Nano-AutoGrad: A Micro-Framework Engine Based on Automatic Differentiation for Building and Training Neural Networks

Unraveling the Mathematical Foundations: Building Tiny-PyTorch's AutoGrad Engine and the Essence of Computational Graphs

Haytham Al Ewaidat [1,*] Youness El Brag [2]
Ahmad Wajeeh Yousef E'layan [3] Ali Almakhadmeh [4]

[1]Jordan University of Science and Technology, Faculty of Applied Medical Sciences, Department of Allied Medical Sciences-Radiologic Technology, Irbid, Jordan, 22110
[2]Abdelmalek Essaâdi University of Science and Technology, Faculty of Multi-Disciplinary Larache, Department of Computer Sciences, Ksar El Kebir, Morocco, 92150
[3,4]Jordan University of Science and Technology, Faculty of Applied Medical Sciences, Department of Allied Medical Sciences-Radiologic Technology, Irbid, Jordan, 22110

Correspondence author: Dr. Haytham Al Ewaidat
Department of Allied Medical Sciences-Radiologic Technology
Faculty of Applied Medical Sciences, Jordan University of Science and Technology
PO Box 3030, Irbid 22110, Jordan
Tel: (+962)27201000-26939
Fax: (+962)27201087
E-mail: haewaidat@just.edu.jo

Conflict of interest: The authors declare no conflict of interest of any type.

Data availability: The data and code for this article are available on GitHub: The code implementation can be found in our GitHub repository: https://github.com/deep-matter/Nano-AutoGrad.

Funding: This work is supported by the Jordan University of Science and Technology, Irbid-Jordan, under grant number 20200649.

Abstract

Background

Neural Networks, inspired by the human brain, are a class of machine learning models composed of interconnected artificial neurons. They have a rich history dating back to the 1940s, with notable advancements in the 1980s and 1990s when techniques like back-propagation enabled the training of multi-layer networks. Neural Networks have since experienced a renaissance, achieving state-of-the-art results in diverse domains. However, training them effectively remains a challenge. This thesis introduces Nano-AutoGrad, a system built upon automatic differentiation and optimization methods. Nano-AutoGrad efficiently computes gradients, facilitates parameter optimization, and incorporates mechanisms such as Multi-Perceptrons and Linear models. It also allows for expanding the network architecture with additional layers, enhancing the performance and representation capabilities of Neural Networks. The objective is to design and develop Nano-AutoGrad as an advanced tool for training complex models, leveraging historical advancements and computational graph understanding.

Method and Aim

Nano-AutoGrad is a Micro-Framework that efficiently trains Neural Networks using automatic differentiation and optimization techniques and Computational graphs. It computes gradients by applying the chain rule, enabling parameter updates for improved performance. Optimization methods like Stochastic Gradient Descent (SGD), are utilized to iteratively update network parameters based on computed gradients. The aim of this thesis is to design and develop Nano-AutoGrad as a simple easy compute Micro-Framework for training Neural Networks. By incorporating mechanisms such as Multi-Perceptrons, Linear models, and additional layers, the goal is to enhance network performance and versatility. The study of Nano-AutoGrad in training Linear models and achieving promising results. Nano-AutoGrad's efficient gradient computation and parameter optimization contribute to advancements in Neural Network training

Conclusion

Through the development and evaluation of Nano-AutoGrad, this thesis highlights the effectiveness of historical advancements in Neural Networks, combined with automatic differentiation and optimization techniques. By incorporating mechanisms such as Multi-Perceptrons and Linear models, and expanding the network architecture with additional layers, Nano-AutoGrad demonstrates simple computational yet improved representation capabilities. The study showcases the potential of Nano-AutoGrad to contribute to the field of machine learning by providing a high-abstract tool for training Linear models and optimizing Neural Networks, building upon the rich history and progress in this field. The understanding of computational graphs further enhances the comprehension of how Neural

Networks process information and compute gradients, making them a key aspect of the training process.

Keywords:

Nano-AutoGrad, Neural networks, Automatic differentiation, Computational graphs, Optimization techniques, Linear Model,

Acknowledgement

I would like to express my sincere gratitude toHaytham Al Ewaidat, my internal Co-supervisor, at Jordan University of Science and Technology, for their unwavering support, guidance, and valuable insights throughout the course of this thesis. Without their mentorship and encouragement, this research would not have been possible.

Lastly, I would like to express my deep gratitude to my parents for their unwavering support, encouragement, and belief in me throughout my academic and personal endeavors.

Contents

List of Abbreviations **ii**

List of Figures **iii**

List of Tables **iii**

1 Chapter Introduction **1**
- 1.1 Motivation 1
- 1.2 Aim 3
- 1.3 Objectives 3
  - 1.3.1 Overview of The heart of AI: Backpropagation 3
- 1.4 Study questions 4
- 1.5 Delimitations 4

2 Chapter History of Neural Networks **5**
- 2.1 Overview Neural Network history 5
- 2.2 Neural Networks as Bio-inspired Algorithms 5

3 Chapter Theory **8**
- 3.1 Neural Networks 8
  - 3.1.1 Artificial Neuron 8
  - 3.1.2 Activation and output rules 9
- 3.2 Multi-Layer Perceptron 10

3.3 Tensor Foundation . . . 11
3.3.1 Einstein notation . . . 12
3.4 Tensor calculus . . . 14
3.4.1 Forward Mode . . . 14
3.4.2 Reverse Mode . . . 17
3.4.3 Beyond Forward and Reverse Mode . . . 20
3.5 Optimizations and computational Graph . . . 22
3.5.1 Universal approximation theorem . . . 22
3.5.2 Estimation of the parameters . . . 23
2.3.2.1 Loss Function . . . 23
2.3.2.2 Stochastic Gradient Descent Algorithm . . . 24
3.5.3 Automatic Differentiation . . . 25
2.3.3.1 Computational Graphs . . . 26
2.3.3.2 Computational Graphs and the Chain Rule of Differentiation 26
2.3.3.4 Back-Propagation . . . 28

4 Chapter Methods and Micro-Framework Modeling 30
4.1 Micro-Framework Modeling . . . 30
4.1.1 Micro-Framework documentation . . . 31
4.1.2 Building Neural network using Nano-AutoGrad . . . 35
4.2 Data collection . . . 38

5 Chapter Experiment 40

6 Chapter Discussion 43
6.1 Capabilities and Applications of Nano-AutoGrad . . . 43
6.2 Limitations and Challenges . . . 43
6.3 Future Directions . . . 43

7 Chapter Conclusion 44
7.1 Summary of Findings . . . 44
7.2 Contributions . . . 44
7.3 Final Remarks . . . 44

Bibliography 46

List of Abbreviations

NN - Neural Network
AD - Automatic differentiation
MLP - Multilayer perceptron
SGD - Stochastic gradient descent
DAG - Directed acyclic graph

List of Figures

2.1 Neurons in the cerebral cortex, a part of the mammalian brain 6
2.2 Schematic image of a neuron. 6
2.3 Spike train in the electro-sensory pyramidal neuron of a fish 7

3.1 Schematic representation of an artificial neuron. 9
3.2 Various activation functions for a unit 10
3.3 basic neural network multi-layer perceptron 11
3.4 Expression DAG for Expression (1) 15
3.5 Modeling Complex Functions Neural network 23
3.6 Visualization of the SGD optimization landscape. The plot represents the surface $z = x^2 - y^2$ in three dimensions. The goal of the SGD algorithm is to find the minimum of this surface. 25
3.7 Computational graph for the sum of variables x and y. 26
3.8 Computational graph representing the expression $z = \log(3x^2 + 5xy)$. . . . 27

4.1 Micro-Framework Modeling Nano-AutoGrad (a): Linear Module includes all necessary components to initialize weight Matrix and mathematical logic, (b): AutoGrad is Engine responsible for computing function derivative Order built on top of a dynamically built Directed Acyclic Graph (DAG) (c): Optimizer SGD to update Weight model Through Back-Propagation (d): Activation Func collection of the non-linear function used for connectivity between Layer 31
4.2 Graph Update weights(DAG) Update weights 38
4.3 Samples Data Minist Digit 39
4.4 Generating Data from make_moons Distribution Function 39

5.1 our web application interface 40
5.2 Hyper-Parameters Tuning Setting and Specify Task 40
5.3 Training Step Plot of the Loss Function and Accuracy 41
5.4 Output Prediction of Digits 41
5.5 Output of Approximation of Sparsity Between 2D Data Point Samples . . . 42

List of Tables

3.1 Comparison of different linear algebra notations. 13

1 Chapter Introduction

1.1 Motivation

The field of neural networks, also known as connectionist models or parallel distributed processing, experienced a surge of interest with the introduction of simplified neurons by McCulloch and Pitts in 1943 [1]. These neurons were initially proposed as models of biological neurons and as conceptual components for computational circuits. However, the enthusiasm for neural networks waned when Minsky and Papert published their book "Perceptrons" in 1969 [2], which highlighted the limitations of perceptron models. As a result, funding for neural networks was redirected, and many researchers left the field. Only a handful of dedicated researchers, such as Teuvo Kohonen, Stephen Grossberg, James Anderson, and Kunihiko Fukushima, continued their efforts.

The renewed interest in neural networks emerged in the early 1980s when significant theoretical advancements were made, including the discovery of error backpropagation and advancements in hardware capabilities. This revival is evident in the increasing number of scientists, funding allocations, conferences, and journals dedicated to neural networks. Nowadays, most universities have dedicated neural networks [3] research groups within their psychology, physics, computer science, or biology departments. Artificial neural networks [4] can be characterized as computational models with unique properties such as adaptability, learning ability, generalization, and the ability to cluster or organize data through parallel processing. However, it is crucial to determine the extent to which neural networks outperform existing non-neural models in specific applications. This question remains unanswered and subject to ongoing research. Biological systems[5] often serve as a source of inspiration for neural network models. However, our understanding of biological systems, even at the lowest cell level, is limited. As a result, the models we employ for artificial neural systems may oversimplify the complexity of biological systems.

In this study, we provide an introduction to artificial neural networks, exploring their theoretical foundations [6], practical applications, and the challenges associated with deep learning. Deep learning models, a subclass of neural networks, excel at discovering latent hierarchical structures within large datasets. However, their complexity, characterized by numerous layered models and billions of parameters, makes them inherently difficult to comprehend, even for experts in the field.The term "deep learning" encompasses three interconnected meanings: knowledgeable, reflecting the model's accuracy in specific image processing tasks; layered, visualizing the learned hierarchical structures; and impenetrable, representing the inherent lack of interpretability and understanding of the algorithmic operations. In this course, we delve into these three dimensions and explore their intricate relationship with each other. Moreover, the advent of modern artificial intelligence (AI) systems, equipped with billions of elementary components and empowered by deep learning, has achieved remarkable milestones in tackling tasks that were once considered exclusive to natural intelligence. Deep learning utilizes artificial neural networks as

a model, which, while loosely inspired by biological neural networks, represents a flexible set of functions built from basic computational blocks called neurons. This model of computation differs significantly from the traditional programming paradigm used in conventional computers.

Deep neural networks, comprising multiple layers of parallel neurons organized sequentially, hold immense power in learning representations of the world. This process of representation learning transforms data into increasingly refined forms that facilitate solving complex tasks. Such capability is considered a hallmark of success in both artificial and biological intelligence. Despite the accomplishments and widespread interest in deep learning, the theoretical understanding of this framework is still in its infancy. Theoretical analyses often involve unrealistic assumptions that fail to capture the essence of deep neural networks as they are commonly used in practice. There exists a considerable gap between theory and practice, with practitioners achieving groundbreaking results that outpace theoretical advancements. Consequently, the deepness of deep learning remains a subject of limited theoretical exploration, despite the wealth of empirical evidence highlighting its importance in the framework's success.

When it comes to gradient-based optimization in machine learning, the calculation of derivatives plays a crucial role. Several methods exist for computing derivatives, including Manual Differentiation, Numerical Differentiation, Symbolic Differentiation, and Automatic Differentiation [7] [3]. Manual Differentiation involves applying fundamental derivative rules to compute derivatives, but it can be time-consuming, especially for complex functions. Numerical Differentiation, on the other hand, relies on finite differences and is relatively easy to implement. However, it may suffer from accuracy issues due to round-off and truncation errors. Moreover, Numerical Differentiation does not scale well for gradients involving millions of parameters, making it unsuitable for machine learning tasks.

