Differentiable programming in machine learning

MARIJA A. KOSTIĆ, University of Belgrade,
School of Electrical Engineering, Belgrad

DRAŽEN D. DRAŠKOVIĆ, University of Belgrade,
School of Electrical Engineering, Belgrade

This paper explains automatic differentiation, discussing two primary modes—forward and backward—and their respective implementation methods. In the context of issues encountered in machine learning and deep learning, the forward mode is deemed more suitable as it efficiently differentiates functions with numerous inputs compared to outputs. Given Python's pivotal role in the ML landscape, the paper elaborates on two widely used deep learning libraries—PyTorch and TensorFlow. While both these libraries support automatic differentiation, they adopt distinct approaches, each carrying its unique strengths and weaknesses.

Key Words: programming paradigms, machine learning, deep learning, frameworks

1. INTRODUCTION

Early electronic computers were colossal devices that took up multiple rooms, demanded substantial amounts of power, and carried a hefty price tag in the millions of dollars. Those responsible for programming these machines were required to work with machine language, a series of bits that directly commands and controls a processor's operations. Of course, this way of working was very tedious and infeasible for larger programs and projects. Consequently, assembly languages were invented. They allowed expression of processor operations with mnemonic abbreviations. Every computer necessitated programming in its distinct assembly language. With the further development of computers people started thinking about machine independent language. The mid-1950s brought about the first high-level programming language, Fortran [1].

Programming languages offer abstractions, fundamental principles, and control structures that programmers utilize to create well-designed programs [2]. Today, there are thousands of high-level programming languages with new ones constantly emerging. According to [1] there are three reasons for this: 1) evolution of computer science; 2) special purposes which require design of specific languages; and 3) personal preference of people. One such special purpose that is becoming truly relevant and popular in recent years is machine learning (ML).

Nowadays, ML powers many aspects of modern society and drives significant advances in many different fields. The achievement can be credited to the creation of more advanced ML models, the accessibility of extensive datasets essential for training, and the availability of significant computational power required for these tasks. Advances in hardware further encouraged the development of new software libraries, programming languages, compiler technologies, etc. in order to exploit it [3].

Traditional ML methods often possess a restricted capability to handle natural data in its raw, unprocessed form. They require considerable engineering work and domain expertise to design features relevant for the problem being solved. Deep learning (DL), a subset of ML techniques, is gaining popularity due to its minimal requirement for manual engineering [3].

DL models consist of numerous layers that progressively learn intricate data representations through multiple levels of abstraction. They are usually implemented like neural networks. The loss function, which measures the error of predictions, is a function of weights which define the network. The optimization of this function involves the utilization of gradient-based optimization techniques, necessitating iterative computations of derivatives [3]. In recent times, several DL frameworks have emerged, including TensorFlow [4],
Theano [5], and PyTorch [6]. These frameworks employ automatic differentiation (AD) and are representatives of a programming paradigm called differentiable programming. In this paradigm, derivatives of numerical functions defined programmatically are calculated automatically [7].

The second section gives an overview of the standard programming paradigms. The third section describes AD, its two main modes of operation and standard implementation approaches. The fourth section gives examples of AD usage in ML inside the Python ecosystem. Last section concludes this paper.

2. PROGRAMMING PARADIGMS

Programming languages can be classified into groups, called programming paradigms, based on their shared concepts, features, and model of computation. Within the same language several paradigms may be supported [1, 8]. Two primary approaches exist: declarative, emphasizing what the computer should execute, and imperative, focusing on how the execution should take place.

Within the imperative programming paradigm, programmers direct the machine on how to alter its state. This paradigm, rooted in the classic „Von Neumann” model of computation, relies on modifying variables stored alongside the program as the fundamental method of computation. Imperative programs contain series of assignments, commands that perform calculations, control constructs, and I/O operations [9, 1].

Natural abstraction in imperative programming is the procedure. In procedural programming a program is built from one or more procedures or functions. A potentially complex set of commands and control structures is encapsulated in a manner that allows it to be regarded as a singular entity or unit. This collection represents the implementation of a function – function body. Functions can be parametrized by the function caller [2]. This model allows the programmer to focus on the function interface rather than its implementation details. First procedural languages appeared in 1960s including Fortran, COBOL and BASIC. Latter appeared Pascal and C (1970s).

An extension of procedural abstraction is data abstraction. Programmers could define their own new data types by grouping related constants, variables, functions etc. One example of this are classes, main building blocks for object-oriented languages. Every occurrence or instance of a class represents an object. Within the object-oriented programming model, the program is structured as a gathering of objects that communicate by exchanging messages, thereby altering their respective states. Each object "hides" its own internal state which can be manage via specific interface i.e., object’s member functions [1, 9].

