Automatic Differentiation

Andrews, David Hu, Niki Ye, Xiaomeng

***Abstract*—Automatic differentiation (AD, algorithmic differentiation, computational differentiation) is a computational technique that connects mathematical concepts and application in common machine learning algorithms. Due to the fact that computer calculation is composed of basic arithmetic operations and functions, partial derivatives of a more complex function can be calculated by deriving and applying the chain rule to the constituent operations and functions [9]. In this paper, we explore the fundamentals of automatic differentiation, specifically forward mode and reverse mode accumulation, with mathematical proofs. We will showcase the utility of automatic differentiation in optimizing neural network training by implementing a Python framework designed to perform automatic differentiation and tackle simple optimization problems. Our findings not only shows that automatic differentiation is more advanced compared to traditional numerical and symbolic differentiation methods, but also highlights its potential in various fields of artificial intelligence.**

I. Introduction

It is an understatement to say that derivatives are an integral part of our civilization. They enable us optimize our engineering choices, help make financial decisions, and aid in our understanding of the world through physics. However, derivatives become prohibitively hard to compute by hand when the functions that are to be differentiated grow to unmanageable size and complexity. As a result, we have devised algorithms for computing gradients on computers.

A simple kind of differentiation algorithm is **numerical differentiation** which involves computing the limit definition of some differentiable function $f : \mathbb{R} \to \mathbb{R}$ at a point x with a fixed $h \ll 1$.

$$\frac{df}{dx} \approx \frac{f(x+h) - f(x)}{h}$$

While this approach is simple and very fast, the output is necessarily an approximation of the actual value of the derivative at point x. Even worse, errors from floating point calculations are more apparent than usual at these small scales, leading to even more inaccuracies [7].

To fix these inaccuracies, **symbolic differentiation** can be used. This method explicitly computes the algebraic expression of the derivative of a function. While this method produces precise expressions for the derivative of a function, it ultimately suffers from its large space and time complexity: it stores and manipulates whole algebraic expressions and duplicated computation when encountering the use of chain rule [13].

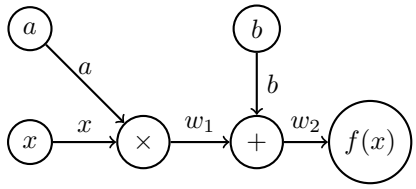
To resolve the approximations and runtime issues with both of these algorithms, **automatic differentiation** can be used. In automatic differentiation, the derivative of a function is computed at a point without having to store intermediate algebraic expressions or resorting to inaccurate approximations. This is done by first decomposing a function into a **computation graph**, a data structure which represents the data flow between the **elementary operations** such as $+$, $\times$, and **elementary functions** such as $\sin$ that compose to form f. Then, the graph is traversed one or more times to evaluate the function in terms of its elementary operations and to recursively use the chain rule to calculate the derivative. Not only is this algorithm computationally efficient and exact, it has the property that it generalizes to all computations that can be done on computers, as all programs are formed from elementary operations.

II. Forward Mode Accumulation

There are largely two variants of automatic differentiation, forward mode accumulation and reverse mode accumulation. In both cases, a function f is converted into a computation graph of elementary operations and is evaluated given some x. For example, consider

$$f(x) = ax + b$$

where f's computation graph can be visualized as:



We can represent the series of elementary operations that we perform in the forward traversal of the graph at $x = 4$ with $a = 2$ and $b = 5$ as follows:

1) $a = 2$
2) $x = 4$
3) $w_1 = a \times x$
4) $b = 5$
5) $w_2 = b + w_1$
6) $f(x) = w_2$

In the context of the computation graph, let us call the successor of a node, or elementary operation, be the consumer of the result of its computation and the predecessor of a node be the source of its inputs. In general, a node can have multiple inputs from different nodes and has one unique output which can be passed to several successor nodes as their input. [9]