Symbolic Differentiation [8] provides an automated version of Manual Differentiation and overcomes some of its limitations. However, it often leads to expression swelling, where the resulting expressions become unwieldy and difficult to handle effectively. In contrast, Automatic Differentiation leverages the chain rule of differentiation. It introduces a change in the domain of variables to include derivative values and modifies the semantics of operators to propagate derivatives according to the chain rule. Automatic Differentiation dynamically generates numerical derivative evaluations during code execution, rather than producing derivative expressions at the time of implementation. This approach enables efficient computation of derivatives and is particularly well-suited for machine learning, where gradients with respect to numerous parameters need to be calculated.

While deep learning has flourished with the utilization of gradient-based optimization and automatic differentiation, theoretical understanding of the framework still lags behind practical advancements [4]. Theoretical analyses often rely on unrealistic assumptions that fail to capture the complexities and nuances of deep neural networks in real-world applications. Closing the gap between theory and practice is a critical area of research. By bridging this divide, we can uncover deeper insights into the workings of deep learning models, enhance interpretability, and unlock further potential for advancement.

In this section, we will explore the motivations behind neural networks and deep learning, investigate their theoretical foundations, and delve into practical applications. By gaining a comprehensive understanding of these topics, we aim to equip you with the knowledge and skills necessary to harness the power of neural networks and navigate the intricacies of deep learning.

1.2 Aim

The aim of Nano-Autograd is to provide a simple and efficient Micro-Framework for training Neural Networks by enabling effortless computation of gradients using automatic differentiation and optimization techniques. It empowers users to leverage the full expressive power of a modern high-level programming language (Python) and a mature numerical library (Numpy) when defining loss functions. Nano-Autograd supports higher-order derivatives, allowing the computation of gradients for functions composed of gradients. By incorporating mechanisms such as Multi-Perceptrons, Linear models, and additional layers, the goal is to enhance network performance and versatility. The efficient gradient computation and parameter optimization offered by Nano-Autograd contribute to advancements in the training of Neural Networks.

1.3 Objectives

The objectives of this study are as follows:

1. Develop Nano-Autograd, a simple and efficient Micro-Framework for training Neural Networks.
2. Implement automatic differentiation and optimization techniques in Nano-Autograd to enable effortless computation of gradients and efficient parameter updates.
3. Support higher-order derivatives in Nano-Autograd, allowing the computation of gradients for functions composed of gradients.
4. Incorporate mechanisms such as Multi-Perceptrons, Linear models, and additional layers into Nano-Autograd to enhance network performance and versatility.
5. Provide comprehensive documentation and examples to facilitate the usage and understanding of Nano-Autograd by researchers and practitioners.

By achieving these objectives, this study aims to contribute to the field of Neural Networks by providing a user-friendly and efficient tool for training models, enabling researchers and practitioners to explore and innovate in the realm of deep learning.

1.3.1 Overview of The heart of AI: Backpropagation

It is important to note that Nano-Autograd builds upon the fundamental concept of backpropagation, which forms the core of many modern deep learning frameworks, including PyTorch. Backpropagation [9] is a powerful algorithm that allows for efficient computation of gradients in Neural Networks, enabling effective parameter updates through optimization techniques.

PyTorch, being a widely used and mature deep learning framework, provides extensive functionality and support for training large-scale and complex Neural Networks. It offers a range of advanced features, optimization algorithms, and extensive documentation, making it a preferred choice for researchers and practitioners working with more sophisticated model architectures. While Nano-Autograd aims to provide a simpler alternative for training Neural Networks, it acknowledges the importance of frameworks like PyTorch for tackling larger-scale problems and leveraging cutting-edge deep learning advancements. Researchers and practitioners should consider the specific requirements of their tasks and the scale of their models when choosing between Nano-Autograd and frameworks like [10].

1.4 Study questions

This study aims to address the following Study questions:

1. How can Nano-Autograd be developed as a simple and efficient Micro-Framework for training Neural Networks?
2. How can automatic differentiation and optimization techniques be implemented in Nano-Autograd to enable effortless computation of gradients and efficient parameter updates?
3. To what extent can Nano-Autograd support higher-order derivatives, allowing the computation of gradients for functions composed of gradients?
4. How does the incorporation of mechanisms such as Multi-Perceptrons, Linear models, and additional layers in Nano-Autograd enhance network performance and versatility?

By addressing these research questions, this study aims to provide insights into the development and capabilities of Nano-Autograd, contributing to the broader understanding and application of automatic differentiation and optimization techniques in the field of Neural Networks.

1.5 Delimitations

This study has the following delimitations:

Focus on Nano-Autograd: The study and development efforts in this study are centered around Nano-Autograd as a specific Micro-Framework for training Neural Networks. While Nano-Autograd aims to provide simplicity and efficiency, its scope is limited to simple model architectures and may not be suitable for more complex and large-scale networks.

Simplicity vs. Scalability: Nano-Autograd is designed to be a simple and easy-to-use Micro-Framework, prioritizing ease of implementation and understanding. However, its focus on simplicity may result in limitations when it comes to handling large-scale models and complex computational graphs. For more scalable and advanced Neural Network training, alternative frameworks such as PyTorch, which offer extensive capabilities and optimizations, should be considered.

2 Chapter History of Neural Networks

2.1 Overview Neural Network history

The idea of an artificial Neural Network goes back to 1940 when Walter Pitts and Warren McCulloch discovered that neurons in the human brain essentially perform combinations of logical operations and have binary outputs [11] that depend on a specific threshold: active or not active. Based on that fact, mathematical models were built and created great interest in the Artificial Intelligence community. However, at that time, computers had just been invented and were not close to being able to handle algorithms of such complexity. In the following years, scientists lost interest in Neural Networks due to a lack of progress in the field and other, at the time, more promising methods. In 1970 Seppo Linnainmaa discovered Backpropagation, that later should revolutionize the performance of Neural Networks. Still, Neural Networks were not the focus of the Artificial Intelligence Community until 2015, when students won the ImageNet Large Scale Visual Recognition Competition with remarkable results. Since then, Neural Networks are an uprising topic in Artificial Intelligence.

2.2 Neural Networks as Bio-inspired Algorithms

Different regions in the mammalian brain perform different tasks. The cerebral cortex is the outer part of the mammalian brain, one of its largest and most developed segments. We can think of the cerebral cortex as a thin sheet (about 2 to 5 mm thick) that folds upon itself to form a layered structure with a large surface area that can accommodate large numbers of nerve cells.

neurons. The human cerebral cortex contains about 1010 neurons. They are linked together by nerve strands (axons) that branch and end in synapses. These synapses are the connections to other neurons. The synapses connect to dendrites, branches extending from the neural cell body that is designed to collect input from other neurons in the form of electrical signals. A neuron in the human brain may have thousands of synaptic connections with other neurons.

The resulting network of connected neurons in the cerebral cortex is responsible for the processing of visual, audio, and sensory data. 2.1 shows neurons in the cerebral cortex. This drawing was made by Santiago Ramón y Cajal more than 100 years ago. By microscope, he studied the structure of neural networks in the brain and documented his observations by ink-on-paper drawings like the one reproduced in 2.1.

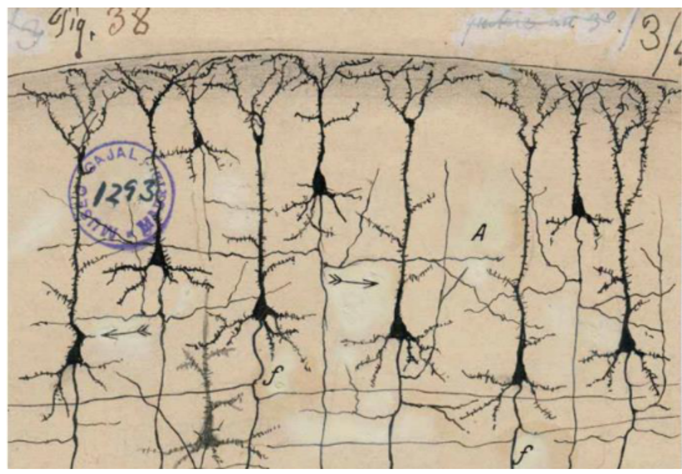

Figure 2.1: Neurons in the cerebral cortex, a part of the mammalian brain

One can distinguish the cell bodies of neural cells, their axons (f), and their dendrites. The axons of some neurons connect to the dendrites of other neurons, forming a neural network A schematic image of a neuron is drawn in 2.2. Information is processed from left to right. On the left are the dendrites that receive signals and connect to the cell body of the neuron where the signal is processed.

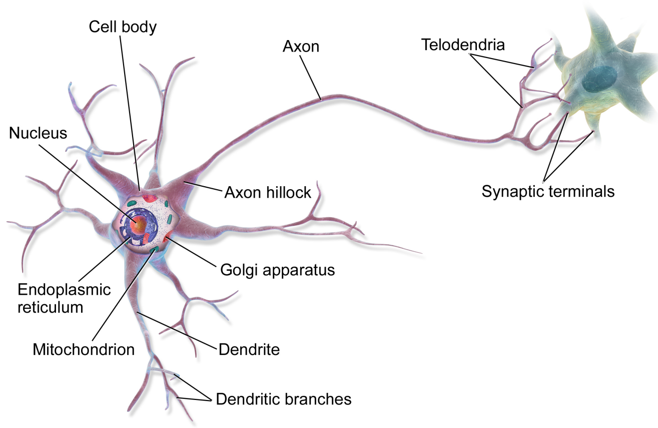

Figure 2.2: Schematic image of a neuron.

The right part of the Figure shows the axon, through which the output is sent to other neurons. The axon connects to their dendrites via synapses. Information is transmitted as an electrical signal. 2.3 shows an example of the time series of the electric potential for a pyramidal neuron in fish. The time series consists of an intermittent series of electrical-potential spikes. Quiescent periods without spikes occur when the neuron is inactive, during spike-rich periods we say that the neuron is active.

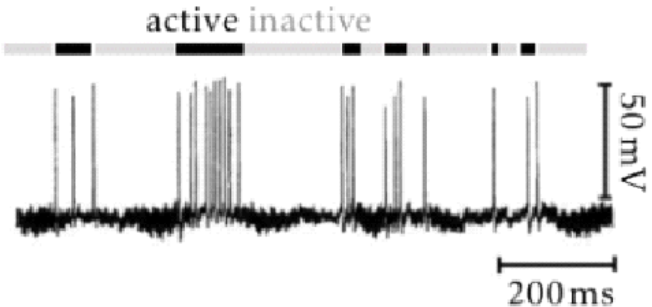

Figure 2.3: Spike train in the electro-sensory pyramidal neuron of a fish

3 Chapter Theory

3.1 Neural Networks

In this section, we will delve into the topic of single-layer neural networks, which includes some of the classical approaches to neural computing and learning problems. We will first discuss the representational power of single-layer networks and their learning algorithms, providing examples of their usage. Subsequently, we will address the representational limitations encountered by single-layer networks.

Two classical models will be described in the first part of this chapter: the Perceptron, proposed by Rosenblatt in the late 1950s [1] An artificial neural network is an application that is non-linear with respect to its parameters θ, and it associates an input x with an output $y = f(x, \theta)$. For simplicity, we assume that y is unidimensional, although it could also be multi-dimensional. The function f has a specific form that will be explained. Neural networks can be used for regression or classification tasks. The parameters θ are estimated from a learning sample, and the optimization function is non-convex, which can result in finding local minimizers. The success of neural networks stems from the universal approximation theorem proposed by Cybenko (1989) and Hornik (1991) [1, 12]. Furthermore, LeCun (1986) introduced an efficient method called backpropagation of the gradient, which enables the computation of gradients for neural networks and facilitates the attainment of local minimizers for quadratic criteria [13]. The use of neural networks, with their flexibility in representation and their efficient optimization techniques, has contributed significantly to the advancement of artificial intelligence and machine learning.

3.1.1 Artificial Neuron

An artificial neuron, denoted as f_j, is a function of the input $x = (x_1, \ldots, x_d)$ weighted by a vector of connection weights $w_j = (w_{j,1}, \ldots, w_{j,d})$, along with a neuron bias b_j. This neuron is associated with an activation function φ, and its output y_j is computed as follows:

$$y_j = f_j(x) = \varphi\left(\sum_{i=1}^{d} w_{j,i} x_i + b_j\right). \tag{3.1}$$

Various activation functions can be considered for the artificial neuron. Some commonly used activation functions include:

- The identity function:

$$\varphi(x) = x \tag{3.2}$$

- The sigmoid function (or logistic function):

$$\varphi(x) = \frac{1}{1 + \exp(-x)} \tag{3.3}$$

- The hyperbolic tangent function :

$$\tanh : \varphi(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} = \frac{\exp(2x) - 1}{\exp(2x) + 1} \tag{3.4}$$

- The hard threshold function:

$$\varphi_\beta(x) = \begin{cases} 1, & \text{if } x \geq \beta \\ 0, & \text{otherwise} \end{cases} \tag{3.5}$$

- The Rectified Linear Unit (ReLU) activation function:

$$\varphi(x) = \max(0, x) \tag{3.6}$$

The artificial neuron can be represented schematically as follows, where $\Sigma = \langle w_j, x \rangle + b_j$:

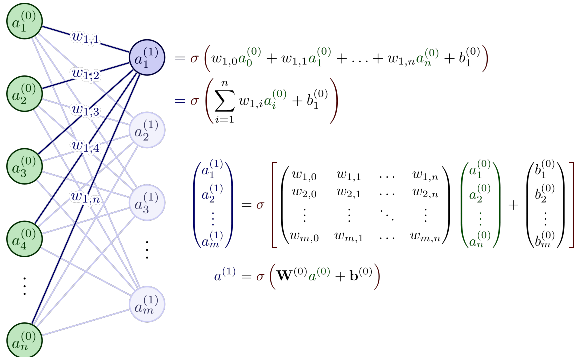

Figure 3.1: Schematic representation of an artificial neuron.