Object orientation in programming languages has become more popular, and widely accepted in the industry. Because of that many languages are said to be object-oriented. According to [2] a programming language is object-oriented if it has objects and supports following four features:

- Dynamic lookup – when a message is sent to an object, the object itself "chooses" how to respond. Code that is to be executed is not determined by some static property.
- Abstraction – implementation details are hidden behind a specific interface. Manipulation of object’s hidden data usually goes through its set of public functions.
- Subtyping – subtype objects fulfill all functionalities of supertype objects and can always be used as their substitute.
- Inheritance – ability to define new class of objects by reusing the definition of the existing class of object. Inheritance is a relation on implementations, while the subtyping is relation on interfaces [2].

Simula is widely recognized as the initial language incorporating the fundamental characteristics of an object-oriented (OO) programming language. Numerous popular languages in use today are multi-paradigmatic, embracing varying degrees of support for object-oriented (OO) programming, often in conjunction with imperative, procedural, and functional programming paradigms. Significant OO languages include C++, Java, Python, C#, Scala, Ruby, JavaScript etc. [2].

Within the declarative programming paradigm, the programmer specifies the properties or characteristics of the desired outcome without explicitly detailing how the computation should be performed [9]. A declarative program is viewed as a formula about which one can reason without any reference to the underlying implementation or computational mechanisms [2].

Functional programming conceptualizes a problem as an assembly of mathematical functions, where each function maps inputs to corresponding outputs. It is founded on lambda calculus developed by Church [10]. Unlike in imperative programming, there is no notion of state, which means that assignment statements make no sense in functional programming. It is impossible to change any value, but it is possible to make its revised copy. Thus, the effect of a loop is achieved via recursion. Functions in functional programming languages interact through functional composition, conditionals, and recursion [9]. In these
languages, functions are considered first-class citizens, allowing them to be assigned to variables, passed as arguments, and returned as values. Several functional languages, such as Lisp, Clojure, OCaml, Haskell, and F#, are widely used in both industry and education. Moreover, several other programming languages, including C++, Java, C#, Python, Rust, Scala, and Perl, offer support for functional programming paradigms alongside their primary features.

The logic programming paradigm differs substantially from the other programming paradigms. Languages based on logic programming offer an alternative perspective on problem-solving methodologies. Logic or constraint-based programs specify goals of the computations. These goals are expressed as a set of rules, or assertions about the constraints and solutions of the problem. There is no sequence of commands to be conducted. Logic programming finds application in two primary domains: artificial intelligence and database information retrieval [9]. One of the first logic programming languages was Prolog, initially developed for application in natural language processing. Soon after, it was realized that logic programming could serve as a versatile, general-purpose programming language [11].

Dataflow programming is a paradigm where a program is represented as a directed graph, illustrating the flow of information (tokens) between operations (nodes). An operation initiates execution as soon as all its required inputs become accessible. This means that dataflow languages are inherently parallel [1]. Dataflow programming languages exhibit certain similarities with functional languages. Examples include the following programming languages: Id, Val, and Sisal.

Besides declarative and imperative programming, there are other paradigms, such as: concurrent programming, event-driven programming, scripting, array programming and differentiable programming.

Concurrent programming enables multiple tasks to progress simultaneously at potentially unpredictable points during their execution. Unlike sequential programs that have a single active execution context, concurrent programs may have more than one. The distinction between sequential and concurrent execution is mostly independent of the previous paradigm classification. Some languages have explicitly concurrent features (e.g., Java, C++, C#), while in others concurrent programming is supported through special libraries or compilers (e.g., Fortran, C) [2].

Event-driven programs are designed to respond to a wide range of conceivable event sequences. This sequence is not governed by the program but by the input data. In event-driven programs generally there is a main loop that listens for the events and then triggers appropriate call back functions. An extensively known instance of an event-driven program is an graphical user interface (GUI), wherein the program's progression relies on events such as mouse clicks, movements, scrolling, key presses, and releases, among others [9]. Event-driven programming can be written in any programming language, with different levels of difficulty depending on the high-level abstractions that are provided by the language.

Scripting languages are high-level programming languages which execute many common programming processes at runtime instead of during compilation. Traditional languages prioritize aspects like efficiency, maintainability, portability, and static error detection. Conversely, scripting languages emphasize flexibility, swift development, local customization, and dynamic (run-time) error checking.

They are sometimes called glue languages because of their ability to coordinate components drawn from different programs. Scripting languages commonly share several characteristics, including the ability to function in both batch and interactive modes, concise expression, absence of explicit declarations, flexible dynamic typing, convenient access to system resources, advanced capabilities for pattern matching and string manipulation, as well as support for high-level data types. General-purpose programming languages include Perl, Python, and Ruby [1].

Array programming paradigm generalizes certain operations so that they can be applied to an entire set of values (array) at once instead of individual values one at a time. Here, it is assumed that all those values (elements of the array) possess similar properties and can be processed in the same manner.

Languages supporting array programming possess the capability to extend operations initially performed on individual elements (scalars) to seamlessly apply to vectors, matrices, and higher-dimensional arrays, allowing for vectorized operations [12]. Some examples are: MATLAB, Octave, R, Julia etc. Other languages support array programming through specialized libraries (NumPy in Python [13], and Eigen [14] at C++).