To perform forward mode accumulation on the computation graph, we must start at the **leaves** of the graph, i.e. nodes without any input and thus must be inputs themselves, and traverse from one input at a time to the output. First, we set

the **seed** for each input, which are the initial values for the derivative which we will accumulate from during the traversal. In the case of the function above, our seeds for traversing from x would be:

$$\text{Seed for x: } \frac{\partial x}{\partial x} = 1$$
$$\text{Seed for a: } \frac{\partial a}{\partial x} = 0$$
$$\text{Seed for b: } \frac{\partial b}{\partial x} = 0$$

Using the constants for x, a, and b declared earlier, we will start our traversal from x. In the first step of the traversal, we calculate that

$$w_1 = 2 \times 4 = 8$$

and

$$\frac{\partial w_1}{\partial x} = \frac{\partial w_1}{\partial x} \cdot \frac{\partial x}{\partial x} + \frac{\partial w_1}{\partial a} \cdot \frac{\partial a}{\partial x} = 2 \cdot 1 + 3 \cdot 0 = 2$$

by the multidimensional chain rule. We can now continue to the next step where we calculate that

$$w_2 = 5 + 8 = 13$$

and

$$\frac{\partial w_2}{\partial x} = \frac{\partial w_2}{\partial b} \cdot \frac{\partial b}{\partial x} + \frac{\partial w_2}{\partial w_1} \cdot \frac{\partial w_1}{\partial x} = 1 \cdot 0 + 1 \cdot 2 = 2$$

We apply the multidimensional chain rule yet again and we reuse already computed values from the previous step, such as $\frac{\partial w_1}{\partial x}$. Now that we have found a value for $\frac{\partial w_2}{\partial x}$, we have effectively found $\frac{\partial f}{\partial x}$. Verifying our solution, we can algebraically differentiate $\frac{\partial f}{\partial x}$ which is equivalent to a, which is equal to 2, matching our solution.

In order to get the full gradient of each variable with respect to each input, we would need to run 2 more passes of the forward accumulation algorithm on a and b separately, making sure to set the seeds for the algorithm properly.

Notice that throughout this process, we calculated the derivative with respect to x at every step of the forward traversal process, meaning we computed values at the same time as we applied chain rule. Moreover, the derivative at each step is the derivative of the elementary operation with respect to its parameters multiplied by the parameters' derivatives with respect to the input variable. In other words, we have the following relation for intermediate result w_i:

$$\frac{\partial w_i}{\partial x} = \sum_{j \in \text{predecessors of } w_i} \frac{\partial w_i}{\partial w_j} \cdot \frac{\partial w_j}{\partial x}$$

By performing this algorithm recursively starting at the input nodes, we can effectively calculate the derivative of any intermediate value in the network with respect to an input in $O(e \cdot n)$ time, where e is the number of elementary operations in the graph and n is the input dimension of the function $f : \mathbb{R}^n \to \mathbb{R}^m$. This concludes the forward mode accumulation algorithm. [5]

III. Reverse Mode Accumulation

While forward mode accumulation is optimal for some types of functions such as $f : \mathbb{R}^m \to \mathbb{R}^n$ where $m \ll n$ since fewer passes through the graph are needed, functions such as neural networks tend to have thousands, millions, or even billions of inputs with a single output representing the loss value. In this case, reverse mode accumulation is optimal as it requires n passes as opposed to m passes to compute the full gradient of the function.

Reverse mode accumulation is a very similar process to forward mode accumulation. In reverse mode, we first do a forward traversal of the graph solely to calculate all the intermediate values and then a backward traversal to calculate the derivatives of the intermediate values with respect to a single output. In forward mode, we calculate the derivatives and intermediate values in one forward pass and as a result, the derivatives are of the form $\frac{\partial w_i}{\partial x}$ as they measure x's influence on the value of each intermediate value; in reverse mode, we calculate derivatives of the form $\frac{\partial y}{\partial w_i}$ which measures the intermediate value's influence on the output's value. At the end of either traversal, we end up with $\frac{\partial y}{\partial x}$. [9]