These activation functions introduce non-linearity into the neural network, enabling it to learn complex patterns and make non-linear decisions. The choice of activation function depends on the nature of the problem and the desired behavior of the neural network.

3.1.2 Activation and output rules

after we introduce a general Overview of Neural networks we need to understand the connectivity between neurons which leads us to activation functions [14] [15], moreover, We also need a rule which gives the effect of the total input on the activation of the unit. We need a function F_k which takes the total input $s_k^{(t)}$ and the current activation $y_k^{(t)}$ and produces a new value of the activation of the unit k:

$$y_k^{(t+1)} = F_k(y_k^{(t)}, s_k^{(t)}) \tag{3.7}$$

Often, the activation function is a non-decreasing function of the total input of the unit:

$$y_k^{(t+1)} = F_k(s_k^{(t)}) \tag{3.8}$$

In some cases, the output of a unit can be a stochastic function of the total input of the unit. In that case, the activation is not deterministically determined by the neuron input, but the neuron input determines the probability p that a neuron gets a high activation value:

$$p(y_k = 1 | s_k) = \sigma(s_k) \tag{3.9}$$

Historically, the sigmoid function was mostly used as the activation function since it is differentiable and allows us to keep values in the interval [0, 1]. Nevertheless, it has a problem: its gradient is very close to 0 when $|x|$ is not close to 0. Figure 3 represents the sigmoid function and its derivative. With neural networks with a high number of layers (which is the case for deep learning), this causes trouble for the backpropagation algorithm to estimate the parameters (backpropagation is explained in the following). This is why the sigmoid function was supplanted by the rectified linear function. This function is not differentiable at 0, but in practice, this is not really a problem since the probability to have an entry equal to 0 is generally null. The ReLU function also has a sparsification effect. The ReLU function and its derivative are equal to 0 for negative values, and no information can be obtained in this case for such a unit. This is why it is advised to add a small positive bias to ensure that each unit is active. Several variations of the ReLU function are considered to make sure that all units have a non-vanishing gradient [16] and that for $x < 0$, the derivative is not equal to 0. Namely

$$\varphi(x) = \max(x, 0) + \alpha \min(x, 0) \tag{3.10}$$

where α is either a fixed parameter set to a small positive value, or a parameter to estimate.

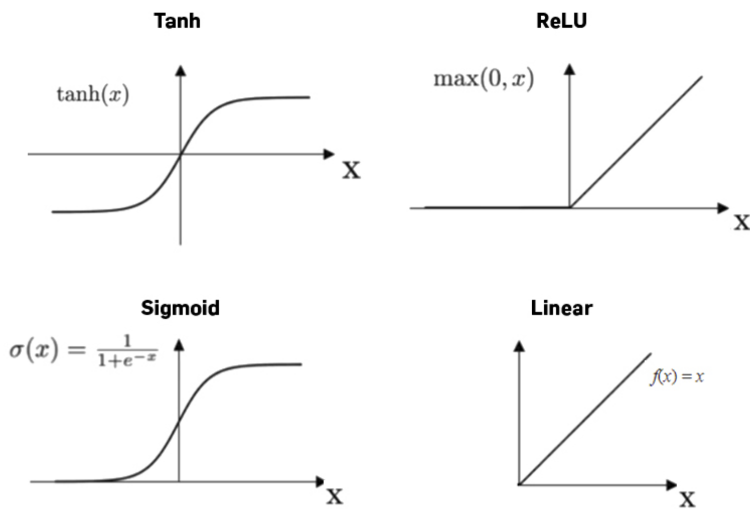

Figure 3.2: Various activation functions for a unit

3.2 Multi-Layer Perceptron

A multi-layer perceptron [17] (or neural network) is a structure composed of several hidden layers of neurons, where the output of a neuron in one layer becomes the input of a neuron in the next layer. Additionally, the output of a neuron can also be the input of a neuron in the same layer or in previous layers (this is the case for recurrent neural networks). The last layer called the output layer, may apply a different activation function compared to the hidden layers, depending on the type of problem at hand, such as regression or classification. Figure 4 represents a neural network with three input variables, one output variable, and two hidden layers.

Multi-layer perceptrons shown in Fig 3.3 have a basic architecture where each unit (or neuron) in a layer is connected to all the units in the next layer but has no direct

connection with the neurons in the same layer. The architectural parameters include the number of hidden layers and the number of neurons in each layer. The choice of activation functions is also left to the user. For the output layer, as mentioned earlier, the activation function is generally different from the ones used in the hidden layers.

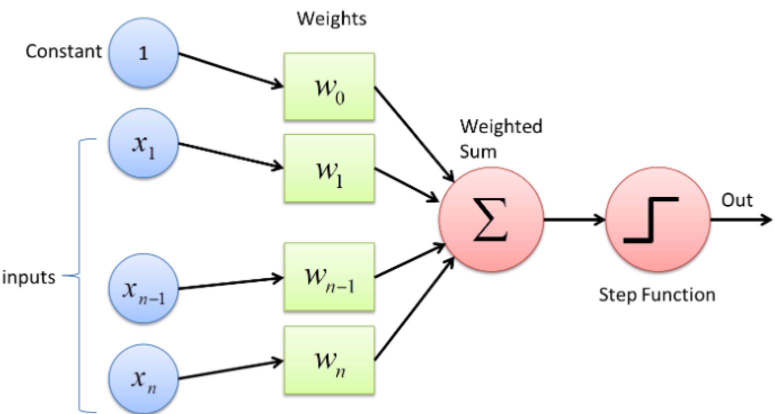

Figure 3.3: basic neural network multi-layer perceptron

In regression tasks, no activation function is applied to the output layer. For binary classification, where the output provides a prediction of $P(Y = 1/X)$ (since this value is in the range $[0,1]$), the sigmoid activation function is commonly used. In multi-class classification, the output layer contains one neuron per class i, giving a prediction of $P(Y = i/X)$. The sum of these predicted probabilities should be equal to 1. The multidimensional softmax function is typically employed for this purpose:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Let us summarize the mathematical formulation of a multi-layer perceptron with L hidden layers. We set $h^{(0)}(x) = x$.

For $k = 1, \ldots, L$ (hidden layers):

$$a^{(k)}(x) = b^{(k)} + W^{(k)} h^{(k-1)}(x)$$
$$h^{(k)}(x) = \varphi(a^{(k)}(x))$$

For $k = L + 1$ (output layer):

$$a^{(L+1)}(x) = b^{(L+1)} + W^{(L+1)} h^{(L)}(x)$$
$$h^{(L+1)}(x) = \psi(a^{(L+1)}(x)) := f(x, \theta)$$

where φ is the activation function and ψ is the output layer activation function (e.g., softmax for multiclass classification). At each step, $W^{(k)}$ is a matrix with the number of rows equal to the number of neurons in layer k and the number of columns equal to the number of neurons in layer $k - 1$.

3.3 Tensor Foundation

Computing derivatives of tensor expressions, also known as tensor calculus, is a fundamental task in machine learning. A key concern is the efficiency of evaluating the expressions

and their derivatives that hinge on the representation of these expressions. Recently, an algorithm for computing higher-order derivatives of tensor expressions like Jacobians or Hessians has been introduced that is a few orders of magnitude faster than previous state-of-the-art approaches. Unfortunately, the approach is based on Ricci notation and hence cannot be incorporated into automatic differentiation frameworks from deep learning like TensorFlow, PyTorch, Nno-Autograd, or JAX that use the simpler Einstein notation. This leaves two options, to either change the underlying tensor representation in these frameworks or to develop a new, provably correct algorithm based on Einstein notation. Obviously, the first option is impractical. Hence, we pursue the second option. Here, we show that using Ricci notation is not necessary for an efficient tensor calculus and develop an equally efficient method for the simpler Einstein notation. It turns out that turning to Einstein notation enables further improvements that lead to even better efficiency.

3.3.1 Einstein notation

In tensor calculus, three types of multiplication are distinguished: inner, outer, and element-wise multiplication. Indices play a crucial role in distinguishing between these types. Given tensors A, B, and C, any multiplication of A and B can be expressed as:

$$C[s_3] = \sum_{(s_1 \cup s_2) \setminus s_3} A[s_1] \cdot B[s_2]$$

Here, C represents the result tensor, s_1, s_2, and s_3 are the index sets of the left argument, the right argument, and the result tensor, respectively. The summation symbol is relevant only for inner products, which are denoted by shared upper and lower indices in Ricci calculus. If we prefer not to distinguish between upper and lower indices, the summation must be explicitly included through the result tensor. The standard practice for achieving this is to exclude the index for summation from the index set of the result tensor. Therefore, the index set of the result tensor is always a subset of the union of the index sets of the multiplication's arguments, i.e., $s_3 \subseteq (s_1 \cup s_2)$.

In the following discussion, we denote the generic tensor multiplication simply as $C = A *(s_1, s_2, s_3) B$, where s_3 explicitly represents the index set of the result tensor. This notation is essentially identical to the tensor multiplication `einsum` in libraries such as NumPy, TensorFlow, PyTorch, and the Tensor Comprehension Package [18].

The $*(s_1, s_2, s_3)$-notation closely resembles standard Einstein notation. In Einstein notation, the index set s_3 of the output is omitted, and the convention is to sum over all shared indices in s_1 and s_2. However, this restriction limits the types of multiplications that can be represented. The set of multiplications representable in standard Einstein notation is a proper subset of the set representable by our notation. For example, standard Einstein notation cannot directly represent element-wise multiplications. Nonetheless, in the following discussion, we refer to the $*(s_1, s_2, s_3)$-notation simply as Einstein notation, as it is the standard practice in all deep learning frameworks. Table 3.1 provides a comparison of different linear algebra notations.

Table 3.1 shows examples of tensor expressions in standard linear algebra notation, Ricci calculus, and Einstein notation. The first group demonstrates an outer product, the second group shows inner products, and the last group provides examples of element-wise multiplications. As seen in Table 3.1, Ricci notation and Einstein notation are syntactically similar. However, they have significant semantic differences. Ricci notation distinguishes between co- and contravariant dimensions/indices, whereas Einstein notation does not. Although this may seem like a minor difference, it has substantial implications when

Table 3.1: Comparison of different linear algebra notations.

Ricci Notation	Einstein Notation	Vectorized Notation
$y^i x_j$	$y * (i, j, ij)x$	yx
$A_{ij} x_j$	$A * (ij, j, i)x$	Ax
$y^i x_i$	$y * (i, i, \emptyset)x$	$y \cdot x$
$A_{ij} B_{jk}$	$A * (ij, jk, ik)B$	AB
$y^i x_i$	$y * (i, i, i)x$	$y \cdot x$
$A_{ij} \mathrm{diag}(x)$	$A * (ij, i, ij)x$	$A \cdot \mathrm{diag}(x)$

computing derivatives. For example, when using Ricci notation, both forward and reverse mode automatic differentiation can be treated in the same way [5], but this is not the case with Einstein notation.

We can demonstrate that the generic tensor multiplication operator $*(s_1, s_2, s_3)$ is associative, commutative, and satisfies the distributive property. Our tensor calculus, which we introduce in the next section, relies on all three properties. By $s_1 s_2$, we denote the concatenation of the index sets s_1 and s_2. An example where the concatenation of two index sets is used is the outer product of two vectors, as shown in the first row of Table 3.1.