Differentiable programming constitutes a programming paradigm wherein entire programs possess the capability of being differentiated entirely. This concept is typically achieved through the application of AD, which encompasses a range of techniques used to differentiate numeric functions expressed as computer programs both efficiently and accurately. Differentiable programming languages integrate AD as a first-class feature [15].
3. AUTOMATIC DIFFERENTIATION

Derivatives hold significant importance in various domains such as sensitivity analysis (for model validation), solving inverse problems (like data assimilation), and optimizing designs (including the selection of simulation parameters) [16]. Additionally, derivatives play a pivotal role in computational statistics, for instance, in Bayesian inference utilizing Hamiltonian Monte Carlo sampling techniques, as well as in the training process of neural networks [17]. The techniques used for computing derivatives can be categorized into four main groups [7]:

- Manual differentiation involves the process of manually deriving derivatives and then coding them into the program. Earlier machine learning algorithms relied on computing gradients and Hessians of an objective function. For the newly introduced models, researchers spend a considerable time and effort on the manual derivation [7] (Figure 1, top right).

- Numerical derivation uses finite difference approximations. In its basic form, it follows the limit definition of the derivative \( \frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \). The implementation of manual differentiation is straightforward, yet it suffers from high inaccuracy stemming from round-off and truncation errors, primarily due to limited precision in computations and the selected value of step h. While these errors might be acceptable in some DL scenarios, the method’s linear complexity for gradient differentiation in n dimensions makes it unsuitable for ML and DL models with millions or billions of parameters [7]. Numerical differentiation serves as a useful tool for rapidly verifying the accuracy of computed gradients, especially in cases involving manual differentiation or when utilizing a new or unfamiliar library for differentiation [18] (Figure 1, bottom right).

- Symbolic differentiation involves the automatic manipulation of expressions through transformations that embody differentiation rules, essentially automating the process of manual differentiation. Advanced computer algebra systems like Mathematica, Maxima, and Maple offer support for symbolic differentiation. Its notable advantage lies in generating comprehensible mathematical expressions that can be easily understood and analyzed. On the other hand, there are significant shortcomings. One of them is the problem of expression swell where naïve application of symbol rewriting rules can lead to an explosion of symbolic terms within complex and cryptic expressions. Thus, the runtime calculation of derivative values is inefficient and computationally infeasible. Another limitation of symbolic differentiation is its reliance on predefined rules, which might not encompass the complexity of defining certain intricate functions. Because symbolic differentiation is differentiating mathematical expressions and not a given procedure it is not possible to use statements like if-else statements or for/while loops [7, 18] (Figure 1, center right).

- Automatic differentiation, also known as algorithmic differentiation, comprises a set of methods used to compute derivatives of a function defined by a computer program. Unlike symbolic differentiation, which produces derivative expressions, AD computes derivatives by accumulating values throughout code execution, generating numerical derivative evaluations [19, 7] (Figure 1, bottom left).

AD encompasses a range of techniques designed for the efficient and precise computation of derivatives for numeric functions expressed within computer programs. It involves a non-standard interpretation of the provided computer program, integrating derivative values and adjusting the semantics of operators to propagate derivatives based on the chain rule of differential calculus [7].

AD shares certain similarities with both numerical and symbolic differentiation. However, it differs from classic symbolic differentiation in that it does not generate symbolic representations of derivatives. Moreover, it does not rely on finite difference approximation methods typical in numerical differentiation. Instead, AD leverages the chain rule from calculus to generate numerical values of derivatives. This is achieved by accumulating values computed during the execution of program instructions [16]. The advantage of AD over other methods lies in its ability to precisely compute derivatives at machine precision while incurring only a minor constant overhead. Additionally, AD can be implemented in regular code with minimal modifications. Unlike symbolic differentiation, which demands organizing code into closed-form expressions, AD accommodates branching, loops, and recursion [7, 20]. Due to its versatility, AD has found extensive application in various industries and academic fields, encompassing computer simulations in engineering, design optimization, computational finance, atmospheric sciences, structural mechanics, optimal control, and many others [7].

3.1. Main modes of automatic differentiation

AD uniquely interprets a computer program by enhancing the code to compute derivatives concurrently with standard computations. This process
Differential programming in machine learning involves breaking down the program into primitive operations (such as unary and binary operations, transcendental functions, etc.) that possess readily available derivatives. The derivatives of the original program are then obtained through the application of the chain rule from calculus, which integrates the derivatives of these primitive operations [7]. This computation process can be represented with evaluation trace, also called a Wengert list [21]. Computational graph is a useful visual representation of the given trace. It clearly shows dependencies between intermediate variables. Figure 2 shows computational graph for the example \( f(x_1, x_2) = \ln x_1 + x_1 x_2 - \sin x_2 \). It’s crucial to reiterate that AD has the capability to differentiate not only closed-form expressions but also code employing control structures like branching, recursion, loops, and procedure calls. This versatility stems from the fact that any code execution eventually generates a numeric evaluation trace comprising specific input, intermediate, and output variables. Unlike storing and manipulating intermediate symbolic forms, AD instantly computes these values for given input, effectively addressing the problem of expression swell. Moreover, the specific control path taken during execution doesn’t impact the computation of derivatives [7, 18]. AD operates in two primary modes: forward mode (left to right) and reverse mode (right to left). These modes correspond to the direction within the computation graph where the chain rule is evaluated.