In this way, we can formulate a relation for the value of the derivative at each intermediate value in the graph during reverse mode accumulation:

$$\frac{\partial y}{\partial w_i} = \sum_{j \in \text{successors of } w_i} \frac{\partial w_j}{\partial w_i} \cdot \frac{\partial y}{\partial w_j}$$

At the beginning of our traversal, we would, similar to forward mode accumulation, set one output seed to 1 and the rest to 0 to ensure proper computation of the output variable's gradient with respect to each intermediate values and the inputs. As such, reverse mode accumulation requires n passes through e nodes, making its time complexity $O(e \cdot n)$ for a function $f : \mathbb{R}^m \to \mathbb{R}^n$. [5]

IV. Algorithm Implementation

Since forward and reverse mode are algorithms which are implemented on computers, we can formulate the following pseudo-code for both of them. Importantly, we used **topological sort** in these algorithms, which means that we sort the computation graph in order of dependencies so that when we traverse the graph in the forward or reverse direction, each node's dependencies are fulfilled before calculating its own derivative or value. Since computational graphs are directed and acyclic, topological sort is very suitable in this case.

Algorithm 1 Forward Mode Accumulation

Require: Set all $\frac{\partial w_i}{\partial x}$ to 0 and $\frac{\partial x}{\partial x}$ to 1.
  procedure FORWARD(graph)
    for each node in TOPOLOGICALSORT(graph) **do**
      node.value ← evaluate(node)
      for each pred in node.predecessors **do**
        deriv ← computeDerivative(node, pred)
        node.deriv ← node.deriv + deriv × pred.deriv
      end for
    end for
  end procedure

Algorithm 2 Reverse Mode Accumulation

Ensure: All derivatives in the graph are set to 0 except for the output being differentiated with respect to, which is set to 1
Require: Graph to be pre-computed before reverse is called
  procedure REVERSE(graph)
    for each node in REVERSEDTOPOLOGICALSORT(graph) **do**
      for each successor in node.successors **do**
        deriv ← computeDerivative(successor, node)
        node.deriv ← node.deriv + deriv × successor.deriv
      end for
    end for
  end procedure

V. Applications to Variational Inference

Variational Inference (VI) is a fast converging statistical model that involves the usage of optimization techniques to approximate complex probability densities. Compared to classical models for similar estimations under Bayesian Methods like various **Markov Chain Monte Carlo (MCMC) sampling**, VI uses a metric called **Kullback-Leibler(KL) divergence** and apply optimization to it in order to reach the approximation significantly faster and provides a more scalable approach. Introducing **Automatic Differentiation Variantial Inference (ADVI)** to VI algorithms allows gradient-based computations, or more specifically **stochastic gradient ascent**, to be automatically calculated, therefore speeding up the estimation process and scaling the Bayesian models to large dataset and complex models. **Variational Auto-Encoder (VAE)** is another example of a stochastic variational inference algorithm based statistical model. It uses **stochastic gradient backpropagation** to approximate the posterior distribution for the latent vector compressing high dimensional data. [3] [5] [12]

VI. Newton-Raphson Method with Automatic Differentiation

Newton-Raphson Method is a numeric technique used to find roots of differentiable functions. Using the following iterative formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

where x_n is the root approximated after n steps, with x_0 being the initial guess and f being the differentiable function, this method will successively get to the true root under stable conditions. Automatic Differentiation can be applied to overcome the difficulty of having to know the derivative of the function. Automatic Differentiation benefits the calculation of high dimensional derivatives being stored in full $n \times n$ **Hessian matrix**. We can continue applying AD and sorting the operations in a new computational graph repeatedly to expand Newton's Method to differentiable $f : \mathbb{R}^n \to \mathbb{R}^m$ [5] [14].