Lemma 2.1. (Associativity) Let s_1, s_2, s_3, and s_4 be index sets with $s_3 \subseteq s_1 \cup s_2$ and $s_4 \cap (s_1 \cup s_2) = \emptyset$. Then it holds that

$$(A * (s_1, s_2 s_4, s_3 s_4)B) * (s_3 s_4, s_4, s_3)C = A * (s_1, s_2, s_3)(B * (s_2 s_4, s_4, s_2)C)$$

Proof: We have

$$\begin{aligned}
&(A * (s_1, s_2 s_4, s_3 s_4)B) * (s_3 s_4, s_4, s_3)C \\
&= \sum_{s_4} \left(\sum_{(s_1 \cup s_2) \setminus s_3} A[s_1] \cdot B[s_2 s_4] \right) \cdot C[s_4] \\
&= \sum_{((s_1 \cup s_2) \setminus s_3) \cup s_4} A[s_1] \cdot B[s_2 s_4] \cdot C[s_4] \\
&= \sum_{(s_1 \cup s_2) \setminus s_3} A[s_1] \left(\sum_{s_4} B[s_2 s_4] \cdot C[s_4] \right) \\
&= A * (s_1, s_2, s_3)(B * (s_2 s_4, s_4, s_2)C)
\end{aligned}$$

Lemma 2.2. (Commutativity) It holds that

$$A * (s_1, s_2, s_3)B = B * (s_2, s_1, s_3)A$$

Proof: This follows immediately from our definition of the tensor multiplication operator $*(s_1, s_2, s_3)$, the commutativity of the scalar multiplication, and the commutativity of the set union operation.

Lemma 2.3. (Distributive property) Let s_1, s_2, and s_3 be index sets with $s_3 \subseteq s_1 \cup s_2$. It holds that

$$A * (s_1, s_2, s_3)B + A * (s_1, s_2, s_3)C = A * (s_1, s_2, s_3)(B + C)$$

Proof: This follows from the distributive property of scalar multiplication.

3.4 Tensor calculus

Now we are prepared to develop our tensor calculus. We start by giving the definition of the derivative of a tensor-valued expression with respect to a tensor. For the definition, we use $\|A\| = \sqrt{\sum_s A[s]^2}$ as the norm of a tensor A, which coincides with the Euclidean norm if A is a vector and with the Frobenius norm if A is a matrix.

Definition 3.1. (Fréchet Derivative) Let $f : \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_k} \to \mathbb{R}^{m_1 \times m_2 \times \ldots \times m_l}$ be a function that takes an order-k tensor as input and maps it to an order-l tensor as output. Then, $D \in \mathbb{R}^{m_1 \times m_2 \times \ldots \times m_l \times n_1 \times n_2 \times \ldots \times n_k}$ is called the derivative of f at x if and only if

$$\lim_{h \to 0} \frac{\|f(x+h) - f(x) - D \circ h\|}{\|h\|} = 0,$$

where $\circ$ is an inner tensor product.

Here, the dot product notation $D \circ h$ is short for the inner product $D *(s_1 s_2, s_2, s_1) h$, where $s_1 s_2$ is the index set of D and s_2 is the index set of h. For instance, if $D \in \mathbb{R}^{m_1 \times n_1 \times n_2}$ and $h \in \mathbb{R}^{n_1 \times n_2}$, then $s_1 = \{i, j, k\}$ and $s_2 = \{j, k\}$.

In the following, we first describe forward and reverse mode automatic differentiation for expressions in Einstein notation before we discuss extensions like cross-country mode and compression of higher-order derivatives that are much easier to realize in Einstein notation than in Ricci notation. As can be seen from our experiments in Section 4, these extensions allow for significant performance gains.

3.4.1 Forward Mode

Any tensor expression has an associated directed acyclic expression graph (expression DAG). Figure 1 shows the expression DAG for the expression

$$X^\top \left(\exp(X \cdot w) + 1\right)^{-1} \exp(X \cdot w) \quad (1)$$

where $\cdot$ denotes the element-wise multiplication and $^{-1}$ the element-wise multiplicative inverse.

The nodes of the DAG that have no incoming edges represent the variables of the expression and are referred to as input nodes. The nodes of the DAG that have no outgoing edges represent the functions that the DAG computes and are referred to as output nodes.

Let the DAG have n input nodes (variables) and m output nodes (functions). We label the input nodes as $x_0, \ldots, x_{n-1}$, the output nodes as $y_0, \ldots, y_{m-1}$, and the internal nodes as $v_0, \ldots, v_{k-1}$. Every internal and every output node represents either a unary or a binary operator. The arguments of these operators are supplied by the incoming edges.

In forward mode, for computing derivatives with respect to the input variable x_j, each node v_i will eventually store the derivative $\frac{\partial v_i}{\partial x_j}$ which is traditionally denoted as $\dot{v}_i$. It is computed from input to output nodes as follows: At the input nodes that represent the variables x_i, the derivatives $\frac{\partial x_i}{\partial x_j}$ are stored. Hence, these are either unit tensors if $i = j$ or zero tensors otherwise. Then, the derivatives

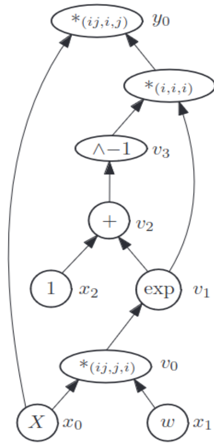

Figure 3.4: Expression DAG for Expression (1)

that are stored at the remaining nodes, here called f , are iteratively computed by summing over all their incoming edges as

$$\dot{f} = \frac{\partial f}{\partial x_j} = \sum_{z:(z,f)\in E} \frac{\partial f}{\partial z} \cdot \frac{\partial z}{\partial x_j} = \sum_{z:(z,f)\in E} \frac{\partial f}{\partial z} \cdot \dot{z},$$

where $\frac{\partial f}{\partial z}$ is the partial derivative of node f with respect to z, and the multiplication is tensorial. The so-called pushforwards $\dot{z}$ of the predecessor nodes z of f have been computed before and are stored at z. Hence, the derivative of each function is stored at the corresponding output node y of the expression DAG. Obviously, the updates can be done simultaneously for one input variable x_j and all output nodes y_i. Computing the derivatives with respect to all input variables requires n such rounds.

In the following, we derive the explicit form of the pushforward for nodes of the expression DAG of a tensor expression. For such a DAG, we can distinguish four types of nodes, namely multiplication nodes, general unary function nodes, element-wise unary function nodes, and addition nodes. General unary functions are general tensor-valued functions, while element-wise unary functions are applied to each entry of a single tensor. The difference can be best explained by the difference between the matrix exponential function (general unary function) and the ordinary exponential function applied to every entry of the matrix (element-wise unary function). The pushforward for addition nodes is trivially just the sum of the pushforward of the two summands. Thus, it only remains to show how to compute the pushforward for multiplication, general unary functions, and element-wise unary function nodes.

Theorem 3.2. Let x be an input variable with index set s_4, and let $C = A * (s_1, s_2, s_3)B$ be a multiplication node of the expression DAG. The pushforward of C is

$$\dot{C} = B * (s_2, s_1s_4, s_3s_4)\dot{A} + A * (s_1, s_2s_4, s_3s_4)\dot{B}.$$

Please note that this is a continuation of the previous LaTeX code I provided. You should combine both parts to have a complete LaTeX document. Let me know if you need any further assistance!

Proof: By the definition of the forward mode, the pushforward $\dot{C}$ is given as

$$\dot{C} = \frac{\partial C}{\partial A} \cdot \dot{A} + \frac{\partial C}{\partial B} \cdot \dot{B}.$$

We show first how to compute $\frac{\partial C}{\partial B} \cdot \dot{B}$. According to Definition 4, it holds that

$$\lim_{h \to 0} \frac{1}{\|h\|} \left\| B(x+h) - B(x) - \dot{B} \circ h \right\| = 0.$$

We have the following sequence of equalities

$$C(x+h) - C(x) - \dot{C} \circ h = A * (s_1, s_2, s_3) B(x+h) - A * (s_1, s_2, s_3) B(x) - \left(A * (s_1, s_2 s_4, s_3 s_4) \dot{B} \right) \circ h$$

$$= A * (s_1, s_2, s_3) B(x+h) - A * (s_1, s_2, s_3) B(x) - \left(A * (s_1, s_2 s_4, s_3 s_4) \dot{B} \right) * (s_3 s_4, s_4, s_3) h$$

$$= A * (s_1, s_2, s_3) B(x+h) - A * (s_1, s_2, s_3) B(x) - A * (s_1, s_2, s_3) \left(\dot{B} * (s_2 s_4, s_4, s_2) h \right)$$

$$= A * (s_1, s_2, s_3) \left(B(x+h) - B(x) - \dot{B} \circ h \right)$$

The first equality follows from the definition of $\dot{C}$, the second from the definition of $\circ$, the third from Lemma 1, the fourth from Lemma 3, and the last from the definition of $\circ$. Thus, we have

$$\lim_{h \to 0} \frac{1}{\|h\|} \left\| C(x+h) - C(x) - \dot{C} \circ h \right\| = \lim_{h \to 0} \frac{1}{\|h\|} \left\| A * (s_1, s_2, s_3) \left(B(x+h) - B(x) - \dot{B} \circ h \right) \right\|$$

$$\leq \|A\| \lim_{h \to 0} \frac{1}{\|h\|} \left\| B(x+h) - B(x) - \dot{B} \circ h \right\| = 0.$$

Hence, we get $\frac{\partial C}{\partial B} \cdot \dot{B} = A * (s_1, s_2 s_4, s_3 s_4) \dot{B}$. Similarly, we get that $\frac{\partial C}{\partial A} \cdot \dot{A} = B * (s_2, s_1 s_4, s_3 s_4) \dot{A}$. Combining the two equalities finishes the proof.

Theorem 3.3. Let x be an input variable with index set s_3, let f be a general unary function whose domain has index set s_1 and whose range has index set s_2, let A be a node in the expression DAG, and let $C = f(A)$. The pushforward of the node C is $\dot{C} = f'(A) * (s_2 s_1, s_1 s_3, s_2 s_3) \dot{A}$, where f' is the derivative of f.

Proof: By Definition 4, we have

$$\lim_{\tilde{h} \to 0} \frac{1}{\|\tilde{h}\|} \left\| f(A + \tilde{h}) - f(A) - f'(A) \circ \tilde{h} \right\| = 0$$

Proof (continued): Let $\tilde{h} = A(x+h) - A(x)$. Since A is differentiable, we have that $\tilde{h} \to 0$ as $h \to 0$. Furthermore, we have that $\|A(x+h) - A(x)\| \leq \frac{1}{c}\|h\|$ for some suitable constant c. Hence, we get

$$0 = \lim_{h \to 0} \frac{1}{\|A(x+h) - A(x)\|} \left\| f(A(x+h)) - f(A) - f'(A) \circ (A(x+h) - A(x)) \right\|$$

$$\geq \lim_{h \to 0} \frac{c}{\|h\|} \left\| f(A(x+h)) - f(A) - f'(A) \circ (A(x+h) - A(x)) \right\| \quad (2)$$

By Definition 4, we also have that

$$\lim_{h \to 0} \frac{1}{\|h\|} \left\| A(x+h) - A(x) - \dot{A} \circ h \right\| = 0.$$

Hence, we can replace $A(x+h) - A(x)$ with $\dot{A} \circ h$ in (2) and obtain

$$0 \geq \lim_{h \to 0} \frac{c}{\|h\|} \left\| \left(f(A(x+h)) - f(A) - f'(A) \circ (\dot{A} \circ h) \right) \right\|.$$

Note that

$$f'(A) \circ (\dot{A} \circ h) = f'(A(x)) \circ (\dot{A} * (s_1 s_3, s_3, s_1) h) = f'(A(x)) * (s_2 s_1, s_1, s_2)(\dot{A} * (s_1 s_3, s_3, s_1) h)$$

$$= \Big(f'(A(x)) * (s_2 s_1, s_1 s_3, s_2 s_3)\dot{A}\Big) * (s_2 s_3, s_3, s_2) h = \Big(f'(A(x)) * (s_2 s_1, s_1 s_3, s_2 s_3)\dot{A}\Big) \circ h.$$

Hence, we obtain

$$0 \geq \lim_{h \to 0} \frac{c}{\|h\|} \left\| \Big(f(A(x+h)) - f(A) - \Big(f'(A(x)) * (s_2 s_1, s_1 s_3, s_2 s_3)\dot{A}\Big) \circ h\Big) \right\|.$$

Thus, we get $\dot{C} = f'(A) * (s_2 s_1, s_1 s_3, s_2 s_3)\dot{A}$ as claimed. ■

Theorem 3.4. Let x be an input variable with index set s_2, let f be an element-wise unary function, let A be a node in the expression DAG with index set s_1, and let $C = f(A)$ where f is applied element-wise. The pushforward of the node C is $\dot{C} = f'(A) * (s_1, s_1 s_2, s_1 s_2)\dot{A}$, where f' is the derivative of f.

