstream gradients $g \cdot v_1$ to Node $v_1'$, and $g \cdot v_2$ to Node $v_2'$. Then, BP continues to traverse $v_1'$ and $v_2'$ recursively. Node $v_1'$ calculates the gradient by

$$v_1' = g \cdot v_2 \cdot [\sigma(x)(1 - \sigma(x))] = g \cdot v_2 \cdot [v_1(1 - v_1)],$$  

based on the $\sigma(x)$ derivative, $\sigma(x)' = \sigma(x) \cdot (1 - \sigma(x))$. Node $v_2'$ calculates the gradient $v_2' = g \cdot v_1 \cdot 1$ for $x'$. Finally, Node $x'$ has the gradient

$$x' = v_1' + v_2' = g \cdot v_2 \cdot [v_1(1 - v_1)] + g \cdot v_1$$  

from the two branches $v_1'$ and $v_2'$.

Recomputation (2) has the same forward pass as BP but releases $v_1$ and $v_2$ to reduce the memory footprint, only save the source variable $x$. Therefore, Recomputation has to recalculate the values of $v_1$ and $v_2$ in the backward pass with extra computation overhead, as show in Fig. 2 middle.

Forward-AD (3) derives and accumulates the derivative $v_3'$ of Swish in the forward pass and multiplies with $g$ in the backward pass. FAD simultaneously calculates Node $v_1' = v_1(1 - v_1)$ and $v_2' = 1$ when Node $v_1$ and $v_2$ are executed. Finally, FAD continues to calculate

$$v_3' = v_1 \cdot v_2' + v_1' \cdot v_2 = v_1 \cdot 1 + v_1(1 - v_1) \cdot v_2,$$  

where $v_1(1 - v_1) \cdot v_2$.
when the forward graph merges the Node $v_1$ and $v_2$ into $v_3$ using multiplication operation. FAD can only store the intermediate variable $v'_3$ for back-propagation with upstream gradient $g$ and achieve the final gradient

$$x' = g \cdot v'_3 = g \cdot [v_1 + v_1(1 - v_1) \cdot v_2] \tag{4}$$

directly for node $x'$.

Obviously, FAD is much more efficient than recomputation because of the memory locality without any extra computation. FAD can reduce all the intermediate variables only save the derivative $v'_3$ for the activation function instead of the variables $x$, $v_1$, and $v_2$ comparing to BP algorithm.

**D. Forward AD Applicability**

Autodiff can generalize the derivative of a function $f: \mathbb{R}^N \rightarrow \mathbb{R}^M$ by computing the Jacobian matrix $J_f$ with the shape of $(M \times N)$, where $M$ is the length of output vector and $N$ is for the input vector. We can compute the $J_f$:

$$J_f = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_M}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_N} & \cdots & \frac{\partial y_M}{\partial x_N} \end{bmatrix}$$

Forward AD computes the Jacobian–vector products (JVP): $J_f v$ and back-propagation computes the vector-Jacobian products (VJP): $(v^T J_f)^T = J_f^T v$. For example of Swish case, FAD compute the $v'_1$ and $v'_2$ with

$$v'_{v_1,2} = [v'_1, v'_2]^T = J_{f,v} x' = \begin{bmatrix} \frac{\partial y_1}{\partial x} \\ \frac{\partial y_2}{\partial x} \end{bmatrix} \cdot [1],$$

where $x' = 1$ and compute the $v'_3$ with

$$v'_{v_3} = [v'_3] = J_{f,v} v'_{v_1,2} = \begin{bmatrix} \frac{\partial y_1}{\partial x} \\ \frac{\partial y_2}{\partial x} \end{bmatrix} \begin{bmatrix} \frac{\partial v_1}{\partial x} \\ \frac{\partial v_2}{\partial x} \end{bmatrix} \cdot [1]$$

For BP, we have

$$v'_v = J_{f,v} J_{f,v'} v' = J_{f,v} J_{f,v'} v'_{v_1,2},$$

where $v'_{v_1,2} = [1]$.