Forward accumulation mode, also known as tangent linear mode, is conceptually most simple type. Computations of derivatives and intermediate computations of the original program are calculated in parallel. Considering the derivative of the example function \( f(x_1, x_2) = \ln x_1 + x_1 x_2 - \sin x_2 \) with respect to \( x_1 \), in each step, next to intermediate variable \( v_1 \), a derivative \( \frac{\partial v_1}{\partial x_1} \) is calculated as well (Table 1).

By following chain rule, for each elementary operation in the forward primal trace (left-hand side in Table 1), the corresponding tangent (derivative) trace is generated (right-hand side in Table 1). This is done in a single pass over the program. Forward mode can be extended to support functions with multiple inputs and multiple outputs. Functions \( f: \mathbb{R} \to \mathbb{R}^m \) are differentiated efficiently in just one forward pass. On the other hand, functions \( f: \mathbb{R}^n \to \mathbb{R} \) require \( n \) evaluations. This is the reason why in general case \( f: \mathbb{R}^n \to \mathbb{R}^m \) where \( n \gg m \), this technique is not preferred [7].

In Forward mode AD, there are constant memory demands and computational complexity that scales in proportion to the count of input variables. The backpropagation algorithm is extensively utilized in ML, particularly in training neural networks. It calculates the gradients of the loss function concerning the network’s weights while searching for the minima. Iterating backward, from the last layer to the first layer of the network, gradients with respect to each weight are computed using the chain rule [7].

Figure 1 - Four differentiation approaches applied to the same mathematical expression

---

**Numerical Differentiation**

\[ f'(x) : \]
\[ h = 0.000001 \]
\[ \text{return} \ (f(x+h) - f(x)) / h \]

**Automatic Differentiation**

\[ f'(x) : \]
\[ (v, dv) = (x, 1) \]
\[ \text{for} \ i = 1 \text{ to } 3 \]
\[ 8*v*dv \]

**Symbolic Differentiation of the closed-from**

\[ f'(x) : \]
\[ \text{return} \ 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2) \]
\[ + 64(1-x)(1-2x)^2(1-8x+8x^2)^2 \]
\[ - 64x(1-2x)^3(1-8x+8x^2)^2 \]
\[ - 256x(1-x)(1-2x)(1-8x+8x^2)^2 \]

**Manual Differentiation**

\[ f(x) : \]
\[ v = x \]
\[ \text{for} \ i = 1 \text{ to } 3 \]
\[ v = 4*v*(1-v) \]
\[ \text{return} \ v \]

or, in closed form,

\[ f(x) : \]
\[ \text{return} \ 64*x*(1-x)*(1-2x)^2 \]

---

Figure 1 - Four differentiation approaches applied to the same mathematical expression

---

TEHNIKA – ELEKTROTEHNIKA 72 (2023) 6 703
Table 1. Forward mode automatic differentiation of \( y = f(x_1, x_2) = \ln x_1 + x_1 x_2 - \sin x_2 \) with respect to \( x_1 \) evaluated at point \( (x_1, x_2) = (2, 5) \). On the left is presented original evaluation trace, and on the right are show tangent operations that augment corresponding original operations [7].

<table>
<thead>
<tr>
<th>Forward Primal Trace</th>
<th>Forward Tangent (Derivative) Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_{-1} = x_1 )</td>
<td>( v_{-1} = x_1 )</td>
</tr>
<tr>
<td>( v_0 = x_2 )</td>
<td>( v_0 = x_2 )</td>
</tr>
<tr>
<td>( v_1 = \ln v_{-1} )</td>
<td>( v_1 = \frac{v_{-1}}{v_{-1}} ) i</td>
</tr>
<tr>
<td>( v_2 = v_{-1} + v_0 )</td>
<td>( v_2 = v_{-1} + v_0 + v_0 + v_{-1} )</td>
</tr>
<tr>
<td>( v_3 = \sin v_0 )</td>
<td>( v_3 = v_0 \cos v_0 )</td>
</tr>
<tr>
<td>( v_4 = v_1 + v_2 )</td>
<td>( v_4 = v_1 + v_2 )</td>
</tr>
<tr>
<td>( \Delta y = v_5 )</td>
<td>( \Delta y = v_5 )</td>
</tr>
</tbody>
</table>

The reverse accumulation mode is also called the \textit{adjoint} or \textit{cotangent linear} mode. It aligns with the generalized backpropagation algorithm. In this mode, the chain rule is assessed in the reverse order of the original program’s execution flow. Each intermediate variable \( v_i \) is complemented with an adjoint \( \bar{v}_i = \frac{\partial y}{\partial v_i} \) which represents the sensitivity of a considered output \( y \) with respect to the changes in \( v_i \) [7].