Proof (continued): By Definition 4, we have

$$\lim_{h \to 0} \frac{1}{\|h\|} \left\| A(x+h) - A(x) - \dot{A} \circ h \right\| = 0.$$

It follows that for every scalar tensor entry $A(x)_s$, where s is the multi-index of the entry, we have

$$\lim_{h \to 0} \frac{1}{\|h\|} \left| A(x+h)_s - A(x)_s - (\dot{A} \circ h)_s \right| = 0.$$

Proof (continued): Let $f_h(A, x) = f(A(x+h)) - f(A(x))$. Since f is applied entrywise and f' is the derivative of f, we can apply the chain rule for the scalar case, which gives us

$$\lim_{h \to 0} \frac{1}{\|h\|} \left| f_h(A, x)_s - f'(A(x)_s) \cdot (\dot{A} \circ h)_s \right| = 0.$$

Since this equality holds for all multi-indices s, we can sum over these indices and obtain

$$\lim_{h \to 0} \frac{1}{\|h\|} \left\| f_h(A, x) - f'(A(x)) * (\dot{A} \circ h) \right\| = 0.$$

We have

$$f'(A(x)) * (\dot{A} \circ h) = f'(A(x)) * (\dot{A} \circ (A * h)) = (f'(A(x)) * (A * (A * h))) \circ h = (f'(A(x)) * (s1, s1s2, s1s2)\dot{A}) \circ h,$$

where the first and last equalities follow from the definition of $\circ$, and the second equality follows from Lemma 1. Hence, we have

$$\lim_{h \to 0} \frac{1}{\|h\|} \left\| f_h(A, x) - (f'(A(x)) * (s1, s1s2, s1s2)\dot{A}) \circ h \right\| = 0.$$

Thus, we get $\dot{C} = f'(A) * (s1, s1s2, s1s2)\dot{A}$, as claimed.

3.4.2 Reverse Mode

Reverse mode automatic differentiation proceeds similarly to the forward mode, but from output to input nodes. Each node v_i will eventually store the derivative $\frac{\partial y_j}{\partial v_i}$, which is usually denoted as $\bar{v}_i$, where y_j is the function to be differentiated. These derivatives are computed as follows: First, the derivatives $\frac{\partial y_j}{\partial y_i}$ are stored at the output nodes of the DAG. Hence, again, these are either unit tensors if $i = j$ or zero tensors otherwise. Then, the

derivatives that are stored at the remaining nodes, here called z, are iteratively computed by summing over all their outgoing edges as follows:

$$\bar{z} = \frac{\partial y_j}{\partial z} = \sum_{(z,f)\in E} \frac{\partial y_j}{\partial f} \cdot \frac{\partial f}{\partial z} = \sum_{(z,f)\in E} \bar{f} \cdot \frac{\partial f}{\partial z},$$

where the multiplication is again tensorial. The so-called pullbacks $\bar{f}$ have been computed before and are stored at the successor nodes f of z. This means the derivatives of the function y_j with respect to all the variables x_i are stored at the corresponding input nodes of the expression DAG. Computing the derivatives for all the output functions requires m such rounds.

In the following, we describe the contribution of unary and binary operator nodes to the pullback of their arguments. We have only two types of binary operators, namely tensor addition and tensor multiplication. In the addition case, the contribution of C to the pullback of both of its arguments is simply $\bar{C}$. In Theorem 3.5, we derive the explicit form of the contribution of a multiplication node to the pullback of its arguments. In Theorem 3.6, we derive the contribution of a general unary function, and in Theorem 3.7, we derive the contribution of an element-wise unary function node to its argument.

Theorem 3.4.1 *Let Y be an output node with index set s_4, and let $C = A^*(s_1, s_2, s_3)B$ be a multiplication node of the expression DAG. Then the contribution of C to the pullback $\bar{B}$ of B is $\bar{C}^*(s_4s_3, s_1, s_4s_2)A$, and its contribution to the pullback $\bar{A}$ of A is $\bar{C}^*(s_4s_3, s_2, s_4s_1)B$.*

Proof: Here we only derive the contribution of C to the pullback $\bar{B}$. Its contribution to $\bar{A}$ can be computed analogously. The contribution of C to $\bar{B}$ is $\bar{C} \cdot \frac{\partial C}{\partial B}$. By Definition 3.1, we have for the derivative $\bar{C} = \frac{\partial Y}{\partial C}$ of Y with respect to C that

$$\lim_{\tilde{h}\to 0} \frac{1}{\|\tilde{h}\|} \left\| Y(C+\tilde{h}) - Y(C) - \bar{C} \circ \tilde{h} \right\| = 0.$$

By specializing $\tilde{h} = A * (s_1, s_2, s_3)h$, we get

$$\begin{aligned} Y(C+\tilde{h}) - Y(C) - \bar{C} \circ \tilde{h} &= Y(A * (s_1, s_2, s_3)B + A * (s_1, s_2, s_3)h) \\ &\quad - Y(A * (s_1, s_2, s_3)B) - \bar{C} \circ (A * (s_1, s_2, s_3)h) \\ &= Y(A * (s_1, s_2, s_3)(B+h)) - Y(A * (s_1, s_2, s_3)B) \\ &\quad - \bar{C} * (s_4s_3, s_3, s_4)(A * (s_1, s_2, s_3)h) \\ &= Y(A * (s_1, s_2, s_3)(B+h)) - Y(A * (s_1, s_2, s_3)B) \\ &\quad - (\bar{C} * (s_4s_3, s_1, s_4s_2)A) * (s_4s_2, s_2, s_4)h \\ &= Y(A * (s_1, s_2, s_3)(B+h)) - Y(A * (s_1, s_2, s_3)B) \\ &\quad - (\bar{C} * (s_4s_3, s_1, s_4s_2)A) \circ h. \end{aligned}$$

Furthermore, since f' is the derivative of f, we have

$$\lim_{h\to 0} \frac{1}{\|h\|} \left\| \tilde{h} - f'(A) * (s_2s_1, s_1, s_2)h \right\| = 0. \tag{4}$$

Combining Equations (3) and (4) gives

$$\begin{aligned}
0 &= \lim_{\tilde{h}\to 0} \frac{1}{\|\tilde{h}\|} \left\| Y(f+\tilde{h}) - Y(f) - \bar{f} \circ \tilde{h} \right\| \\
&= \lim_{h\to 0} \frac{1}{\|h\|} \left\| Y(f(A+h)) - Y(f(A)) - \bar{f}(A) \circ (f'(A) * (s_2 s_1, s_1, s_2) h) \right\| \\
&= \lim_{h\to 0} \frac{1}{\|h\|} \left\| Y(f(A+h)) - Y(f(A)) - (\bar{f}(A) * (s_3 s_2, s_2, s_3 s_1) f'(A)) * (s_3 s_1, s_1, s_3) h \right\| \\
&= \lim_{h\to 0} \frac{1}{\|h\|} \left\| Y(f(A+h)) - Y(f(A)) - (\bar{f}(A) * (s_3 s_2, s_2, s_3 s_1) f'(A)) \circ h \right\|.
\end{aligned}$$

Hence, the contribution of the node C to the pullback $\bar{A}$ is

$$\frac{\partial Y}{\partial f} \cdot \frac{\partial f}{\partial A} = \bar{f} * (s_3 s_2, s_2 s_1, s_3 s_1) f'(A).$$

In case the general unary function is simply an element-wise unary function that is applied element-wise to a tensor, Theorem 3.6 simplifies as follows.

Proof (continued)

The contribution of the node C to the pullback $\bar{A}$ is $\bar{f} \cdot \frac{\partial f}{\partial A}$. By Definition 4, we have for the derivative $\bar{f} = \frac{\partial Y}{\partial f}$ of Y with respect to f that

$$\lim_{\tilde{h}\to 0} \frac{1}{\|\tilde{h}\|} \left\| Y(f+\tilde{h}) - Y(f) - \bar{f} \circ \tilde{h} \right\| = 0. \tag{3}$$

By specializing $\tilde{h} = f(A+h) - f(A)$ and setting $f = f(A)$, we get

$$\begin{aligned}
Y(f+\tilde{h}) - Y(f) - \bar{f} \circ \tilde{h} &= Y(f(A+h) - f(A) + f(A)) - Y(f(A)) - \bar{f}(A) \circ (f(A+h) - f(A)) \\
&= Y(f(A+h)) - Y(f(A)) - \bar{f}(A) \circ (f(A+h) - f(A)).
\end{aligned}$$

Furthermore, since f' is the derivative of f, we have

$$\lim_{h\to 0} \frac{1}{\|h\|} \left\| \tilde{h} - f'(A) * (s_2 s1, s1, s_2) h \right\| = 0. \tag{4}$$

Combining Equations (3) and (4) gives

$$\begin{aligned}
0 &= \lim_{\tilde{h}\to 0} \frac{1}{\|\tilde{h}\|} \left\| Y(f+\tilde{h}) - Y(f) - \bar{f} \circ \tilde{h} \right\| \\
&= \lim_{h\to 0} \frac{1}{\|h\|} \left\| Y(f(A+h)) - Y(f(A)) - \bar{f}(A) \circ (f'(A) * (s_2 s1, s1, s_2) h) \right\| \\
&= \lim_{h\to 0} \frac{1}{\|h\|} \left\| Y(f(A+h)) - Y(f(A)) - (\bar{f}(A) * (s_3 s2, s2, s_3 s1) f'(A)) * (s_3 s1, s1, s_3) h \right\| \\
&= \lim_{h\to 0} \frac{1}{\|h\|} \left\| Y(f(A+h)) - Y(f(A)) - (\bar{f}(A) * (s_3 s2, s2, s_3 s1) f'(A)) \circ h \right\|.
\end{aligned}$$

Hence, the contribution of the node C to the pullback $\bar{A}$ is

$$\frac{\partial Y}{\partial f} \cdot \frac{\partial f}{\partial A} = \bar{f} * (s_3 s2, s2, s_3 s1) f'(A).$$

In case the general unary function is simply an element-wise unary function that is applied element-wise to a tensor, Theorem 3.6 simplifies as follows:

Theorem 3.7. Let Y be an output function with index set s_2, let f be an element-wise unary function, let A be a node in the expression DAG with index set s_1, and let $C = f(A)$ where f is applied element-wise. The contribution of the node C to the pullback $\bar{A}$ is

$$\bar{f} * (s_2 s1, s1, s_2 s1) f'(A),$$

where f' is the derivative of f.

Theorem 3.7. Let Y be an output function with index set s_2, let f be an element-wise unary function, let A be a node in the expression DAG with index set s_1, and let $C = f(A)$ where f is applied element-wise. The contribution of the node C to the pullback $\bar{A}$ is

$$\bar{f} * (s_2 s_1, s_1, s_2 s_1) f'(A),$$

where f' is the derivative of f.

Proof:

The contribution of the node C to the pullback $\bar{A}$ is $\bar{f} \cdot \frac{\partial f}{\partial A}$. By Definition 4, we have for the derivative $\bar{f} = \frac{\partial Y}{\partial f}$ of Y with respect to f that

$$\lim_{\tilde{h} \to 0} \frac{1}{\|\tilde{h}\|} \left\| Y(f + \tilde{h}) - Y(f) - \bar{f} \circ \tilde{h} \right\| = 0. \tag{5}$$

By specializing $\tilde{h} = f(A + h) - f(A)$ and setting $f = f(A)$, we get

$$\begin{aligned} Y(f + \tilde{h}) - Y(f) - \bar{f} \circ \tilde{h} &= Y(f(A + h) - f(A) + f(A)) - Y(f(A)) - \bar{f}(A) \circ (f(A + h) - f(A)) \\ &= Y(f(A + h)) - Y(f(A)) - \bar{f}(A) \circ (f(A + h) - f(A)). \end{aligned}$$

Furthermore, since f' is the derivative of f and f is an entrywise function, we have

$$\lim_{h \to 0} \frac{1}{\|h\|} \left\| \tilde{h} - f'(A) * (s_1, s_1, s_1) h \right\| = 0. \tag{6}$$

Combining Equations (5) and (6) gives

$$\begin{aligned} 0 &= \lim_{\tilde{h} \to 0} \frac{1}{\|\tilde{h}\|} \left\| Y(f + \tilde{h}) - Y(f) - \bar{f} \circ \tilde{h} \right\| \\ &= \lim_{h \to 0} \frac{1}{\|h\|} \left\| Y(f(A + h)) - Y(f(A)) - \bar{f}(A) \circ (f'(A) * (s_1, s_1, s_1) h) \right\| \\ &= \lim_{h \to 0} \frac{1}{\|h\|} \left\| Y(f(A + h)) - Y(f(A)) - (\bar{f}(A) * (s_2 s_1, s_1, s_2) f'(A)) * (s_2 s_1, s_1, s_2) h \right\| \\ &= \lim_{h \to 0} \frac{1}{\|h\|} \left\| Y(f(A + h)) - Y(f(A)) - (\bar{f}(A) * (s_2 s_1, s_1, s_2) f'(A)) \circ h \right\|. \end{aligned}$$

Hence, the contribution of the node C to the pullback $\bar{A}$ is

$$\frac{\partial Y}{\partial f} \cdot \frac{\partial f}{\partial A} = \bar{f} * (s_2 s_1, s_1, s_2 s_1) f'(A).$$

3.4.3 Beyond Forward and Reverse Mode

Since the derivative of a function y with respect to an input variable x is the sum over all partial derivatives along all paths from x to y (see, e.g., [7]), we can combine forward and reverse mode. Using $\bar{v} = \frac{\partial y}{\partial v}$ and $\dot{v} = \frac{\partial v}{\partial x}$, we get

$$\frac{\partial y}{\partial x} = \sum_{v \in S} \bar{v} * (s_1 s v, s v s_2, s_1 s_2) \dot{v},$$

where sv is the index set of node v, s_1 is the index set of the output function y, s_2 is the index set of the input node x, and S is the set of nodes in a cut of the expression DAG. General combinations of forward and reverse mode lead to the so-called cross-country mode. We will show that the differentiation of tensor expressions becomes even more efficient by a special instantiation of the cross-country mode and by compressing higher-order derivatives.