**Computation efficiency** Evidently, for cases $f : \mathbb{R}^N \rightarrow \mathbb{R}^M$, $N \leq M$, FAD is efficient for computing derivatives, and vice-versa for BP with $N > M$ [17]. According to our observation, manual differentiation is widely adopted by machine learning frameworks with tremendous optimization for high performance. TensorFlow and PyTorch utilize manual differentiation to compute the derivatives and fuse operations manually. For example, layer/batch normalization, convolution, and matrix multiplication are accelerated by the highly optimized library. That optimization exploits the Jacobian matrix sparsity due to their inner reduction (accumulation) operations and is incompatible with automatic differentiation, neither backward (BP) nor forward (FAD). From the automatic differentiation perspective, the optimization is to minimize numbers of multiplication for each subregion. It is known as the Optimal Jacobian Accumulation (OJA) problem, which has been proved to be an NP-complete problem [18].

To practically implement FAD in the DNN framework, we simplify the applicability of FAD with the function $f: \mathbb{R}^1 \rightarrow \mathbb{R}^M$, specifically, element-wise operations with one input variable and $M \geq 1$ output variables, including most of the activation functions.

**E. Nesting FAD**

We can nest FAD inside the BP algorithm for DNN models. Without loss of generality, let $v_1, v_2, \ldots, v_k$ be $k$ nodes in the topological ordering for the DNN computation graph $G$ and $v_k$ is the loss $L$. The gradient of the node $v_i$ computed by back-propagation is:

$$\frac{\partial L}{\partial v_i} = J_{v_i}^T \frac{\partial L}{\partial v_{i+1}},$$

Here, the $\frac{\partial L}{\partial v_{i+1}}$ is the upstream gradient from the node $v_{i+1}$. The $J_{v_i}$ is the Jacobian matrix of $v_i$. Assume that $v_{n}, \ldots, v_{m}$ in topological ordering are FAD primitive operations and they
can compose to a $\mathbb{R}^1 \to \mathbb{R}^M$ element-wise operation. Then we have

$$\frac{\partial L}{\partial v_n} = \left\{ \prod_{i \in \{n, \ldots, m\}} J_{v_i} \right\}^T \frac{\partial L}{\partial v_{m+1}} = \frac{\partial v_{m+1}}{\partial v_n} \cdot g,$$

where $g$ is the upstream gradient. Therefore, we can exploit FAD to compute the Jacobian matrix for the element-wise operation composed by $v_n, \ldots, v_m$ within the forward pass. Then, FAD updates the gradients with upstream gradient $g = \frac{\partial L}{\partial v_{m+1}}$.

III. DEFINITIONS AND STATIC GRAPH OPTIMIZATION

We optimize and nest element-wise specific FAD in two types of computation graph: static and dynamic with TensorFlow and Pytorch respectively.

A. Definitions

We divide the operations (operator, OP) into two classes: \texttt{fad} (FAD-compatible) and \texttt{nfad} (FAD-incompatible). The primitive operations for FAD basically are \texttt{fad} element-wise functions including:

- The \texttt{unary} operation ($f: \mathbb{R}^1 \to \mathbb{R}^1$) can be the primitive operation, e.g., Node 1 in Fig. 3. The \texttt{binary} operation with one constant input variable can also be regarded as the \texttt{unary} operation.
- For \texttt{binary} operation ($f: \mathbb{R}^2 \to \mathbb{R}^1$), such as addition and multiplication, their two input elements should be originated from the same source element converting $\mathbb{R}^2 \to \mathbb{R}^1$ to $\mathbb{R}^1 \to \mathbb{R}^1$, e.g., Node 2 in Fig. 3.
- Except the \texttt{fad} operations, others operations are \texttt{nfad}. Especially, the \texttt{binary} operation with two tensors from different sources is the \texttt{nfad} (FAD-incompatible) OP and violates the $f: \mathbb{R}^1 \to \mathbb{R}^1$, e.g., Node 3 in Fig. 3.

Manual differentiation [13] is widely used in frameworks. It can greatly impact the efficiency of the element-wise operation. For example, \texttt{Sigmoid} is a classic activation function: $S(x) = \frac{1}{1 + e^{-x}}$ and would produce lots of intermediate variables with the original format. Frameworks implement its enclosed symbolic format as a basic operator. Without loss of practicability, we define the symbolic operators implemented in frameworks as the primitive operation, including \texttt{tanh}, \texttt{sigmoid} and \texttt{softplus}.

The operation composed of finitely many primitive operations is the FAD-compatible operation and can maintain $f: \mathbb{R}^1 \to \mathbb{R}^M$, $M \geq 1$, which is efficient for FAD and the subject matter of this paper. The others are \texttt{nfad} OPs.