In the reverse mode AD, the differentiation process comprises two distinct phases. In the first phase, the original program code (primal program) is run \textit{forward}. Intermediate variables \( v_i \) (left-hand side in Table 2) are populated, and dependencies in the computational graph are recorded. In the second phase, derivatives are calculated by propagating adjoints \( \bar{v}_i \) (right-hand side in Table 2) in the \textit{reverse} order, from outputs to the inputs [7]. For efficient execution, every statement in the adjoint program must have access to the intermediate variables of the original program. Thus, the AD transformation needs to ensure that these intermediate variables are not destroyed or mutated [19].

Figure 2 - Computation graph for the example \( f(x_1, x_2) = \ln x_1 + x_1 x_2 - \sin x_2 \) [7].

An important advantage of reverse mode over forward mode is that it has computation complexity proportional to the number of output variables. This means that in the extreme case \( f: \mathbb{R}^n \to \mathbb{R} \) would require only one application of the reverse mode compared with the \( n \) passes of the forward mode. This is especially relevant for applications of AD in ML where usually there is one error function that is optimized with respect to a large number of parameters.

Thus, reverse mode in the form of backpropagation algorithm is the mainstream technique for optimization of ML models. However, reverse mode has increased storage requirements because all intermediate variables of the primal program need to be saved until adjoint calculations are completed. The memory demands are directly proportional to the quantity of intermediate variables, namely, the operations within the evaluated function [7].

Regardless of the specifics of the implementation, AD requires careful memory management, particularly in reverse mode. In forward mode, one pass through the computation graph computes variables of the original program and their derivatives.

Thus, memory requirement is just twice the requirement for the evaluation of the original functions. Moreover, intermediate variables need to be stored temporarily, only while there exist other variables connected to them that have not been evaluated.

On the other hand, the reverse mode is more complicated because it is necessary to access the computation graph and numerical values of all intermediate variables in order to perform reverse trace. These must be stored in a persistent memory called \textit{tape}. The memory requirements are dynamic, as differentiated programs include loops and conditional statements [17].

3.2. Implementation

The two AD modes offer an understanding of the operations required for a modified program to accurately compute the correct derivatives. Manual transformations are time consuming and prone to errors. Managing these modes can be challenging, especially when the underlying code undergoes frequent alterations.

Hence, automating the code transformation process, even partially, is highly desirable. However, any degree of automation involves a trade-off between the complexity of the transformation process and the
efficiency in terms of space and time of the resulting modified program [16]. AD influences and interacts with the complete toolchain, starting from language design and spanning across intermediate representations, static analysis, to code generation, and ultimately, program execution [19].

Table 2. Reverse mode automatic differentiation of \( y = f(x_1, x_2) = \ln x_1 + x_1 x_2 - \sin x_2 \) with respect to both \( x_1 \) and \( x_2 \) evaluated at point \((x_1, x_2) = (2, 5)\). On the left is presented original evaluation trace, and on the right are shown cotangent operations that augment corresponding original operations [7].

<table>
<thead>
<tr>
<th>Forward Primal Trace</th>
<th>▲Reverse Adjoint (Derivative) Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_{-1} = x_1 ) = 2</td>
<td>( x_1 = \tilde{v}_{-1} ) = 5.5</td>
</tr>
<tr>
<td>( v_{0} = x_2 ) = 5</td>
<td>( x_2 = \tilde{v}_{0} ) = 1.716</td>
</tr>
<tr>
<td>( v_1 = \ln v_{-1} = \ln 2 )</td>
<td>( \frac{\partial v_1}{\partial v_{-1}} = \frac{v_{-1}}{v_{-1}} = 5.5 )</td>
</tr>
<tr>
<td>( v_2 = v_{1} \times v_{0} = 2 \times 5 )</td>
<td>( \frac{\partial v_2}{\partial v_{0}} = \tilde{v}<em>2 \times v</em>{0} = 0.963 + 10 )</td>
</tr>
<tr>
<td>( v_3 = \sin v_{0} = \sin 5 )</td>
<td>( v_0 = \tilde{v}<em>0 \times \frac{\partial v_0}{\partial v</em>{0}} = 5 )</td>
</tr>
<tr>
<td>( v_4 = v_{1} + v_{2} = 0.963 + 10 )</td>
<td>( \tilde{v}_3 = \tilde{v}<em>3 \times \frac{\partial v_3}{\partial v</em>{0}} = -0.284 )</td>
</tr>
<tr>
<td>( v_5 = v_{4} - v_{3} = 10.693 + 0.959 )</td>
<td>( \tilde{v}_4 = \tilde{v}<em>4 \times \frac{\partial v_4}{\partial v</em>{1}} = 1 )</td>
</tr>
<tr>
<td>( y = v_{5} = 11.652 )</td>
<td>( \tilde{v}_5 = \tilde{v}<em>5 \times \frac{\partial v_5}{\partial v</em>{3}} = 1 )</td>
</tr>
</tbody>
</table>

In real-world scenarios, a function \( ff \) written in a chosen programming language is supplied to an AD tool. This tool enhances the program by incorporating additional code to manage dual operations essential for computing both the function and its derivative concurrently. Operator overloading and source transformation are the fundamental computer science concepts used for this purpose [16].