Cross-Country Mode. In both forward and reverse mode, derivatives are computed as sums of products of partial derivatives. In general, the time for evaluating the derivatives depends on the order by which the partial derivatives are multiplied. The two modes multiply the partial derivatives in opposite order. Derivatives are multiplied from input to output nodes in forward mode and vice versa in reverse mode.

If the output function is scalar-valued, then reverse mode is efficient for computing the derivative with respect to all input variables. It is guaranteed that evaluating the derivative takes at most six times the time for evaluating the function itself. In practice, usually a factor of two is observed [7]. However, this is no longer true for non-scalar-valued functions. In the latter case, the order of multiplying the partial derivatives has a strong impact on the evaluation time, even for simple functions (see, e.g., Naumann [24]). Reordering the multiplication order of the partial derivatives is known as cross-country mode in the automatic differentiation literature [25]. Finding an optimal ordering is NP-hard [26] in general.

However, it turns out that significant performance gains for derivatives of tensor expressions can be obtained by the re-ordering strategy that multiplies tensors in order of their tensor order, i.e., multiplying vectors first, then matrices, and so on. We illustrate this strategy with the following example:

$$f(x) = B \cdot g(h(Ax)), \tag{7}$$

where A and B are two matrices, x is a vector, and $g(\cdot)$ and $h(\cdot)$ are vector-valued functions that also take a vector as input. The derivative in this case is $B\text{diag}(u)\text{diag}(v)A$, where $u = g'(h(Ax))$, $v = h'(Ax)$, and $\text{diag}(u)$ is the diagonal matrix with u on its diagonal. Reverse mode multiplies these matrices from left to right, while forward mode multiplies them from right to left. However, it is more efficient to first multiply the two vectors u and v element-wise and then multiply the result with the matrices A and B.

Actually, the structure of Example 7 is not contrived but fairly common in second-order derivatives. For instance, consider the expression $\sum g(h(Ax))$, where g and h are as above, and the sum is over the vector components of the vector-valued expression $g(h(Ax))$. Many machine learning problems feature such an expression as a subexpression, where A is a data matrix and the optimization variable x is a parameter vector. The gradient of this expression has the form of Example 7 with $B = A^T$. As can be seen in the experiments in Section 4, reordering the multiplications by our strategy reduces the time for evaluating the Hessian by about 30%.

Compressing Derivatives. Our compression scheme builds on the reordering scheme (cross-country mode) from above and on the simple observation that in forward as well as in reverse mode, the first partial derivative is always a unit tensor. It is either, in reverse mode, the derivative of the output nodes with respect to themselves or, in forward mode, the derivative of the input nodes with respect to themselves. This unit tensor can always be moved to the end of the multiplications if the order of multiplication is chosen exactly as in our cross-country mode strategy that orders the tensors in increasing tensor order. Then, the multiplication with the unit tensor at the end is either trivial, i.e., amounts to a multiplication with a unit matrix that has no effect and thus can be removed, or leads to a compactification of the derivative.

For example, consider the loss function

$$f(U) = \left\|T - UV^T\right\|^2$$

of the non-regularized matrix factorization problem, which is often used for recommender systems [27]. Here, $T \in \mathbb{R}^{n\times n}$, U, $V \in \mathbb{R}^{n\times k}$, and n is usually large while k is small. The Hessian of f is the fourth-order tensor

$$H = 2(V * (ij, ik, jk)V) * (jl, ik, ijkl)I \in \mathbb{R}^{n\times k\times n\times k},$$

where I is the identity matrix. Newton-type algorithms for this problem solve the Newton system, which takes time in $O((nk)^3)$. However, the Hessian can be compressed to $2(V * (ij, ik, jk)V)$, which is a small matrix of size $k \times k$. This matrix can be inverted in $O(k^3)$ time. The performance gain realized by compression can be significant. For instance, solving the compressed Newton system needs only about 10μs, whereas solving the original system needs about 1 second for a problem of size $n = 1000$ and $k = 10$. For more experimental results, please refer to Section 4.

As another example, consider a simple neural net with a fixed number of fully connected layers, ReLU activation functions, and a softmax cross-entropy output layer. The Hessian of each layer is a fourth-order tensor that can be written as $A*(ijl, ik, ijkl)I$ for a suitable third-order tensor A. In this case, the Hessian can be compressed from a fourth-order tensor to a third-order tensor.

3.5 Optimizations and computational Graph

This section explores the role of the Universal Approximation Theorem [19], Stochastic Gradient Descent (SGD) [20], Computational Graphs [21], and loss functions in optimizing neural networks. The Universal Approximation Theorem guarantees that neural networks can approximate any bounded and regular function. SGD is an optimization algorithm that adjusts network parameters based on the gradients of a loss function to minimize errors. Computational Graphs provide a graphical representation of operations, facilitating efficient computation and automatic differentiation [22]. The choice of an appropriate loss function is crucial as it quantifies the discrepancy between predicted and target outputs. Together, these components form the foundation for optimizing and training neural networks.

3.5.1 Universal approximation theorem

Hornik (1991) [19] showed that any bounded and regular function $f : \mathbb{R}^d \to \mathbb{R}$ can be approximated at any given precision by a neural network with one hidden layer containing a finite number of neurons, having the same activation function, and one linear output neuron. This result was earlier proved by Cybenko (1989) in the particular case of the sigmoid activation function. More precisely, Hornik's theorem can be stated as follows:

Theorem 1. Let φ be a bounded, continuous, and non-decreasing (activation) function. Let K_d be some compact set in $\mathbb{R}^d$ and $C(K_d)$ the set of continuous functions on K_d. Let $f \in C(K_d)$. Then, for all $\varepsilon > 0$, there exists $N \in \mathbb{N}$, real numbers v_i, b_i, and $\mathbb{R}^d$-vectors w_i such that, if we define

$$F(x) = \sum_{i=1}^{N} v_i\varphi(\langle w_i, x\rangle + b_i)$$

then we have

$$\forall x \in K_d, \quad |F(x) - f(x)| \leq \varepsilon.$$

This theorem is interesting from a theoretical point of view. From a practical point of view, this is not really useful since the number of neurons in the hidden layer may be very large. The strength of deep learning lies in the depth (number of hidden layers) of the networks following 3.5 illustrate

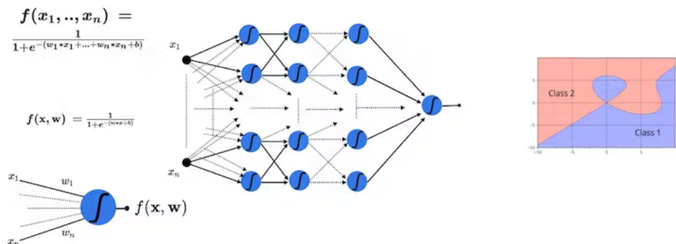

Figure 3.5: Modeling Complex Functions Neural network

3.5.2 Estimation of the parameters

After selecting the architecture of the neural network, the next step is to estimate its parameters, namely the weights (w_j) and biases (b_j), using a learning sample. This estimation is achieved by minimizing a loss function with a gradient descent algorithm. The choice of the loss function is crucial and depends on the specific task at hand. It quantifies the discrepancy between the network's predicted output and the true target values in the training data. The most commonly used loss functions include mean squared error (MSE) for regression problems, binary cross-entropy for binary classification, and categorical cross-entropy for multi-class classification. The selection of the appropriate loss function is guided by the properties of the problem and the desired behavior of the neural network. By minimizing the chosen loss function through techniques like gradient descent, the network's parameters are iteratively adjusted to enhance the model's predictions on the training data. This optimization process aims to find the optimal values for the weights and biases, reducing the difference between the predicted and true values and ultimately improving the accuracy and effectiveness of the neural network.

2.3.2.1 Loss Function

In classical estimation, it is common to maximize the likelihood [23] (or equivalently the logarithm of the likelihood) to estimate the parameters. This is equivalent to minimizing the loss function, which is the negative logarithm of the likelihood. For parameter vector θ estimation, we consider the expected loss function:

$$L(\theta) = -\mathbb{E}_{(X,Y)\sim P}\left(\|Y - f(X,\theta)\|^2\right)$$

If the model is Gaussian, meaning $p_\theta(Y|X = x) \sim \mathcal{N}(f(x,\theta), I)$, maximizing the likelihood is equivalent to minimizing the quadratic loss:

$$L(\theta) = \mathbb{E}_{(X,Y)\sim P}\left(\|Y - f(X,\theta)\|^2\right)$$

For binary classification, where $Y \in \{0, 1\}$, maximizing the log-likelihood corresponds to minimizing the mean squared error. By setting $f(X,\theta) = p_\theta(Y = 1|X)$, the loss function becomes:

$$L(\theta) = -\mathbb{E}_{(X,Y)\sim P}\left[Y \log(f(X,\theta)) + (1-Y)\log(1-f(X,\theta))\right]$$

This loss function is well-suited when using the sigmoid activation function, as the logarithm helps avoid very small gradient values.

Finally, for a multi-class classification problem with k classes, we generalize the previous loss function:

$$L(\theta) = -\mathbb{E}_{(X,Y)\sim P}\left[\sum_{j=1}^{k} 1_{Y=j} \log p_\theta(Y=j|X)\right]$$

Ideally, we would like to minimize the classification error directly [24], but since it is non-smooth, we consider the cross-entropy loss (or a suitable convex surrogate) instead.

2.3.2.2 Stochastic Gradient Descent Algorithm

In optimization problems, Stochastic Gradient Descent (SGD) is a popular and efficient algorithm used to find the minimum of a cost function. It is particularly useful when dealing with large datasets or complex models. SGD belongs to the family of iterative optimization algorithms and is widely used in machine learning and deep learning. The main idea behind SGD is to estimate the gradient of the cost function by using a random subset of training examples at each iteration, instead of considering the entire dataset. This random subset is often referred to as a "mini-batch." By using mini-batches, SGD is able to perform updates to the model parameters more frequently, leading to faster convergence compared to traditional gradient descent methods. SGD follows an iterative update process, where at each iteration, the model parameters are adjusted in the direction that reduces the cost function. The algorithm repeats this process until it reaches convergence or a predefined stopping criterion

Consider the task of minimizing an average of functions:

$$\min_x \frac{1}{n}\sum_{i=1}^{n} f_i(x)$$

This setting is common in machine learning, where the average of functions corresponds to a loss function, and each $f_i(x)$ is associated with the loss term of an individual sample point x_i. The full gradient descent step is given by:

$$x^{(k)} = x^{(k-1)} - t_k \cdot \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(x^{(k-1)}), \quad k = 1, 2, 3, \ldots$$

The idea behind stochastic gradient descent (SGD) is to approximate the full gradient by using a subset of all samples, i.e., a subset of the possible $f_i(x)$'s. following 3.6More formally, the stochastic gradient descent algorithm 2. performs the following iteration:

$$x^{(k)} = x^{(k-1)} - t_k \cdot \nabla f_{i_k}(x^{(k-1)}), \quad k = 1, 2, 3, \ldots$$

where $i_k \in \{1, \ldots, n\}$ is a randomly chosen index at iteration k. Since $\mathbb{E}[\nabla f_{i_k}(x)] = \nabla f(x)$, the estimate is unbiased. The indices i_k are usually chosen without replacement until one full cycle through the entire data set is completed.