VII. Training Deep Neural Networks

Neural networks are large functions whose parameters are optimized through **backpropogation**, a special case of reverse mode accumulation, and **stochastic gradient descent**, an algorithm that iteratively adjusts the parameters of a function using the gradient of the function with respect to a loss function. We can represent models in terms of their function and parameters, (f, θ), where f represents the function and θ represents the parameters of the model. At each step of training, we evaluate loss $= L(f, \theta, X, Y)$, where L is the loss function which outputs a scalar value measuring how much $f(X)$ deviates from $f(Y)$ with (X, Y) representing the set of training dataset input-output pairs. An example loss function is mean squared error (MSE):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i))^2$$

Since we have a single output, a single scalar value, we are able to efficiently employ reverse mode accumulation on the whole function, computing the gradients of θ with respect to the loss value. Once we have done the reverse accumulation process, we can update the model's parameters through the following recurrence relation:

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_{\theta t} y$$

where $\nabla_{\theta t} y$ is the gradient of the parameters with respect to y and α is a small constant representing a learning rate, i.e. how large the step size should be when updating the parameters. Since the gradient represents the direction of largest increase of the output, negating the gradient gives the direction of largest decrease of the function.

VIII. Neural Network Backpropogation Python Framework

We have implemented a custom framework for performing reverse mode accumulation, more specifically backpropogation, on small neural networks.

The Python code for our framework is available on Github [8] [11].

The following is our implementation of Node class (some functions serving similar purposes have been eliminated

for simplicity reasons). The Node class is central to the framework, representing a node in the computation graph. When applying an operation on one or more nodes, a new node is created in the computation graph which keeps track of its parents and the operation it was created with. **Activation Functions** such as `relu` and `sigmoid` are mathematical equations that determine the output of a neuron given one or more inputs from the previous layer. Their primary role is to introduce non-linear properties to the neural network, enabling it to learn more complex representations instead of a simple linear function. The implementation of topological sort uses a post-order Depth-First Search to construct the list for back propagation.

```
class Node:
    def __init__(self, value, op=None, parents=[]):
        self.value = value
        self.op = op
        self.grad = 0
        self.parents = parents

    def __add__(self, other):
        return Add.apply(self, other)

    def __neg__(self):
        return Neg.apply(Neg, self)

    def __mul__(self, other):
        return Mul.apply(self, other)

    def __pow__(self, other):
        return Pow.apply(self, other)

    def __truediv__(self, other):
        return Div.apply(self, other)

    def sigmoid(self):
        return Sigmoid.apply(Sigmoid, self)

    def relu(self):
        return ReLU.apply(ReLU, self)

    def topological_sort(self):
        list = []
        visited = set()

        def build(node):
            if node not in visited:
                visited.add(node)
                for parent in node.parents:
                    build(parent)
                list.append(node)

        build(self)
        return list

    def backward(self, seed=1):
        sorted_nodes = self.topological_sort()
        self.grad = seed

        for node in reversed(sorted_nodes):
            if node.op:
                node.op.backward()
```

Each operation in the framework has a `forward` and `backward` function which describe the operation fully. For example, the `Add` operation distributes the gradient equally to the node's parents. Each operation applies chain rule by multiplying the passed gradient by the child's gradient.

```
class Add(BinaryOp):
    def forward(self):
        self.c = Node(self.a.value + self.b.value,
        self, parents=[self.a, self.b])
        return self.c

    def backward(self):
        self.a.grad += 1 * self.c.grad
        self.b.grad += 1 * self.c.grad

class Mul(BinaryOp):
    def forward(self):
        self.c = Node(
            self.a.value * self.b.value,
            self,
            parents=[self.a, self.b]
        )
        return self.c

    def backward(self):
        self.a.grad += self.b.value * self.c.grad
        self.b.grad += self.a.value * self.c.grad

class Pow(BinaryOp):
    def forward(self):
        self.c = Node(
            self.a.value**self.b.value,
            self,
            parents=[self.a, self.b]
        )
        return self.c

    def backward(self):
        self.a.grad +=
            (self.b.value
            * self.a.value ** (self.b.value - 1)
            * self.c.grad)
        self.b.grad += self.c.value
            * math.log(self.a.value)
            * self.c.grad

class Sigmoid(UnaryOp):
    def forward(self):
        self.b = Node(
            1 / (1 + math.exp(-self.a.value)),
            self,
            parents=[self.a]
        )
        return self.b

    def backward(self):
        self.a.grad += self.b.value *
                    (1 - self.b.value) *
                    self.b.grad
```

