Understanding Forward-Mode Auto Differentiation

Mani Kishan Ghantasala
University of Massachusetts-Amherst
Amherst, Massachusetts, USA
mghantasala@umass.edu

Srimathi Mahalingam
University of Massachusetts-Amherst
Amherst, Massachuseets, USA
srimathimaha@umass.edu

Abstract

This report investigates the performance and memory usage of forward automatic differentiation (AD) in PyTorch for large model training. We focus on the application of Joint Vector Product (JVP) using PyTorch's forward-mode AD capabilities, specifically in the context of training a BERT model on the AG News dataset for text classification, which is categorized into four classes: World, Sports, Business, and Science/Technology.

The primary objective of this study is to understand the efficiency and practicality of forward-mode AD in handling the complexities and computational demands of large-scale natural language processing (NLP) models like BERT. By profiling memory usage and evaluating performance, we aim to identify the strengths and limitations of forward-mode AD, especially in comparison to more traditional back-propagation methods.

Through this comprehensive analysis, the report aims to provide actionable insights into the use of forward-mode AD in PyTorch for large model training, contributing to more efficient and scalable machine learning practices in federated learning environments.

Keywords

PyTorch, JAX, Forward Automatic Differentiation, Joint Vector Product, Memory Profiling, Federated Learning, BERT, Natural Language Processing, Optimization Techniques, Out-of-Memory Errors.

References

1 Introduction

1.1 Task Description

The primary task in this study involves using automatic differentiation to train a machine learning model on a text classification task. Specifically, we aim to classify news articles into predefined categories using a neural network model.

1.2 Dataset

The AG News dataset serves as a standard benchmark for evaluating the effectiveness of machine learning algorithms in classifying news articles. This dataset contains news articles categorized into four classes: World, Sports, Business, and Science/Technology. It is widely used for training and testing text classification models due to its balanced class distribution and substantial size.

1.3 Federated Learning

Federated Learning is a decentralized approach to training machine learning models across multiple devices or servers holding local data samples, without exchanging them. Each device performs local computations to compute gradients based on its local data using automatic differentiation techniques. These gradients are then shared or aggregated with a central server or coordinator. The central server uses the received gradients to update the global model parameters. This process iterates, with the updated global model being redistributed to the devices, and the cycle repeats until convergence criteria are met. Federated learning enhances privacy by keeping raw data localized and reduces the risk of data breaches.

1.4 BERT Model

BERT (Bidirectional Encoder Representations from Transformers) is a pre-trained language representation model built upon the transformer architecture. It is designed to efficiently process sequential data like text by using self-attention mechanisms to weigh the importance of different words in a sentence when encoding their representations. BERT is trained on a large corpus of text in an unsupervised manner, learning to predict masked words in a sentence and the next sentence in a pair. This pre-training enables BERT to capture a wide range of linguistic information, making it highly effective for various natural language processing tasks, including text classification.

1.5 Types of Differentiation

Differentiation, a cornerstone of calculus, plays a pivotal role in various scientific and engineering fields, especially in machine learning for optimization tasks. There are several types of differentiation, each with unique characteristics and applications.

1.5.1 Manual Differentiation

Involves deriving the derivative of a function by hand, using standard calculus rules. This method is primarily educational, helping to understand the principles of calculus and solve simple problems. However, it becomes impractical for complex functions or high-dimensional problems due to its time-consuming nature and susceptibility to errors.

1.5.2 Numeric Differentiation

Approximates the derivative of a function using numerical methods, such as finite differences. This approach is useful when the analytical form of the function is unknown or too complicated to differentiate manually. Despite its practicality, numeric differentiation can suffer from numerical errors and instability, particularly with small step sizes.

1.5.3 Symbolic Differentiation

Computes the exact derivative of a function symbolically using algebraic manipulation software like Mathematica or SymPy. It is suitable for deriving exact analytical derivatives and is valuable in both theoretical and applied mathematics. However, it can be computationally expensive and impractical for very complex functions or high-dimensional problems.

1.5.4 Automatic Differentiation

(AD) is a set of techniques designed to evaluate the derivative of a function specified by a computer program. It leverages the fact that every program, no matter how complex, executes a sequence of elementary operations. AD can be subdivided into forward mode and backward mode.