B. Static computation graph

TensorFlow is the most representative static computation graph framework. It defines the graph statically before a model execution and the graph can not be modified during the run-time. That is friendly to optimize and accelerate the computation of the DNN model without convenience and flexibility for users. Naturally, it is easier for a static graph to implement the nested FAD than a dynamic graph.

There are two steps with the computation graph. First of all, TensorFlow builds a static computation graph with forward pass and backward pass. With the definitions of \texttt{fad} operation, we can divide the forward computation graph into two types of sub-graphs: FAD and non-FAD. Second, each sub-graph can be regarded as a node with in-degree and out-degree. We find FAD sub-graphs with 1 in-degree and $m \geq 1$ out-degree and optimize the computation. Finally, we get FAD intermediate derivatives, which will be saves for the backward gradient computation mentioned in Fig. 4.

This procedure can be implemented after the gradient computation graph built by the \texttt{backward} function before the graph automatic optimization. We eliminate the original connection between forward and backward for intermediate variables and the extra computation nodes in FAD sub-graphs, then attach FAD computation graph on the original forward computation graph and save the only single tensor of FAD derivative for the backward, finally build a new node in backward for accumulating the derivatives and computing the gradient.

After that, TensorFlow will automatically execute the computation graph optimization, such as prune nodes that do not affect the output, eliminate common subexpressions, and simplify arithmetic statements. FAD will not impact the original forward computation and reuse the tensor immediately and is friendly to accelerator memory locality.

IV. DYNAMIC COMPUTATION GRAPH OPTIMIZATION

The insight of FAD optimization on dynamic computation graph is similar to the static one. However, its complexity is more significant because the imperative mode can not provide enough graph information. We design a finite state machine and an interactive approach for dynamic graph optimization to maintain the correctness and efficiency of FAD.

A. Finite State Machine

First, we need to maintain the correctness of FAD algorithm with the pattern: $\mathbb{R}^1 \to \mathbb{R}^M$. This goal is easy to reach in
the static graph from the global perspective but difficult in the dynamic graph. We define T-O (tensor-operator) pairs as the basic optimization unit for the dynamic graph. The T-O pairs have two types of OPs (fad/nfad), and two types of input tensors: fad/nfad tensor is produced by a previous fad/nfad operator. Denote the fad by Y and nfad by N. Then, we have 4 forward states for T-O pairs: NN, NY, YY, and YN.

For the imperative mode, we can regard the next OP as an input event. Therefore, we can construct a finite state machine (FSM) for forward pass depicted in Fig. 5. Naturally, the DNN models can be represented by a finite forward pass state sequence, where the original BP has only NN states. According to the FSM, NY is the beginning of a FAD sequence ended by YN. Based on the fad OP definition in Sec. III-A, the NY forward mode has a single input source, and YN can be referenced by multiple OPs. Therefore, we can adopt the forward mode FSM to ensure that FAD sequence can meet the specific pattern: $\mathbb{R}^1 \rightarrow \mathbb{R}^M$.

Fig. 5 presents an example of a fad sequence execution with $\mathbb{R}^1 \rightarrow \mathbb{R}^M$. The first node (Node 1) has the original NN forward mode and produce a nfad output tensor transferred to the next fad Node 2. Assume that Node 2 has a fad operator and will produce a fad output tensor. Then, we denote the Node 2 with NY forward mode, which is the beginning node of the fad sequence. Then, Node 3 and Node 4 have fad operators and are YY forward modes. Finally, the Node 5 and Node 6 are both end nodes with the nfad and produce nfad output tensors.

### B. Interactive Approach

We should release the intermediate variables tensor appropriately and accurately to reduce the memory footprint with high memory efficiency. For example, the imperative mode has executed the operation with tensor $x$. It is unknown whether the tensor $x$ would be used in the following program fragment. That can impact the tensor when and where to release. Suppose we release the tensor $x$ immediately after FAD computation. That may cause serious fault when $x$ is on the critical path and referenced by the following operators. On the contrary, the tensor $x$ will impact the memory efficiency if we release the tensor too late. Therefore, how and when to release the tensor are essential for the imperative framework. To tackle this issue, we design an interactive approach based on the forward pass FSM with the four modes shown in Fig. 6.

**Parameters** There are two groups of parameters. We denote the original parameters for back-propagation in black. There are four parameters.