We can then create neurons from weights, a bias, and an activation function. Next, we can construct an MLP layer from multiple neurons. A neuron will represent a node in the computational graph. Layers will contain multiple neurons, and each neuron will have an associated weight and bias term.

```
class Neuron:
    def __init__(self, input_dim, activation):
        self.input_dim = input_dim
        self.weights = [
            Node(random.gauss())
            for _ in range(input_dim)
        ]
        self.bias = Node(random.random())
        self.activation = activation

    def __call__(self, x):
```

```
        value = sum(
            self.weights[i] * x[i]
            for i in range(self.input_dim)
        ) + self.bias
        if self.activation == "relu":
            return value.relu()
        elif self.activation == "sigmoid":
            return value.sigmoid()
        elif self.activation == "linear":
            return value

class Layer:
    def __init__(self,
                 input_dim,
                 output_dim,
                 activation):
        self.neurons = [
            Neuron(
                input_dim=input_dim,
                activation=activation
            )
            for _ in range(output_dim)
        ]

    def __call__(self, x):
        output = []
        for neuron in self.neurons:
            output.append(neuron(x))
        return output
```

Using the code below, we are able to generate a graph and calculate the derivative of each node in the compute graph with respect to the output, `a`.

```
w = Node(2)
x = Node(3)
y = Node(4)
z = Node(2)

a = (x + y) ** z - w
a.backward()
```

The graph below serves as an example of the operation performed on a simple function. Each node represent either operations (such as "Add", "Pow") or operands. Each node has a "value" which is the result of the computation or the input value, and a "grad" which represents the gradient of the loss function with respect to that node's value after backpropagation. Each edge in the graph indicate the flow of the computation, connecting the inputs of an operation to its output.

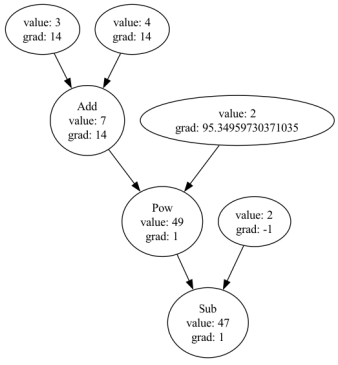


Fig. 1. Visualization of Simple Compute Graph after Reverse Accumulation

The following graph is a visualization of performing backpropagation using the Python Framework we wrote in a large scale. This graph depicts a neuron with two inputs and a MSE loss function applied to its output. After each step of training, the gradient of each node would be reset to 0 to prevent interference, as this graph shows.