Forward Mode AD is efficient for functions with a small number of inputs and a large number of outputs. It computes the derivative by propagating the derivatives of the inputs through each operation to the output. This mode is particularly useful in scenarios where the number of independent variables (inputs) is small, though it becomes inefficient for functions with a large number of inputs compared to outputs.

Backward Mode AD (Reverse Mode AD) is efficient for functions with a large number of inputs and a small number of outputs. It computes the derivative by propagating the derivatives of the output backward through the operations to the inputs. This mode is widely used in machine learning for training neural networks due to the high number of parameters (inputs) and typically a single loss value (output). Despite its efficiency in handling numerous parameters, backward mode AD requires more memory to store intermediate values for backpropagation.

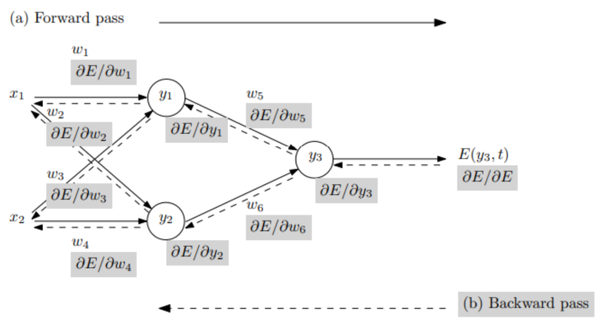

(a) Forward pass
w_1
$\partial E/\partial w_1$
x_1
w_2
$\partial E/\partial w_2$
y_1
$\partial E/\partial y_1$
w_5
$\partial E/\partial w_5$
y_3
$\partial E/\partial y_3$
$E(y_3, t)$
$\partial E/\partial E$
w_3
$\partial E/\partial w_3$
x_2
y_2
$\partial E/\partial y_2$
w_6
$\partial E/\partial w_6$
w_4
$\partial E/\partial w_4$
(b) Backward pass

Figure 1: Overview of backpropagation. (a) Training inputs x_i are fed forward, generating corresponding activations y_i. An error E between the actual output y_3 and the target output t is computed. (b) The error adjoint is propagated backward, giving the gradient with respect to the weights $\nabla w_i E = \left(\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_6}\right)$, which is subsequently used in a gradient-descent procedure. The gradient with respect to inputs $\nabla x_i E$ can be also computed in the same backward pass.

1.6 Jacobian Matrix and Jacobian Vector Product

The **Jacobian Matrix** represents all first-order partial derivatives of a vector-valued function. If $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$, the Jacobian matrix J is an $m \times n$ matrix where each element J_{ij} is the partial derivative of the i-th output with respect to the j-th input.
The **Jacobian Vector Product** (JVP) is the product of the Jacobian matrix and a vector. Forward-mode AD efficiently computes JVP, which is useful for evaluating directional derivatives.

2 Applications and Challenges of AD in Machine Learning

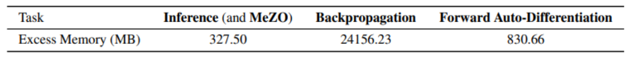
Understanding and selecting the appropriate differentiation method is crucial for optimizing machine learning models. Manual and symbolic differentiation serve educational and theoretical purposes, while numeric differentiation offers practical but approximate solutions. Automatic differentiation, with its forward and backward modes, underpins modern machine learning, facilitating efficient and accurate gradient computation despite its inherent memory and performance challenges.

Automatic differentiation, particularly in its forward and backward modes, is integral to modern machine learning frameworks like PyTorch and TensorFlow. These frameworks leverage AD to compute gradients efficiently, enabling the optimization of complex models such as neural networks. However, implementing AD, especially in large models like BERT, can lead to significant memory overhead. This issue is evident in the JVP operator in PyTorch, which can cause out-of-memory (OOM) errors during profiling.

3 Memory Analysis

Forward auto-differentiation provides significant benefits to estimate gradients for fine-tuning. When $\epsilon \longrightarrow 0$, the Simultaneous Perturbation Stochastic Approximation (SPSA) gradient estimate can be written as $zz^{\top}\nabla L(\theta; B)$. Here, $z^{\top}\nabla L(\theta; B)$ is a Jacobian-vector product (JVP), which can be computed in parallel with a single forward pass, requiring additional memory equivalent to the largest activation in the model.