Algorithm 2 The Stochastic Gradient Descent algorithm; different choices of the learning rate γ may affect convergence.

Require: initial weights $w^{(0)}$, number of iterations T
Ensure: final weights $w^{(T)}$
1: **for** $t = 0$ **to** $T - 1$ **do**
2: randomly select training sample (X_i, Y_i)
3: compute gradient $\nabla\mathcal{L}(w^{(t)}; X_i, Y_i)$
4: select learning rate γ
5: update weights: $w^{(t+1)} := w^{(t)} - \gamma\nabla\mathcal{L}(w^{(t)}; X_i, Y_i)$
6: **end for**
7: **return** $w^{(T)}$

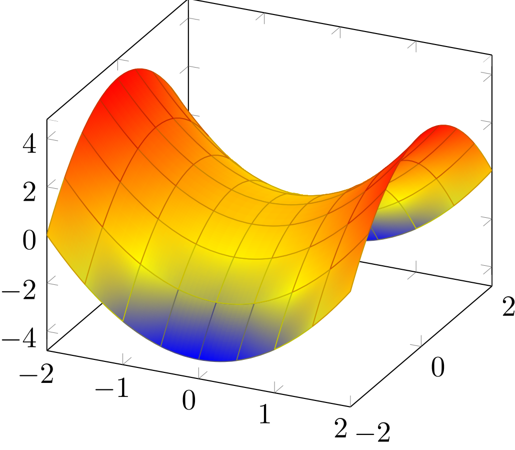

Figure 3.6: Visualization of the SGD optimization landscape. The plot represents the surface $z = x^2 - y^2$ in three dimensions. The goal of the SGD algorithm is to find the minimum of this surface.

3.5.3 Automatic Differentiation

Automatic differentiation is a powerful technique used in computational mathematics and machine learning to efficiently compute derivatives of functions. It provides a systematic way of evaluating derivatives by decomposing complex functions into a series of elementary operations. At its core, automatic differentiation leverages the chain rule of calculus to compute derivatives. By representing functions as compositions of elementary operations, it allows for the calculation of derivatives with respect to input variables or parameters.

The process of automatic differentiation involves building a computational graph that represents the function and its intermediate values. The computational graph is a directed acyclic graph where nodes correspond to operations and edges represent dependencies between them. During the forward pass through the computational graph, each node performs its specific operation and passes the computed value to the next node. This process continues until the final output of the function is obtained. Importantly, the graph also records the derivative of each node with respect to its inputs.

Once the forward pass is completed, the reverse pass, known as back-propagation, is performed to compute the gradients of the function's output with respect to its inputs. The back-propagation process traverses the computational graph in reverse, starting from the output and propagating the derivatives backward through the graph. During back-propagation, each node in the computational graph receives the derivative from the previ-

ous node and multiplies it by its own local derivative. This multiplication corresponds to the application of the chain rule. The computed derivatives are then used to update the parameters of the function through an optimization algorithm, such as gradient descent.

By combining the forward and backward passes, automatic differentiation enables efficient and accurate computation of gradients. It eliminates the need for manual derivation and reduces computational complexity, making it a fundamental technique in various fields, particularly in deep learning. In the following subsections, we will delve into the details of the computational graph and the back-propagation algorithm, which are the key components of automatic differentiation.

2.3.3.1 Computational Graphs

A computational graph is a fundamental concept in the field of automatic differentiation and serves as a graphical representation of a mathematical expression or a computational process. It consists of nodes that represent variables or operations and edges that denote the flow of data between them. In a computational graph, variables are represented as nodes that hold values, while operations are depicted as nodes that perform mathematical computations on their input values. The connections between nodes are directed edges that indicate the flow of data from one node to another. To better understand the concept, let's consider an example. Suppose we have two input variables, denoted as and, and we want to compute their sum. We can construct a computational graph to represent this computation. In this graph, and are represented as input nodes, while the addition operation is depicted as an operation node.

The computational graph for the sum of and would look like this:

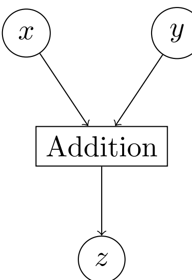

Figure 3.7: Computational graph for the sum of variables x and y.

In this graph, the nodes labeled represent the input variables, while the node labeled represents the addition operation. The output of the addition operation is denoted as , which is another variable node.

By following the connections and operations in the graph, we can compute the value of by evaluating the sum of and. Computational graphs allow us to visualize and understand complex mathematical expressions or computational processes. They provide a clear representation of how variables and operations are interconnected and how data flows through the graph during computation. In the next subsection, we will explore the back-propagation algorithm, which is used to efficiently compute gradients in computational graphs, making it a crucial component of automatic differentiation.

2.3.3.2 Computational Graphs and the Chain Rule of Differentiation

We started by introducing computational graphs as a simple visualization of the flow of data in the previous section within a typical machine learning system (neural networks as

prime examples) by defining the sequence(s) of computations necessary to calculate the end result. The final step in a computational graph (when learning) is the calculation of the loss that quantifies the error that the system makes when processing specific data. These computational graphs provide not only describe the 'forward' flow of the data but in case we are interested in the derivatives of In order to understand automatic differentiation we first have to look at computational graphs. Then using these graphs we look at automatic differentiation

A computational graph describes the flow of data throughout a computation. Consider the expression $z = \log(3x^2 + 5xy)$. In a graph, this can be depicted as shown in the figure below.

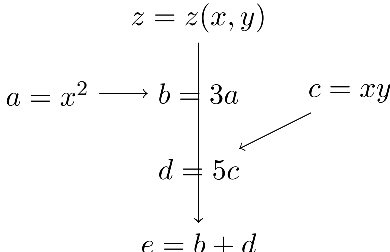

Figure 3.8: Computational graph representing the expression $z = \log(3x^2 + 5xy)$.

We are going to use automatic differentiation to calculate $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$. To start, let's focus on the node representing the expression $z = z(x, y)$, which is part of a larger graph ending up with the final scalar value ℓ. Assuming we have already calculated $\frac{\partial \ell}{\partial z}$, we want to determine the derivatives $\frac{\partial \ell}{\partial x}$ and $\frac{\partial \ell}{\partial y}$.

Applying the chain rule, we have:

$$\frac{\partial}{\partial x}\ell(z(x, y)) = \frac{\partial \ell}{\partial z} \cdot \frac{\partial z}{\partial x},$$

and

$$\frac{\partial}{\partial y}\ell(z(x, y)) = \frac{\partial \ell}{\partial z} \cdot \frac{\partial z}{\partial y}.$$

It should be noted that the notation $\ell(z(x, y))$ can be misleading. It is used here to illustrate that ℓ depends on z. In a complex graph, ℓ may depend on many other values.

The key observation is that for any node in the graph, if we have the derivative of the output ℓ of the entire graph with respect to the output of that particular node, we can calculate the derivative of ℓ with respect to the inputs of that node. This allows us to propagate the derivative calculations from the end of the graph backwards to the beginning. This process is essentially what backpropagation does in neural network learning.

Let's apply these concepts to the example $z = \log(3x^2 + 5xy)$. Using the chain rule and basic derivative rules, we can calculate the partial derivatives as follows:

$$\frac{\partial z}{\partial x} = \frac{6x + 5y}{3x^2 + 5xy},$$

$$\frac{\partial z}{\partial y} = \frac{5x}{3x^2 + 5xy}.$$

To verify these results using the computational graph, we redraw the graph and assign names to all the arrows (values) for ease of reference:

$$
\begin{aligned}
a &= x^2,\\
b &= 3a,\\
c &= xy,\\
d &= 5c,\\
e &= b + d,\\
z &= \log(e).
\end{aligned}
$$

Working from the end to the start, we can calculate the following partial derivatives:

$$
\begin{aligned}
\frac{\partial z}{\partial e} &= \frac{1}{e},\\
\frac{\partial z}{\partial b} &= \frac{\partial z}{\partial e} \cdot \frac{\partial e}{\partial b},\\
\frac{\partial z}{\partial d} &= \frac{\partial z}{\partial e} \cdot \frac{\partial e}{\partial d},\\
\frac{\partial z}{\partial a} &= \frac{\partial z}{\partial b} \cdot \frac{\partial b}{\partial a},\\
\frac{\partial z}{\partial c} &= \frac{\partial z}{\partial d} \cdot \frac{\partial d}{\partial c},\\
\frac{\partial z}{\partial y} &= \frac{\partial z}{\partial c} \cdot \frac{\partial c}{\partial y}.
\end{aligned}
$$

To calculate $\frac{\partial z}{\partial x}$, we observe that the value of x is fed into two nodes in the graph. Therefore, in the backward pass, we need to calculate the derivatives $\frac{\partial a}{\partial x}$, $\frac{\partial z}{\partial a}$, $\frac{\partial c}{\partial x}$, and $\frac{\partial z}{\partial c}$ to obtain $\frac{\partial z}{\partial x}$. Specifically:

$$
\frac{\partial z}{\partial x} = \frac{\partial a}{\partial x} \cdot \frac{\partial z}{\partial a} + \frac{\partial c}{\partial x} \cdot \frac{\partial z}{\partial c} = 2x \cdot \frac{\partial z}{\partial a} + y \cdot \frac{\partial z}{\partial c}.
$$

Substituting the calculated values, we obtain:

$$
\frac{\partial z}{\partial x} = \frac{6x}{3x^2 + 5xy} + \frac{5y}{3x^2 + 5xy} = \frac{6x + 5y}{3x^2 + 5xy}.
$$

Hence, we have successfully reproduced the derivative $\frac{\partial z}{\partial x} = \frac{6x+5y}{3x^2+5xy}$ using the computational graph.

2.3.3.4 Back-Propagation

After understanding how the derivatives flow through a computational graph using the chain rule, we can delve into the process of backpropagation. Backpropagation is a fundamental algorithm used in neural network training to efficiently compute the gradients of the loss function with respect to the network's parameters. The goal of backpropagation is to adjust the parameters of the network in a way that minimizes the loss function. By calculating the gradients of the loss function with respect to each parameter, we can update the parameters in the opposite direction of the gradients, gradually reducing the loss. he process of backpropagation can be summarized in the following steps:

1. Forward Pass: During the forward pass, input data is fed into the network, and the activations of each node in the computational graph are computed layer by layer. The final output of the network is obtained.

2. Loss Calculation: The loss function, which measures the discrepancy between the network's output and the desired output, is evaluated based on the final output.

3. Backward Pass: Starting from the output layer, the derivative of the loss with respect to each parameter is computed using the chain rule. The derivatives are propagated backward through the computational graph, calculating the gradients of the loss function with respect to each node's inputs.

4. Parameter Update: Once the gradients of the loss function with respect to the network parameters are obtained, the parameters are updated using optimization algorithms such as gradient descent or its variants. The update step adjusts the parameters in a direction that minimizes the loss function.

5. Iterative Process: Steps 1 to 4 are repeated iteratively for multiple mini-batches of training examples. The gradients are accumulated over the mini-batches, and the parameter updates are performed to refine the network's weights and biases.

Backpropagation allows for efficient computation of gradients in deep neural networks with multiple layers and millions of parameters. By leveraging the chain rule and propagating the gradients backward through the computational graph, it avoids redundant computations and enables efficient parameter updates general Overview algorithm.

Algorithm 3 Back-propagation Algorithm

Require: Current minibatch of samples (X_i, Y_i)
Ensure: Updated weights and biases
1: **for** $i = 1$ to M **do** ▷ Loop over minibatch samples
2: Perform forward pass: calculate weighted inputs z and activations a for each layer
3: Calculate errors δ for each layer using back-propagation, starting from the last hidden layer
4: Calculate weight gradients: ∇w_i^C and bias gradients: ∇b_i^C using the last two equations
5: **end for**
6: Update weight gradients: $\sum_{i=1}^{M} \nabla w_i^C \to \nabla w^C$ and bias gradients: $\sum_{i=1}^{M} \nabla b_i^C \to \nabla b^C$
7: Update weights and biases using the gradient estimates

Back-Propagation has played a crucial role in the success of deep learning, enabling the training of complex models on large datasets. It has become a cornerstone algorithm for training deep neural networks and has led to significant advancements in various domains, including computer vision, natural language processing, and reinforcement learning. Understanding backpropagation provides insights into the inner workings of neural networks and empowers researchers and practitioners to design and train more effective models.