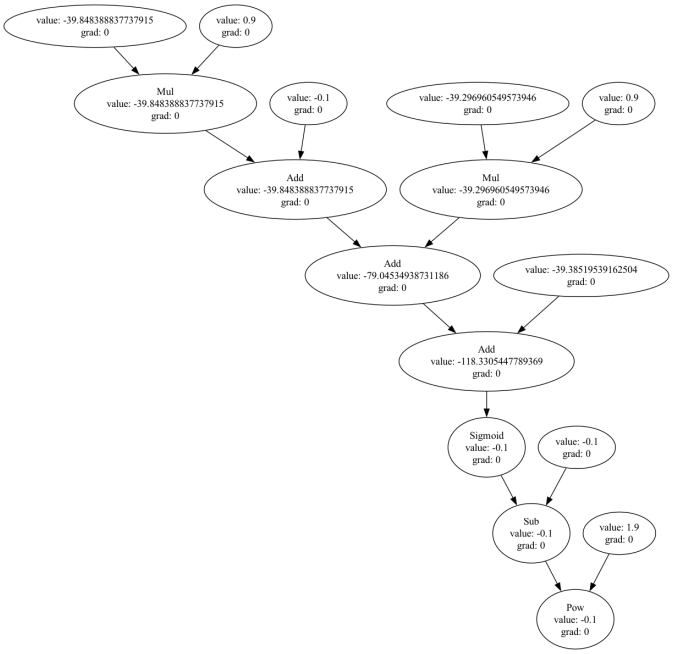


Fig. 2. Visualization of Computation Graph After Backpropagation

IX. MNIST Classification Application

With this framework, one is able to train simple neural networks like multi-layer perceptrons on tasks such as MNIST for classifying 28×28 grayscale handwritten digits. The model can be trained using the method of reverse mode accumulation, also called backpropogation in the context of neural networks.[6] [4]

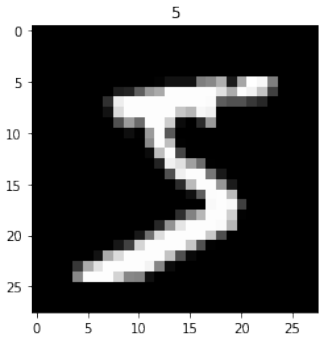


Fig. 3. MNIST digit of a 5

Each neuron in the model can be represented as the following:

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i\right)$$

where σ is the neuron's activation function.

At each layer boundary, every neuron is connected to every other neuron in the previous layer. As such, we can represent a layer of neurons as a linear transformation from $\mathbb{R}^m \to \mathbb{R}^n$, or simply as a $n \times m$ matrix. Then, our network can simply be represented as:

$$y = \sigma\left(W_2 \cdot \sigma\left(W_1 x\right)\right)$$

where $W_1 \in \mathbb{R}^{h \times 784}$, $W_2 \in \mathbb{R}^{10 \times h}$, and σ is the sigmoid function.

When building a model to perform this classification task, we can turn the 28×28 image into a vector of size 784 by flattening the image. The second layer, also called the **hidden layer** contains some number of neurons which then connect to the 10 output neurons, representing each digit from 0 to 9. The output of this last layer is then transformed into a probability distribution by using the **softmax** function and then a **categorical cross-entropy loss function** is used to measure the difference between the predicted and target distribution for the particular training example.

The network can be trained using SGD to match **one-hot** distributions where the correct digit will have 100% probability with all the other digits having 0% probability. The goal of this classification network, in other words, is to match this ideal distribution for each data point.

X. Conclusion

A. Summary of Main Result

In this paper, we gave an overview of differentiation algorithms, explored automatic differentiation with forward and reverse mode accumulation in depth, discussed applications of automatic differentiation, and built and tested a general framework for reverse mode accumulation.

B. History

Forward mode accumulation automatic differentiation was introduced by Robert Edwin Wengert in 1964 through his research paper *A Simple Automatic Derivative Evaluation Program*. It was believed that "The technique permits computation of numerical values of derivatives without developing analytical expressions for the derivatives." [1] Reverse accumulation as well as backpropagation was being discussed around the same time and was formally integrated and published by Seppo Linnainmaa in 1970s. However, reverse accumulation and backpropagation was not widely used before the significant increase in the computational speed. In order to make training large networks easier, those like Schmidhuber had been looking for approaches to pretrain the model layer by layer. It turns out that the first few layers of a very big neural network model would not have much impact on the results, so pretraining them would help reduce computational time by a significant amount. With the invention of faster computers and better algorithms, neural networks gained popularity, alongside with backpropagation. [2]

C. Limitations

1) **Functions cannot be black boxes**: To perform forward and reverse mode accumulation on a function, the precise definition of the function must be known to correctly calculate the gradients. This poses an issue in situations where some aspect of the problem may not be clearly known, such as in complex environments where genetic algorithm or reinforcement learning algorithms perform well without relying on a loss function.
2) **Decrease in efficiency with larger scale**: Forward mode accumulation's efficiency decreases as the number of inputs increases because it requires a separate differentiation pass for each input variable. Similarly, reverse mode accumulation's efficiency decreases as the number of output increases.
3) **Simulating individual neurons is slow**: While our framework is useful for small experiments and educational purposes, a more widely used deep learning framework such as PyTorch uses tensors as the fundamental building block of the library, enabling fast acceleration with optimized linear algebra routines and multiple processors on CPUs and GPUs.