In accordance with findings from the paper titled 'Automatic Differentiation in Machine Learning: a Survey' authored by Atılım G¨une¸s Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind, and edited by L´eon Bottou, which profiled the memory usage of inference and JVP for RoBERTa-large using JAX with a batch size of 16 on the MultiRC task the following was observed:



Task	**Inference** (and **MeZO**)	**Backpropagation**	**Forward Auto-Differentiation**
Excess Memory (MB)	327.50	24156.23	830.66

Figure 2: Memory consumption of RoBERTa-large when using batch size 16 with the MultiRC task. The reported memory does not include the cost of storing the model on the GPU, which is required for all three cases.

The resulting memory usage during inference, backpropagation, and forward auto-differentiation are summarized in Figure 2. We observed that forward auto-differentiation is substantially more memory-efficient than backpropagation but less memory-efficient than inference.

3.1 Current Results

The current results of our memory analysis reveal significant challenges when using forward automatic differentiation (AD) in PyTorch. Contrary to the expectations set forth in the literature, specifically in the paper by Baydin et al. [4], our experiments indicate that the forward mode AD implementation is highly prone to out-of-memory (OOM) errors.

A typical OOM error encountered during our tests showed that while attempting to allocate 20.00 MiB of memory, the GPU, despite having a total capacity of 11.92 GiB, could not fulfill the request due to fragmented and unallocated memory. This error occurred even though 11.78 GiB of memory was allocated by PyTorch and 6.55 MiB was reserved but unallocated. The error log indicated issues with memory fragmentation and allocation inefficiencies. These results indicate several critical issues:

Firstly, **Memory Fragmentation**: Despite having sufficient overall memory, fragmentation and allocation inefficiencies prevent successful memory allocation for additional operations. This fragmentation leads to situations where there is enough total memory, but it is not contiguous, causing allocation failures.

Secondly, **Garbage Collection**: Attempts to manually force garbage collection were unsuccessful in mitigating the OOM errors. This suggests that PyTorch's garbage collection mechanisms may not be efficiently freeing up memory, contributing to the persistent memory shortages during intensive operations.

Lastly, **Inefficient Memory Management**: The PyTorch profiler's memory allocation and management appear to be suboptimal, leading to excessive memory consumption and subsequent OOM errors. The profiler's overhead exacerbates the memory issues, indicating a need for more efficient memory management strategies within PyTorch to handle large-scale models and operations effectively.

3.2 Real-Time GPU Monitoring

The NVIDIA System Management Interface (nvidia-smi) is a command-line utility, part of the NVIDIA driver, that provides monitoring and management capabilities for NVIDIA GPUs. To monitor GPU usage in real-time, we used the command 'watch -n 1 nvidia-smi', which updates the GPU status every second.

In our experiments, we employed 'watch -n 1 nvidia-smi' to monitor real-time memory usage during both backpropagation and forward mode automatic differentiation (AD) processes. This allowed us to observe dynamic changes in memory allocation and utilization.

From our observations, backpropagation showed consistent memory utilization, with fewer instances of memory fragmentation compared to forward mode AD. Specifically, during backpropagation, the maximum memory usage observed was 2,463 MiB, and the process maintained steady memory usage without significant spikes, indicating efficient memory management.

In contrast, forward mode AD displayed significant memory inefficiencies and fragmentation, leading to out-of-memory (OOM) errors. During forward mode AD, we observed frequent spikes in memory usage, which indicated issues with memory allocation. The maximum memory usage for forward mode AD reached 12,193 MiB, and these fluctuations suggested that PyTorch's forward mode AD implementation might have memory handling issues that require attention.

The real-time monitoring facilitated by 'watch -n 1 nvidia-smi' highlighted the need for better memory management strategies in PyTorch's forward mode AD implementation to mitigate observed inefficiencies and achieve more reliable model training.

4 Objectives

Following the observations from the real-time monitoring, the next steps in our project are:
Explore Memory Allocation Challenges: To identify memory allocation challenges during the use of the JVP operator in PyTorch's profiler forward auto mode differentiation.
Determine Source of Memory Issues: To ascertain whether the memory issues originate from the JVP implementation or PyTorch profiler's memory allocation mechanisms.
Investigate Memory Profiling: To utilize memory profiling tools to pinpoint the root cause of out-of-memory (OOM) errors.