- **back_fn** (BFN) is the backward function, which is implemented by the original framework;
- **saved_tensors (node)** (STN) is short for the saved intermediate tensors for computing the gradient using BFN;
- **grad_node** is a BP node with BFN and STN;
- **next_node** (NXN) denotes the pointer of the next BP node.

We need to add four new FAD parameters (blue) to support the forward pass FSM and interactive approach.

- **isFAD** (FAD) identifies the fad tensor;
- **saved_tensors (tensor)** (STT) is the same as STN but temporary. It can be recursively released when the tensor is released;
- **fad_tensors** (FTR) is the fad result tensors. Particularly, the FTR sets a scalar value 1 for the NY (beginning) mode;
- **src_node** (SCN) is the head node of a FAD sequence. The last YY nodes’ NXN will point it in the post-process.

### Execution

There are four basic execution components and each forward mode executes the different execution components shown in Tbl. III and Fig. 6.

- **A. Forward computation** execute the forward operation and is the basic unit for all modes.
- **B. Forward-AD computation** can immediately calculate the derivatives for the node with the forward differentiation mode. It is only for modes with a fad tensor (YY and YN modes). FAD computation exploits the original BFN with the STT and FTR of the input tensor, such as the fad_t1 of YY mode in Fig. 6.
- **C. Forward-AD post-process** is only for YN mode (end state). The post-process is only for the YN mode. It will save FAD result tensors as the STN and update BFN and NXN for the previous node. For example, the YN mode in Fig. 6 computes FAD result fad_t2 and saves it for the STN of Node $f_2$. Then, FAD post-process replaces the BFN with a direct Multiplication with upstream gradient and updates the NXN with the head Node $f$. For a binary operator, the post-process will merge (accumulate) FAD results from the two input tensors.
- **D. Parameters update** is also the basic unit for all modes, which updates the node and tensor parameters. For the NN mode, it is the original update process for the BP node. The NY and YY modes only update the information with the tensor and will skip the BP nodes, which will not be used in the Forward-AD. For the YN mode, it will update the BP node based on the Forward-AD post-process.

| NN | NY | YY | YN |
|----|----|----|----|
| A  | B  | A  | D  |
| A  | B  | A  | B  |
| A  | B  | C  | D  |

**TABLE III: Forward pass execution**
**Resource release** After 4 modes are executed sequentially, the Tensors $\mathbf{w, x}$ and Node $f_1$ have no reference on them in Fig. 6. The Tensor $y$ may have a reference determinet by derivative function of $f_2$. It is known that PyTorch exploits the reference count and garbage collector mechanism of Python. PyTorch can make sure that all intermediate variables are released as soon as they become unneeded [14]. When the Tensor $\mathbf{w, x}$ and node $f_1$ are released, the STT and FTR referenced by Tensor $\mathbf{w, x}$ are also released automatically and recursively.

**Efficiency** The interactive approach utilizes information from individual operators with high efficiency and robustness. First, the approach does not influence the original forward mode (NN mode). Second, the approach will skip the intermediate nodes, such as Node $f_1$ and original Node $f_2$, leading to an efficient backward gradient execution. The last updated Node $f_2$ can directly pass the gradient to the head Node $f$. Besides, if FAD sequence only has a single $\text{fad}$ operator, i.e., NY $\rightarrow$ YN post-process (C) in YN mode will easily skip FAD computation (E) to avoid the extra computation with FTR = 1. Therefore, the approach is highly robust to the pattern $\mathbb{R}^1 \rightarrow \mathbb{R}^M, M \geq 1$ based on the FSM.

Finally, we exploit the interactive approach to optimize the dynamic computation graph for the T-O pairs and embed the nested FAD computation inside the forward pass without influence on BP. The mechanism is based on FSM and original garbage collector and maintains FAD’s correctness and high memory efficiency.

**V. Evaluation**

In this section, we perform the nested Forward-AD overhead and memory efficiency evaluations in both static and dynamic computation graphs.

**A. Methodology**

For the implementation of the static computation graph, we modify TensorFlow and embed a hook function for FAD optimization as detailed in Sec. III-B before execution. Then, TensorFlow can optimize the computation graph without saving the intermediate variables. For the implementation of the dynamic computation graph, we use the “Autograd Function” in PyTorch to override the specific $\text{fad}$ operators, including add, mul, exp, tanh, softplus, sigmoid, etc.