Operator overloading (OOL) in modern programming languages offers the most direct method for implementing AD by redefining the semantics of elementary operations. All primitive operations are overloaded so that they additionally perform tracing operations. For each operation executed in the primal program its inputs and results are saved on a tape. At the end, the tape contains a linear trace of all numerical operations in the program. The intermediate values from the tape are then processed by the adjoint program in reverse in order to calculate the derivatives [19].

Operator overloading has several advantages. It is straightforward to implement. It is also easy to use because developers and researchers can implement their models as regular programs with arbitrary control flow statements. A notable drawback is that the adjoint program necessitates a distinct derivative interpreter, and if embedded within the host language, it can complicate debugging and performance analysis. Another disadvantage involves the overhead incurred with each function call. This can pose a challenge, especially if higher-level operations are considerably slower compared to primitive operations. Additionally, in the adjoint program, OOL does not allow for ahead-of-time optimization [19].

AD implemented with OO approaches can be found in various frameworks. In the ML domain, examples include PyTorch [22], Autograd [23], JAX [24], and Chainer [25]. Additionally, non-machine learning-oriented AD frameworks utilizing OO methodologies encompass ADOL-C [26] and CppAD [27].

Source transformation represents a more labor-intensive approach to building an AD tool. Its complexity is comparable to that of developing a compiler [16].

In operator overloading, the adjoint program is dynamically constructed each time during execution. However, with source transformation (ST), this isn't the case. ST requires the user to define a computation graph using a domain-specific language (DSL), which could be an extension of an existing programming language or a completely new language. Through
During execution, the framework interprets the program with various inputs while maintaining a fixed computation graph. This highlights the advantage of source transformation over operator overloading: AD transformation is conducted only once per program, avoiding overhead during runtime. Additionally, the adjoint program is accessible during compilation, allowing optimization ahead of time. Furthermore, there is the potential to leverage parallelization opportunities presented by the computation graph, like in dataflow programming. However, source transformation approach imposes limitations on the user. Some of them are limiting use of loops, recursion, and higher order functions. Another downside is that ST produces unintuitive control flow which can be difficult to debug [19].

Despite not handling AD transformations during runtime, ST needs to guarantee access to intermediate variables from the forward pass by the adjoint program. Similar to operator overloading (OOL), ST can achieve this by saving intermediate variables either on the tape or directly within the computation graph [19]. Widely used machine learning frameworks like TensorFlow [4], Theano [5], and MXNet [28] adhere to the dataflow paradigm, utilizing computation graphs as their intermediate representations. AD is implemented using the approach of source transformation.

4. AUTOMATIC DIFFERENTIATION AND MACHINE LEARNING

Many of ML models rely on gradient-based optimization methods that necessitate recurrently computing derivatives to fit to a dataset. This becomes particularly relevant in DL, where intricate neural network models are constructed, comprising complex mathematical expressions operating on matrices and tensors. Typically, the loss function in DL is a scalar computation that gauges the prediction error of the network concerning the corresponding labels. This loss function depends on the weights that define the network, which can amount to millions. Furthermore, modern ML development is based on the ability to quickly iterate over new models and aggressively exploit computational resources through parallelism, low interpreter overhead, distributed computing etc. [19].

Developing a DL model requires connection and orchestration of a lot of different pieces. High level of abstraction can support easier, faster and quality solutions by doing the heavy lifting tasks. This allows for researchers and developers to focus on tasks that matter rather than spending their time on fundamental operations. DL frameworks such as popular Keras [29], PyTorch [6], TensorFlow [4], CNTK [30], provide this king of abstraction [18].

ML and DL models are naturally expressed in terms of tensors, matrices, and higher-order arrays. Moreover, modern GPUs and TPUs (Tensor Processing Units) are also highly optimized for tensor operations. All this has led to the implementation of modern ML frameworks in terms of tensor operations. In many cases, these frameworks are implemented in faster languages such as C++, with primary APIs accessible through Python. This setup is designed to minimize slower computations in Python and reduce the overhead of communication between Python and C++ for tensor operations [31]. According to reference [6], four trends in scientific computing have gained increasing significance in the realm of DL:

- Array-based programming – the development of DSLs turned multidimensional arrays (tensors) into first-class objects that can be manipulated by language operators. Furthermore, specialized libraries like NumPy [13], Eigen [14], and Torch [32] have facilitated array programming within general-purpose languages like Python and C++.
- Automatic differentiation – the development of AD [7] has considerably simplified the exploration of diverse ML approaches, all while retaining the efficiency and precision of gradient-based optimization. The burden of calculating derivatives is removed from the developers and the researchers.
- Open-source Python ecosystem – Scientific community moved from proprietary software (e.g., MATLAB) towards open-source Python ecosystem with a vast repository of libraries and datasets. Even though there are open-source libraries for ML in other programming languages (e.g., List, C++, Lua), because of the network effect of a large Python ecosystem, most ML and DL frameworks converged to Python interface.
- Hardware accelerators – The availability and widespread adoption of parallel hardware, such as GPUs, have supplied the essential computational power needed for the efficient execution of DL methods.

Although AD is extensively employed in various other domains, its application as a general-purpose tool has been relatively underutilized within the ML community. The situation is changing with the emergence of DL [3] and development of different frameworks that power rapid prototyping and code reuse [7]. Some of them are Theano [5], PyTorch [6], Torch [32], and TensorFlow [4].
As general-purpose AD gains traction in ML, two distinct categories of systems are surfacing. One set follows the define-and-run method, employing static computation graphs. In these systems, a model is created as a computational graph structure before execution. This pre-constructed graph is then used with varying inputs during runtime. TensorFlow is a prominent framework utilizing this approach. Conversely, the define-by-run technique utilizes dynamic computation graphs, a form of general-purpose AD. In this approach, a model is a standard program within the host programming language. During execution, the model generates a computational graph spontaneously as it operates. PyTorch is among the newer systems supporting this more dynamic approach [7].

PyTorch, a machine learning library, offers an imperative and Python-centric programming style, executing dynamic tensor computations swiftly through AD and GPU acceleration. It prioritizes preserving Python’s imperative programming model over potential advantages associated with graph-metaprogramming. This choice allows users to leverage a wide array of Python features. Facebook Artificial Intelligence research group has been actively developing PyTorch since 2016 [6].

PyTorch supports reverse-mode AD of scalar functions, the most important form of AD for DL applications that usually differentiate a single scalar loss. To implement AD, PyTorch overloads primitive instructions so that they record intermediate variables from an evaluation trace (operator overloading approach) [22]. Some novel design and implementation choices are:

- Dynamic, define-by-run execution – PyTorch allows its users to define the function to be differentiated by using any Python features they want (e.g., recursion, conditional statements, and loops). There’s no need to predefine a static graph structure, as required in TensorFlow, which is then differentiated beforehand and executed multiple times. Instead, this method involves differentiating at each iteration, avoiding the upfront graph creation but incurring the cost of repeated differentiation [22].
- Immediate, eager execution – Tensor computations are run when the framework encounters them. There is no materialization of the forward graph. This facilitates the concurrent processing of CPU and GPU tasks but sacrifices the chance to optimize the entire network as a whole and perform batching [22].
- In-place operations – Users frequently want to update tensors in-place instead of allocating new tensors when they know it is unnecessary. These operations pose challenges for AD as they can modify essential data needed during the differentiation phase. Furthermore, they necessitate complex transformations of the tape during the process [22]. PyTorch addresses these difficulties with the mechanisms of invalidation and aliasing. A tensor version control system tracks tensor modifications and ensures that the right data is used. Although AD manages simpler mutations automatically, exceedingly intricate cases may lead to user errors. Such errors act as signals prompting users to potentially reconsider restructuring their program, particularly concerning performance optimizations [22].
- No tape – PyTorch employs a selective approach by recording only the pertinent segments of the computation graph, allowing users to blend independent graphs as desired. Additionally, it promptly releases memory from irrelevant sections of the computation graph, which proves crucial for handling extensive graphs [22].
- Core logic in C++ – PyTorch started as a Python library, but soon it became obvious that interpreter overhead was too great for core AD logic. Most of PyTorch is implemented in C++ to improve runtime efficiency [22].

Listing 1 demonstrates how example function
\[ y = f(x_1, x_2) = \ln x_1 + x_1 x_2 - \sin x_2 \]
can be implemented and differentiated. The computation graph that is created during the execution of the program is shown in Figure 3.

```python
import torch

def f(x1, x2):
    return torch.log(x1) + x1*x2 - torch.sin(x2)

x1 = torch.tensor([2.], requires_grad=True)
x2 = torch.tensor([5.], requires_grad=True)
y = f(x1, x2)

print("y:", y)
y.backward(torch.tensor([1.])))

print("x1 gradient:", x1._grad)
print("x2 gradient:", x2._grad)

y_val: tensor([11.6521], grad_fn=<SubBackward0>)
```

Listing 1. PyTorch implementation of example function
\[ y = f(x_1, x_2) = \ln x_1 + x_1 x_2 - \sin x_2 \] [20].
TensorFlow, an open-source numerical computation library, employs data flow graphs and is optimized for extensive distributed training and inference tasks. It's versatile, suitable for research, development, and production environments, running across various systems from CPUs to GPUs and large-scale distributed setups [36].