5 Environmental Setup

The experiments were conducted on a high-performance computing system to ensure efficient processing and accurate results. The system was powered by an Intel(R) Xeon(R) CPU E5-2620 v3 running at 2.40GHz, providing robust computational capabilities for the intensive tasks involved in training and profiling large machine learning models. Complementing the CPU, the system featured an NVIDIA GeForce GTX TITAN X GPU with 16GB of VRAM, which is crucial for handling the parallel processing demands of forward automatic differentiation (AD) and backpropagation operations in neural networks.

The operating system used was Ubuntu 20.04.6 LTS, a stable and widely-used Linux distribution known for its performance and compatibility with machine learning frameworks such as PyTorch and JAX. The system was equipped with 251 GiB of RAM, ensuring ample memory for large-scale model training and avoiding potential bottlenecks due to insufficient memory allocation. This setup was essential for conducting thorough memory profiling and addressing the challenges of out-of-memory (OOM) errors during the experiments.

6 Methodologies Employed

In order to reach our goals of exploring memory allocation challenges, determining the source of memory issues, and investigating memory profiling to identify the root cause of out-of-memory (OOM) errors, we tried three different profiling approaches.

6.1 Approach 1: Different Profiling Methods

We employed various profiling methods to monitor and analyze memory usage during our experiments. The methods included:

6.1.1 Scalene Profiler

Scalene is a high-precision CPU, GPU, and memory profiler designed for Python programs. It provides detailed insights into memory allocation and usage patterns. However, we encountered a significant challenge as Scalene is incompatible with multithreaded code, which limited its applicability in our experiments involving complex neural network operations.

6.1.2 Memory Timeline

This method involves visualizing memory usage over time to identify patterns and peaks in memory allocation. The Memory Timeline tool helped us track memory consumption during different phases of model training and profiling, providing valuable insights into when and where memory spikes occurred.

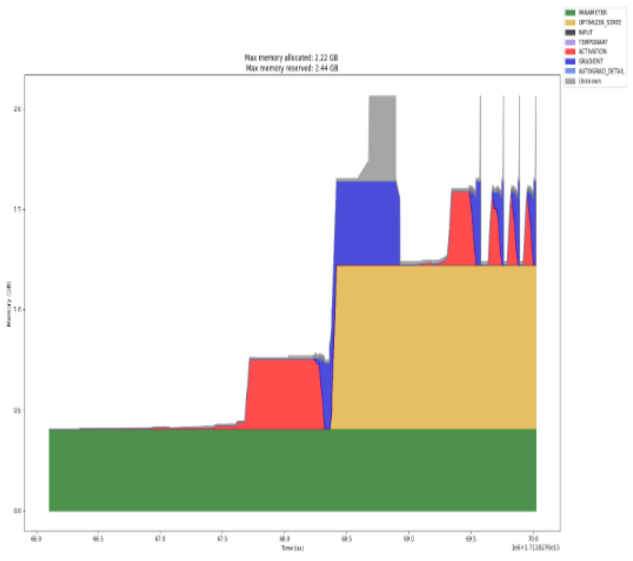


Figure 3: Memory timeline during backward mode AD.

The memory timeline analysis reveals distinct patterns in memory usage during backpropagation and forward mode automatic differentiation (AD). During back propagation (Figure 3), the memory usage increases steadily, reaching a maximum allocation of 11.32 GB, which indicates efficient memory management with minimal fragmentation.

In contrast, the forward mode AD (Figure 4) shows significant fluctuations and spikes in memory usage, peaking at 22.2 GB. This suggests considerable memory fragmentation and inefficient allocation, leading to higher overall memory consumption.

6.1.3 Eventlist Traces

Eventlist Traces gives the stacktrace of function calls over time, allowing us to trace the sequnce of operations in a given program. This method was useful in pin-pointing specific events that might have contributed to OOM errors. By exploring the EventList with Chrome traces, we could visualize the timeline of memory events

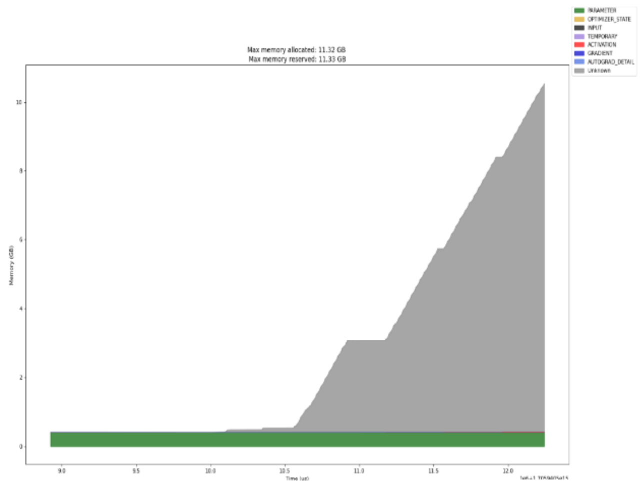


Figure 4: Memory timeline during forward mode AD.