Owing to the space limitation, we evaluate two popular neural networks, ResNet [1] (CNN) and BERT [2] (Transformer), which cover tasks from the computer vision and NLP domain. We evaluate ResNet-50 for image classification on the ImageNet [4] dataset. ResNet has been widely used as the backbone in many applications, e.g., Mask R-CNN for instance segmentation [19]. For the state-of-the-art Transformer model family, we use BERT-base with 128 sequence length on GLUE (general language understanding evaluation) [20] dataset. We only present the results on four datasets (MRPC, ColA, SST-2, and MNLI). Finally, we implement the activation functions Mish [6], Swish [7], and GELU [8] on the networks respectively. All the experiments are conducted on the NVIDIA RTX 2080Ti GPU [21] with 11 GB of off-chip GDDR-based global memory.

**B. Memory Footprint Reduction**

By injecting FAD into DNN frameworks, we can reduce the memory footprint and allow larger batches for the DNN training. Therefore, we first adopt the metric of maximum supported batch size to represent the amount of memory footprint. Fig. 7 depicts the maximum batch size results for
Recomputation First, we measure the runtime overhead of for ResNet-50.

Even though forward automatic differentiation is widely studied in mathematics [36] and some deep learning frameworks (such as TensorFlow [15] and JAX [37]) have supported forward AD, they cannot dynamically nest forward AD into backward AD to update weight, users have to manually combine the separate forward/backward AD sequence.

| Dataset | MRPC | CoLA | SST-2 | MNLI |
|---------|------|------|-------|------|
| Acc.    | Time | MCC  | Time  | Acc. | Time | Acc. | Time |
| Base    | 86.02| 111s | 59.12 | 258s | 91.40| 0.57h| 83.36| 3.34h|
| REC     | 86.02| 118s | 60.06 | 277s | 91.74| 0.61h| 83.54| 3.52h|
| FAD     | 86.02| 101s | 60.06 | 235s | 91.74| 0.51h| 83.54| 2.95h|

TABLE IV: The fine-tuning time and accuracy for BERT-Base model on four datasets (MRPC, CoLA, SST-2, MNLI). MCC is the Matthews correlation [22].
In this paper, we propose the nested Forward-AD in the DNN framework that reduces the memory footprint. The key insight is that FAD is much more memory efficient than BP with the specific pattern ($\mathbb{R}^I \rightarrow \mathbb{R}^M, M \geq 1$). We deploy nested FAD in static and dynamic computation graph design with up to 1.97× memory reduction than the baseline model without the overhead.