In TensorFlow, users create a static computation graph explicitly, offering a pre-defined representation of the entire computation for repeated application to data batches. This upfront visibility theoretically aids performance and scalability. However, this approach sacrifices ease of use, debugging simplicity, and flexibility in representing various types of computations [4]. A TensorFlow program that implements example function $y = f(x_1, x_2) = \ln x_1 + x_1 x_2 - \sin x_2$ is shown in Listing 2. The computation graph and its nodes are explicitly referenced.

AD in TensorFlow is done using a source transformation approach. Similar to other tensors, gradient tensors are calculated by extending the TensorFlow graph. For instance, when determining the gradient of a tensor C concerning a tensor I, TensorFlow follows a specific sequence. Initially, it identifies the pathway in the computation graph from I to C. Then, at each step backward along this path, TensorFlow adds a node to the graph, constructing the partial gradients sequentially using the chain rule. These new nodes compute the gradient function in the forward path. This means that when a symbolic expression is differentiated, a new symbolic expression representing the gradients is produced, i.e., TensorFlow supports symbolic differentiation [4]. In contrast to PyTorch, TensorFlow is language-agnostic, utilizing an intermediate representation to store and compile computation graphs. These graphs can be generated, imported, and executed within the TensorFlow framework of various languages, extending beyond Python to include languages like C++ or JavaScript. TensorFlow employs the XLA linear algebra compiler to compile its programs [4].

TensorFlow’s core design principles are: 1. Dataflow graphs of primitive operators: TensorFlow represents fundamental mathematical operations as nodes within its dataflow graph, encompassing individual operations like matrix multiplication and updates to mutable state [4].

2. Deferred execution: TensorFlow programs typically occur in two phases. Initially, users define the program as a dataflow graph with placeholders for input data and state variables. Subsequently, an optimized program is executed across available devices.

3. Common abstraction for heterogeneous accelerators: TensorFlow introduces a universal abstraction to support a variety of specialized devices like TPUs, multicore CPUs, and GPUs, unifying their functionalities within the framework [4].

```python
import tensorflow as tf

def f(x1, x2):
    return tf.log(x1) + x1*x2 - tf.sin(x2)

g = tf.Graph()
with g.as_default():
    x1 = tf.constant(2.0)
    x2 = tf.constant(5.0)
    y = f(x1, x2)

with tf.Session() as sess:
    y_val = sess.run(y)
    print("y_val: ", y_val)

    x1_g = tf.gradients(y, x1)
    x2_g = tf.gradients(y, x2)

    x1_grad, x2_grad = sess.run((x1_g, x2_g))
    print("x1 gradient: ", x1_grad[0])
    print("x2 gradient: ", x2_grad[0])

y_val: 11.652072
x1 gradient: 5.5
x2 gradient: 1.7163378
```

Listing 2. TensorFlow implementation of example function $y = f(x_1, x_2) = \ln x_1 + x_1 x_2 - \sin x_2$ [20].
AD as a valuable tool in computational statistics and machine learning. Nonetheless, there are instances where obtaining derivatives beyond numerical values becomes essential. Symbolic differentiation methods, for instance, offer insights and aid in problem analysis. However, when dealing with complex functions involving numerous variables, the resultant expressions for derivatives quickly become intricate, diminishing the likelihood of easy interpretation.

This paper explains what AD is, describes two AD modes, forward and backward together with the two implementation approaches. For the problems in ML and DL, forward mode is more adequate because it differentiates faster functions with more inputs than outputs. Because Python plays an important role in the ML ecosystem, two of its most popular libraries for DL are described: PyTorch and TensorFlow. Both support AD but use two very different approaches with their own advantages and disadvantages.

6. ACKNOWLEDGMENT

This work was financially supported by the Ministry of Science, Technological Development and Innovation of the Republic of Serbia, contract number: 451-03-47/2023-01/200103.

REFERENCES


REZIME

DIFERENCIJABILNO PROGRAMIRANJE U MAŠINSKOM UČENJU

U ovom radu opisana je automatska diferencijacija i diskutovano je o dva primarna režima — unapred i unazad — i njihovim odgovarajućim metodama implementacije. U kontekstu problema sa kojima se susrećemo u mašinskom učenju i dubokom učenju, režim unapred se smatra prikladnijim, jer efikasno razlikuje funkcije sa numeričkim ulazima u poređenju sa izlazima. S obzirom na ključnu ulogu programskog jezika Python u oblasti mašinskog učenja, ovaj rad elaborira dve široko korišćene biblioteke dubokog učenja — PyTorch i TensorFlow. Iako obe ove biblioteke podržavaju automatsku diferencijaciju, one usvajaju različite pristupe, od kojih svaka nosi svoje prednosti i nedostatke.

Ključne reči: programske paradigme, mašinsko učenje, duboko učenje, radni okvir