and identify moments where memory spikes or leaks occurred, helping us to pinpoint the underlying causes of these issues being in the jvp() call from pytorch, Suggesting it to be an issue with a combination of jvp operation and pytorch profiler. But it could also be case that jvp operator's memory management might also be the reason for this. This method helped us find that the error of OOM occurs during the execution of jvp() after few epochs (notably 3rd in our experiments).

6.1.4 Memory Snapshot

Taking snapshots of memory usage at specific points in time helped us capture the state of memory allocation during critical stages of our experiments. Memory Snapshots were particularly useful for comparing memory states before and after key operations, helping us understand the impact of specific processes on overall memory usage. The snapshots during backpropagation AD showed a well-managed memory allocation (the timeline trace of memory showed a gradual decrease), while those taken during forward mode AD indicated some cache allocations that were not deleted over time, contributing to memory inefficiencies (the timeline trace of memory shows a consistent increase which is a patterned increase).

Our analysis with these profiling methods revealed several critical insights into memory usage patterns and potential issues. This method showed us that the memory allocations mostly consists of memory allocator function used in pytorch to allocate tensors. This insight implies there is a huge scale memory allocation happening which may be pytorch saving the information for profiling, or jvp saving some information that the profiler is also trying to save. The Scalene profiler, while powerful, was limited by its incompatibility with multithreaded code. The Memory Timeline and Eventlist Traces provided valuable visualizations and logs that helped identify specific points of inefficiency and fragmentation in memory usage. Memory Snapshots offered a detailed comparison of memory states, highlighting areas where memory was not efficiently managed or deallocated.

Overall, these profiling methods revealed the issue being the allocation of memory that's not being cleaned up overtime, this underscored the need for more robust memory management strategies, particularly for forward mode AD in PyTorch. Addressing these issues could lead to significant improvements in the efficiency and reliability of training large and complex neural network models.

6.2 Approach 2: Dual Numbers Implementation in PyTorch

To address the memory allocation challenges observed with the JVP operator, we explored the use of dual numbers in PyTorch's forward mode automatic differentiation (AD). Dual numbers provide a mathematical framework for efficiently computing derivatives, which can be advantageous in reducing memory usage during the training process.

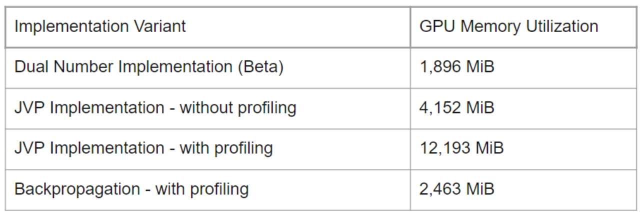
6.2.1 Concept of Dual Numbers

Mathematically, forward mode AD using dual numbers involves evaluating a function using a dual number, which can be defined as a truncated Taylor series of the form:

$$v + \epsilon e,$$

where $v, e \in \mathbb{R}$ and ϵ is a nil potent number such that $\epsilon^2 = 0$ and $\epsilon \neq 0$. This representation allows the tangent value to be carried alongside the primal value, mirroring symbolic differentiation rules. The chain rule applies as expected, facilitating efficient derivative computation.

To provide a quantitative comparison, the following table summarizes the memory usage across different methods.



Implementation Variant	GPU Memory Utilization
Dual Number Implementation (Beta)	1,896 MiB
JVP Implementation - without profiling	4,152 MiB
JVP Implementation - with profiling	12,193 MiB
Backpropagation - with profiling	2,463 MiB

Figure 5: The results indicate significant differences in memory utilization across the different implementations

From these results, several key observations can be made:
The **dual numbers implementation** is highly effective in reducing memory usage during forward mode AD, making it a promising alternative for memory-intensive tasks.
The **JVP implementation**, while useful for specific computations, incurs higher memory costs, especially when profiling is enabled.
Backpropagation, though more efficient than JVP with profiling, does not achieve the same level of memory efficiency as the dual numbers implementation.