ACKNOWLEDGMENT

This work was supported by the National Key R&D Program of China under Grant 2021ZD0110104, the National Natural Science Foundation of China (NSFC) grant (U21B2017, 620272297, and 61832006). The authors would like to thank the anonymous reviewers for their constructive feedback for improving the work. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770–778.
[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” arXiv preprint arXiv:1810.04805, 2018.
[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in Advances in neural information processing systems, 2017, pp. 5998–6008.
[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in 2009 IEEE conference on computer vision and pattern recognition. Ieee, 2009, pp. 248–255.
[5] C. Manning and H. Schutze, Foundations of statistical natural language processing. MIT press, 1999.
[6] D. Misra, “Mish: A self regularized non-monotonic neural activation function,” arXiv preprint arXiv:1908.08681, 2019.
[7] P. Ramachandran, B. Zoph, and Q. V. Le, “Searching for activation functions,” arXiv preprint arXiv:1710.05941, 2017.
[8] D. Hendrycks and K. Gimpel, “Gaussian error linear units (gelus),” arXiv preprint arXiv:1606.06164, 2016.
[9] P. Jain, A. Jain, A. Nouisimha, A. Gholami, P. Abbeel, K. Keutzer, I. Stoica, and J. E. Gonzalez, “Checkmate: Breaking the memory wall with optimal tensor rematerialization,” MLsys, 2019.
[10] M. Rhu, N. Gimelshein, J. Clemons, A. Zulqifri, and S. W. Keckler, “vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design,” in 2016 49th MICRO, 2016, pp. 1–13.
[11] M. Rhu, M. O’Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, “Compressing dma engine: Leveraging activation sparsity for training deep neural networks,” in 2018 HPCA, 2018, pp. 78–91.
[12] T. Chen, B. Xu, C. Zhang, and C. Guestrin, “Training deep nets with sublinear memory cost,” arXiv preprint arXiv:1604.06174, 2016.
[13] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, “Automatic differentiation in machine learning: a survey,” Journal of Machine Learning Research, vol. 18, pp. 1–43, 2018.
[14] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” 2017.
[15] 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 OSDI, 2016, pp. 265–283.
[16] Y. LeCun, Y. Bengio et al., “Convolutional networks for images, speech, and time series,” The handbook of brain theory and neural networks, vol. 3361, no. 10, p. 1995, 1995.
[17] F. Wang, D. Zheng, J. Decker, X. Wu, G. M. Essertel, and T. Rompf, “Demystifying differentiable programming: Shift/reset the penultimate backpropagator,” Proceedings of the ACM on Programming Languages, vol. 3, no. ICFP, pp. 1–31, 2019.
[18] U. Naumann, “Optimal jacobian accumulation is np-complete,” Mathematical Programming, vol. 112, no. 2, pp. 427–441, 2008.
[19] K. He, G. Gkioxari, P. Dollár, and R. Girshick, “Mask r-cnn,” in 2017 IEEE International Conference on Computer Vision (ICCV), 2017, pp. 2980–2988.
[20] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, “Glue: A multi-task benchmark and analysis platform for natural language understanding,” in ICLR, 2019.
[21] NVIDIA, “NVIDIA Turing GPU Architecture Whitepaper,” 2018.
[22] B. W. Matthews, “Comparison of the predicted and observed secondary structure of 14 phage lysozyme,” Biochimica et Biophysica Acta (BBA)-Protein structure, vol. 405, no. 2, pp. 442–451, 1975.
[23] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, “Capuchin: Tensor-based gpu memory management for deep learning,” in Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 80–90.
[24] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, “Superneurons: Dynamic gpu memory management for training deep neural networks,” in Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming, 2018, pp. 41–53.
[25] M. Rupam, T. Kaul, and T. Krishna, “Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training,” in Proceedings of the IEEE/ACM 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020, pp. 860–873.
[26] E. Choukroun, M. B. Sullivan, M. O’Connor, M. Erez, J. Pool, D. Nellans, and S. W. Keckler, “Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus,” in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020, pp. 926–939.
[27] M. Nagel, R. A. Amjad, M. Van Baalen, C. Louizos, and T. Blankevoort, “How or down? adaptive rounding for post-training quantization,” in International Conference on Machine Learning. PMLR, 2020, pp. 7197–7206.
[28] C. Guo, C. Zhang, J. Leng, Z. Liu, F. Yang, Y. Liu, M. Guo, and Y. Zhu, “Ant: Exploiting adaptive numerical data type for low-bit deep neural network quantization,” arXiv preprint arXiv:2208.14266, 2022.
[29] C. Guo, Y. Qiu, J. Leng, X. Gao, C. Zhang, Y. Liu, F. Yang, Y. Zhu, and M. Guo, “SQunt: On-the-fly data-free quantization via diagonal hessian approximation,” in International Conference on Learning Representations, 2022. [Online]. Available: https://openreview.net/forum?id=JXhROKNZzOc
[30] E. Qin, A. Samadjar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, “Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training,” in 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2020, pp. 58–70.
[31] Y. Qiu, J. Leng, C. Guo, Q. Chen, C. Li, M. Guo, and Y. Zhu, “Adversarial defense through network profiling based path extraction,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), June 2019.
[32] Y. Gao, Y. Qiu, J. Leng, M. Guo, and Y. Zhu, “Prolemy: Architecture support for robust deep learning,” in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2020, pp. 241–255.
[33] C. Guo, B. Y. Houhe, J. Leng, Y. Qiu, Y. Guan, Z. Wang, X. Jia, X. Li, M. Guo, and Y. Zhu, “Accelerating sparse dnn models without hardware-support via tile-wise sparsity,” in SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2020, pp. 1–15.
[34] Y. Wang, C. Zhang, Z. Xie, C. Guo, Y. Liu, and J. Leng, “Dual-side sparse tensor core,” in 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2021, pp. 1083–1095.
[35] J. Revels, M. Lubin, and T. Papamarkou, “Forward-mode automatic differentiation in Julia,” arXiv:1607.07902 [cs.MS], 2016.
[36] J. Bradbury et al., “JAX: composable transformations of Python+NumPy programs,” 2018. [Online]. Available: http://github.com/google/jax