XI. Future Work

Work is being conducted to further understand how gradient descent works well for training large neural networks. There is some current work suggesting that gradient descent and backpropagation is a method of data compression. In effect, training a neural network is analogous to creating a minimal program which can produce the whole dataset. Further work can explore this in more depth, exploring how different neural network architectures effect how gradient descent and backpropogation promote information compression and encoding, leading to more mechanistic understanding of differences in performance between various neural network architectures. [10]

References

[1] R. E. Wengert. "A simple automatic derivative evaluation program". **in***Commun. ACM*: 7.8 (**august** 1964), **pages** 463–464. ISSN: 0001-0782. DOI: 10.1145/355586.364791. URL: https://doi.org/10.1145/355586.364791.

[2] Tim Dettmers. *Deep Learning in a Nutshell: History and Training*. NVIDIA Technical Blog, **december** 2015. URL: https://developer.nvidia.com/blog/deep-learning-nutshell-history-training/#:~:text=Backpropagation%20was%20derived%20already%20in (**urlseen** 08/02/2024).

[3] Alp Kucukelbir **andothers**. "Automatic Differentiation Variational Inference". **in***arXiv (Cornell University)*: (**march** 2016). DOI: 10.48550/arxiv.1603.00788. (**urlseen** 20/01/2024).

[4] Grant Sanderson. *Backpropagation calculus*. **byeditor**Josh Pullen. www.3blue1brown.com, **november** 2017. URL: https://www.3blue1brown.com/lessons/backpropagation-calculus (**urlseen** 08/02/2024).

[5] Charles C. Margossian. "A review of automatic differentiation and its efficient implementation". **in***WIREs Data Mining and Knowledge Discovery*: 9 (**march** 2019). DOI: 10.1002/widm.1305. (**urlseen** 13/12/2020).

[6] Kiran Achyutuni. *The Role of Automatic Differentiation in Machine Learning*. Medium, **may** 2020. URL: https://medium.com/deep-dives-into-computer-science/like-backpropagation-is-there-forward-propagation-as-well-fedb22828b36 (**urlseen** 08/02/2024).

[7] *Numerical differentiation*. Wikipedia, **july** 2020. URL: https://en.wikipedia.org/wiki/Numerical_differentiation.

[8] Andrej. *micrograd*. GitHub, **november** 2022. URL: https://github.com/karpathy/micrograd.

[9] *Automatic differentiation*. Wikipedia, **february** 2022. URL: https://en.wikipedia.org/wiki/Automatic_differentiation#:~:text=In%20mathematics%20and%20computer%20algebra (**urlseen** 08/02/2024).

[10] AWS. *What is a Neural Network? AI and ML Guide - AWS*. Amazon Web Services, Inc., 2023. URL: https://aws.amazon.com/what-is/neural-network/#:~:text=A%20neural%20network%20is%20a (**urlseen** 08/02/2024).

[11] George Hotz. *tinygrad/tinygrad*. GitHub, **april** 2024. URL: https://github.com/tinygrad/tinygrad (**urlseen** 22/04/2024).

[12] Ankush Ganguly **and** Samuel Earp. *An Introduction to Variational Inference*. URL: https://arxiv.org/pdf/2108.13083.pdf (**urlseen** 15/03/2024).

[13] Noel Welsh. *Symbolic Differentiation*. www.creativescala.org. URL: https://www.creativescala.org/case-study-gradient-descent/symbolic-differentiation.html (**urlseen** 15/03/2024).

[14] Wanchaloem Wunkaew, Yuqing Liu **and** Kirill Golubnichiy. *Using The Newton-Raphson method with Automatic Differentiation to numerically solve Implied Volatility of stock option through Binomial Model*. URL: https://arxiv.org/pdf/2207.09033.pdf (**urlseen** 16/03/2024).