6.2.2 Challenges

While the dual numbers implementation has shown promising results in reducing memory usage for simpler neural networks like convolutional neural networks (CNNs), it faces significant challenges when applied to more complex models such as BERT. This limitation is likely due to the current beta status of the dual numbers implementation in PyTorch, which may not yet fully support the intricate architectures and large-scale computations required by transformer models like BERT. As a result, the dual numbers approach, though efficient for simpler models, cannot be reliably used for complex transformer models at this stage.

6.3 Approach 3: Use of JAX

To further investigate and address the memory allocation challenges observed in PyTorch, we explored the use of JAX, a high-performance machine learning library designed for high-speed numerical computing. Despite its advantages, integrating JAX with our existing PyTorch-based workflow presented several challenges.

6.3.1 Challenges

The primary challenges encountered when using JAX included:

- **Library Incompatibility**: PyTorch and JAX libraries are not designed to be used together. This incompatibility required us to implement separate workflows for each library, complicating the experimental setup.
- **Model Compatibility**: We could not directly use the AutoModelForSequenceClassification from the Hugging Face Transformers library in JAX. This limitation necessitated the use of equivalent models in JAX.
- **Documentation and Examples**: There is less documentation and fewer examples available for JAX compared to PyTorch, making it more challenging to implement and troubleshoot.

To address these challenges, we implemented a comparative analysis by developing models using both PyTorch JVP and JAX JVP.

6.3.2 Implementation Details

The experimental setup for our comparative analysis involved developing models using both PyTorch and JAX. Specifically, we used BertForSequenceClassification from PyTorch and FlaxBertForSequenceClassification from JAX. The model checkpoint utilized was `bert-base-uncased`, a widely-used pretrained model for various natural language processing tasks.

For the training process, we employed two different methods: Backpropagation and Forward Mode AD (with JVP). Each model was trained for a total of 5 epochs. The training dataset comprised 1,000 samples, while the testing dataset included 100 samples. We used a batch size of 8 examples per batch to balance computational efficiency and performance.

The AG News dataset, which is a standard benchmark for text classification, was used for the classification tasks.

6.3.3 Observations

- **Peak Memory Usage**: The peak memory usage for PyTorch JVP was significantly higher (4,488 MiB) compared to PyTorch backpropagation (2,458 MiB). This indicates that forward mode AD with JVP in PyTorch incurs a substantial memory overhead.
- **Training Time**: The training time for PyTorch JVP (89.67 s) was also longer than for PyTorch backpropagation (69.35 s), suggesting additional computational complexity in JVP.

- **Challenges with JAX**: Due to the integration challenges and limited documentation, the results for JAX backpropagation and JVP were not available. Further work is required to establish a stable and efficient JAX workflow for complex models like BERT.

The use of JAX presents a promising direction for addressing memory allocation challenges in forward mode AD. However, significant hurdles remain, particularly in terms of library compatibility and model implementation. Future work will focus on overcoming these challenges, improving documentation and examples for JAX, and conducting comprehensive comparisons across different frameworks to optimize memory and computational efficiency.

7 Future Work

Based on the observations and challenges encountered during our experiments, several areas of future work have been identified to further investigate and address memory issues in forward mode automatic differentiation (AD).

7.1 Exploring `jax.jacfwd` Implementation

One of the key areas of future work involves exploring the `jax.jacfwd` implementation in JAX. The `jax.jacfwd` function is designed to compute forward-mode Jacobian-vector products efficiently. By delving deeper into this implementation, we aim to:

- **Evaluate Efficiency**: Assess the memory and computational efficiency of `jax.jacfwd` compared to PyTorch's forward mode AD.
- **Model Compatibility**: Test the compatibility of `jax.jacfwd` with complex models like BERT, ensuring it can handle large-scale computations without significant memory overhead.
- **Performance Benchmarking**: Perform comprehensive benchmarking to compare the performance of `jax.jacfwd` with other differentiation techniques.

7.2 Investigating JVP Implementation in PyTorch and JAX

Another critical area of future work is a detailed investigation of the JVP implementation in both PyTorch and JAX. The goals of this investigation include:

- **Memory Usage Analysis**: Thoroughly analyze memory allocation patterns and identify potential inefficiencies in the JVP implementation.
- **Optimization Strategies**: Develop and test optimization strategies to reduce memory usage during JVP computations.
- **Implementation Improvements**: Collaborate with the PyTorch and JAX communities to contribute improvements to the JVP implementation, making it more robust and memory-efficient.

7.3 Checking for Memory Leaks in PyTorch Profiler

Given the significant memory overhead observed with PyTorch profiling, it is essential to investigate whether the PyTorch Profiler has any memory leaks. This investigation will involve:

- **Profiling Tools Evaluation**: Use various profiling tools to monitor memory usage and detect potential leaks during the training process.
- **Codebase Review**: Conduct a detailed review of the PyTorch Profiler codebase to identify and rectify any memory management issues.
- **Patch Development**: Develop patches or recommend changes to the PyTorch Profiler to address identified memory leaks and improve its overall efficiency.

7.4 Implementing Enhanced Memory Management Techniques

To mitigate the memory challenges faced, future work will also focus on implementing and testing advanced memory management techniques, including:

- **Garbage Collection Optimization**: Enhance garbage collection processes to ensure timely deallocation of unused memory, reducing the risk of out-of-memory (OOM) errors.
- **Memory Pooling**: Implement memory pooling strategies to reuse memory blocks effectively, minimizing fragmentation and improving allocation efficiency.
- **Memory Profiling and Monitoring**: Develop more sophisticated memory profiling and monitoring tools to provide real-time insights into memory usage and identify bottlenecks proactively.

8 Findings

Our investigation into the memory usage and performance of different automatic differentiation techniques in PyTorch revealed several key findings.

Firstly, we observed that the 'JVP()' function in PyTorch consumes significantly more memory than backpropagation during model training. This increased memory usage poses a critical issue, especially for training large models. In contrast, 'JVP()' in evaluation mode ('eval()') takes less memory, likely because, in evaluation mode, PyTorch does not forcefully store the attention matrices that are otherwise needed during training. However, even in evaluation mode, the 'JVP()' function exhibits signs of memory handling issues, indicating potential problems with memory management in its implementation in PyTorch.

Additionally, the occurrence of out-of-memory (OOM) errors when using the PyTorch profiler suggests a possible data leak. Although we have not yet thoroughly explored this area, the profiler's significant memory overhead points to inefficiencies or leaks in how memory is being handled and allocated during profiling.

Moreover, the beta implementation of dual numbers in PyTorch has shown promising results for simpler neural networks, such as convolutional neural networks (CNNs). However, it fails to train more complex models like BERT. This limitation suggests that the beta version of the dual numbers implementation does not yet support the handling of parameters in complex networks effectively. The challenges associated with parameter handling in these networks may be more intricate, requiring more robust support and optimizations in future versions.

These findings highlight critical areas for further investigation and improvement in the implementation and memory management of automatic differentiation techniques in PyTorch. Addressing these issues will be essential for optimizing the performance and reliability of training large and complex models.

Acknowledgments

We would like to thank Professor Hui Guan from the Computer Science Department at the University of Massachusetts, Amherst, for their invaluable guidance and support throughout this project. Their insights and feedback were instrumental in the successful completion of this research. Additionally, we appreciate the resources and facilities provided by the university that enabled us to conduct our experiments.

We also extend our gratitude to Kunjal Panchal, a PhD student from the Computer Science Department at the University of Massachusetts, Amherst, whose current research project provided a foundation for our work. Her assistance and expertise were crucial in navigating the challenges we encountered during this study.

References

[1] Mengwei Xu, Dongqi Cai, Yaozong Wu, Xiang Li, and Shangguang Wang. FwdLLM: Efficient FedLLM using Forward Gradient. In *Proceedings of the Beijing University of Posts and Telecommunications*, 2021.

[2] Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D. Lee, Danqi Chen, and Sanjeev Arora. Fine-Tuning Language Models with Just Forward Passes. In *arXiv preprint arXiv:2205.12345*, 2022. Available at {smalladi, tianyug, eshnich, ad27, jasonlee, danqic, arora}@princeton.edu.

[3] Nicholas W. Brady, Maarten Mees, Philippe M. Vereecken, and Mohammadhosein Safari. Implementation of Dual Number Automatic Differentiation with John Newman's BAND Algorithm. *Journal of The Electrochemical Society*, 168(11):113501, 2021. Published on behalf of The Electrochemical Society by IOP Publishing Limited.

[4] Atılım Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic Differentiation in Machine Learning: a Survey. *Journal of Machine Learning Research*, 18(153):1-43, 2018.