4 Chapter Methods and Micro-Framework Modeling

n this chapter, we will delve into the building blocks of our micro-framework called Nano-AutoGard, which is based on the NumPy package. We will break down the details into sub-sections to provide a comprehensive explanation of each component. Our goal is to develop a lightweight framework that allows users to understand and implement neural networks from scratch using a minimalistic approach. By leveraging the power of NumPy, a popular numerical computing library in Python, we can efficiently perform matrix operations and automatic differentiation, which are essential for training neural networks.

Throughout this chapter, we will cover various aspects of the Nano-AutoGard framework, including the construction of neural network architectures, the implementation of forward and backward passes, the calculation of gradients using automatic differentiation, and the optimization of network parameters using gradient-based algorithms. Now, let's dive into the sub-sections and explore each part of the Nano-AutoGard micro-framework in detail, starting with the construction of neural network architectures.

4.1 Micro-Framework Modeling

In this sub-section, we introduce our micro-framework called Nano-AutoGrad shown architecture modeling in 4.1. It is a tiny autograd engine that implements backpropagation, also known as reverse-mode auto diff. The engine is built on top of a dynamically built Directed Acyclic Graph (DAG), The Nano-AutoGrad engine consists of approximately 100 lines of code, making it extremely lightweight and easy to understand. It offers the capability to perform automatic differentiation on scalar values within the graph. Each computation in the graph, such as addition, multiplication, and exponentiation, is broken down into its individual components to enable backpropagation. One of the main advantages of Nano-AutoGrad is its educational value. It serves as a useful tool for understanding the inner workings of automatic differentiation and backpropagation algorithms.In the following subsections, we will delve into the key components and functionalities of the Nano-AutoGrad micro-framework, providing a detailed explanation of its implementation.

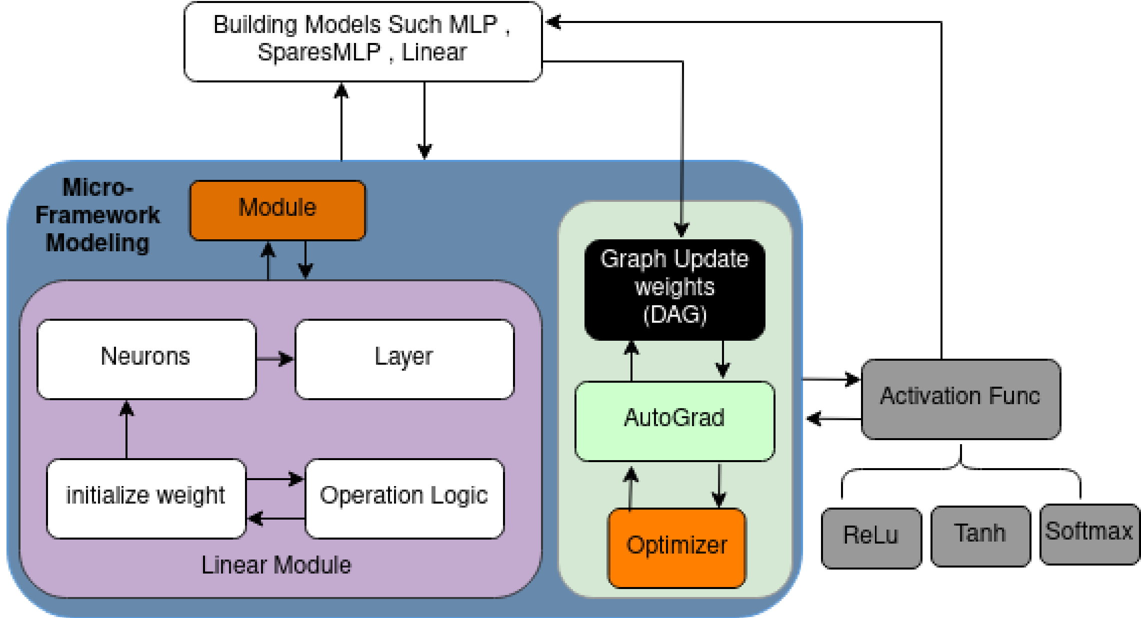

Figure 4.1: Micro-Framework Modeling Nano-AutoGrad (a): Linear Module includes all necessary components to initialize weight Matrix and mathematical logic, (b): AutoGrad is Engine responsible for computing function derivative Order built on top of a dynamically built Directed Acyclic Graph (DAG) (c): Optimizer SGD to update Weight model Through Back-Propagation (d): Activation Func collection of the non-linear function used for connectivity between Layer

4.1.1 Micro-Framework documentation

1. Initialize Weight:

```
class Module:

    def zero_grad(self):
        """
        Set the gradients of all parameters in the module to zero.
        """
        for p in self.parameters():
            p.grad = 0

    def parameters(self):
        """
        Return a list of all parameters in the module.
        """
        return []
```

   initializes the weight by assigning a random value within the range [-1, 1] if no data is provided. It also initializes the gradient to 0.

2. Mathematical Operations:

```
def __add__(self, other):
        other = other if isinstance(other, Value) else Value(other)
        out = Value(self.data + other.data, (self, other), '+')

        def _backward(keep_graph=False):
            self.grad += out.grad
```

```
            other.grad += out.grad
        out._backward = _backward

        return out

def __mul__(self, other):
    other = other if isinstance(other, Value) else Value(other)
    out = Value(self.data * other.data, (self, other), '*')

    def _backward(keep_graph=False):
        if keep_graph:
            self.grad += other * out.grad
            other.grad += self * out.grad
        else:
            self.grad += other.data * out.grad
            other.grad += self.data * out.grad
        out._backward = _backward

        return out

def __pow__(self, other):
        assert isinstance(other, (int, float)), "only supporting int/
    float powers for now"
        out = Value(self.data**other, (self,), f'**{other}')

        def _backward(keep_graph=False):
            if keep_graph:
                self.grad += (other * self**(other-1)) * out.grad
            else:
                self.grad += (other * self.data**(other-1)) * out.grad
        out._backward = _backward

        return out

def __neg__(self): # -self
    return self * -1

def __radd__(self, other): # other + self
    return self + other

def __sub__(self, other): # self - other
    return self + (-other)

def __rsub__(self, other): # other - self
    return other + (-self)
```

This block defines various mathematical operations for the values in the micro-framework. For example, *neg* calculates the negative value, *add* performs addition, sub performs subtraction, and so on. The *repr* method returns a string representation of the value.

3. AutoGrad:

```
def backward(self):

        # topological order all of the children in the graph
        topo = []
        visited = set()
        def build_topo(v):
            if v not in visited:
                visited.add(v)
```

```
9                 for child in v._prev:
10                    build_topo(child)
11                topo.append(v)
12        build_topo(self)
13
14        # go one variable at a time and apply the chain rule to get
    its gradient
15        self.grad = 1
16        for v in reversed(topo):
17            v._backward()
```

This code implements the backward pass or backpropagation in the AutoGrad engine. It creates a topological order of all the nodes in the graph and then applies the chain rule to calculate the gradients of each variable. The 'backward' method sets the gradient to 1 and applies the *backward*function for each variable in the reversed topological order.

1. Function Backward steps:

(a) Initialize an empty list `topo` to store the variables in topological order and a set `visited` to keep track of visited variables.

(b) Define a helper function $build_topo(v)$ that performs a depth-first search (DFS) traversal of the computational graph starting from variable `v`. This function recursively visits all the children of `v` and adds them to the `topo` list.

(c) In the $build_topo$ function:

   i. Check if the variable `v` has been visited before. If not, add it to the `visited` set to mark it as visited.

   ii. For each child variable *child* in $v._prev$ (the children of `v` in the computational graph), recursively call $build_topo(child)$ to visit the child variable and its descendants.

   iii. Append the current variable `v` to the `topo` list.

(d) Call the $build_topo$ function with `self` as the starting variable. This will traverse the computational graph and populate the `topo` list with variables in topological order.

(e) Set the gradient of `self` to 1, indicating the starting point for the chain rule computation.

(f) Iterate over the variables in `topo` in reverse order (from last to first) using the `reversed` function. For each variable `v`, call the $_backward$ method of `v`. This method applies the chain rule to update the gradient of `v` based on its dependencies and the gradients of its children.

By performing the topological sorting, the algorithm ensures that each variable's gradient is computed correctly based on the order of operations in the computational graph. This allows for an efficient and accurate computation of gradients during backpropagation.

Algorithm 4 Topological Sorting

```
1:  procedure BuildTopo(v)
2:      if v is not visited then
3:          Mark v as visited
4:          for each child c in v._prev do
5:              BuildTopo(c)                                  ▷ Recursive call
6:          end for
7:          Append v to topo list
8:      end if
9:  end procedure

10: procedure Backward
11:     Initialize an empty topo list
12:     Initialize an empty visited set
13:     BuildTopo(self)                                       ▷ Build topological order
14:     Set the gradient of self to 1
15:     for each variable v in reverse order of topo do
16:         Call v._backward()                                ▷ Apply chain rule
17:     end for
18: end procedure
```

2. **Algorithm Topology sorting DAG:** The algorithm consists of two procedures BuildTopo and Backward. The BuildTopo procedure performs a depth-first search traversal of the computational graph starting from a variable v. It recursively visits all the children of v and adds them to the *topo* list in topological order. The *visited* set is used to keep track of visited variables to avoid revisiting them.

   The Backward procedure initializes an empty *topo* list and *visited* set. It then calls BuildTopo to build the topological order by traversing the computational graph starting from the variable $self$. The gradient of $self$ is set to 1 to indicate the starting point for the chain rule computation. Finally, it iterates over the variables in reverse order of *topo* and calls their $_backward()$ method to apply the chain rule and update the gradients.

   By following this algorithm, the variables' gradients can be computed correctly and efficiently during the backward pass of the autograd process.

4. Activation Functions:

```python
def relu(self):
        out = Value(0 if self.data < 0 else self.data, (self,), 'ReLU')

        def _backward(keep_graph=False):
            self.grad += (out.data > 0) * out.grad
        out._backward = _backward

        return out

def softmax(self):

        out =  Value(np.exp(self.data) / np.sum(np.exp(self.data), axis=1)[:, None], (self,), 'softmax')
        softmax = out.data
```

```
        def _backward():
            self.grad += (out.grad - np.reshape(
            np.sum(out.grad * softmax, 1),
            [-1, 1]
              )) * softmax
        out._backward = _backward

        return out
```

the defined activation functions used in the micro-framework. The *relu* function calculates the Rectified Linear Unit (ReLU) activation, and the *softmax* function computes the softmax activation. Both functions also include the backward pass calculations for gradient computation.

4.1.2 Building Neural network using Nano-AutoGrad

After we dive into how we built our Nano-Grad a micro-Framework Engine here we will provide some examples of how to build an MLP model and Linear model using Nano-AutoGrad.

Building Neural Network Linear Model

1. **Multi-Layer Perceptron (MLP)** a feed-forward artificial neural network that generates a set of outputs from a set of inputs. An MLP is characterized by several layers of input nodes connected as a directed graph between the input and output layers.

```

from autograd.core.engine import Value
from autograd.core.nn import MLP , Layer , Module


class MLP(Module):

    def __init__(self, nin, nouts):
        sz = [nin] + nouts
        self.layers = [Layer(sz[i], sz[i+1], nonlin=i!=len(nouts)-1)
    for i in range(len(nouts))]

    def __call__(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

    def parameters(self):
        return [p for layer in self.layers for p in layer.parameters()
    ]

    def __repr__(self):
        return f"MLP of [{', '.join(str(layer) for layer in self.
    layers)}]"
```

2. **Linear Model:** module that creates layer feed-forward network with n inputs and m output which we provide similar Pipeline as Pytorch

```

import autograd.torch.nn as nn
import autograd.torch.tensor as Tensor
```

```
import autograd.torch.optim as SGD
import autograd.functiona as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.l1 = nn.Linear(784, 1568, name='l1')
        self.l2 = nn.Linear(1568, 392, name='l2')
        self.l3 = nn.Linear(392, 10, name='l3')

    def forward(self, x):
        z = F.relu(self.l1(x))
        z = F.relu(self.l2(z))
        out = F.log_softmax(self.l3(z))
        return out

model = Model()
optimizer = autograd.optim.SGD(model.parameters(), lr=5e-2,
    weight_decay=1e-4)
scheduler = autograd.optim.lr_scheduler.LinearLR(optimizer,
    start_factor=1.0, end_factor=0.75, total_iters=num_epochs)
```

Training Neural Network Linear Model

1. **Loss function:** a function that calculates the error or discrepancy between predicted and actual values

```
def binary_cross_entropy(input, target):
    """
    Computes the binary cross entropy loss between input and target
    tensors.

    Args:
        input: The input tensor.
        target: The target tensor.

    Returns:
        The binary cross entropy loss.
    """
    return -(target * input.log() + (1 - target) * (1 - input).log()).
    sum() / target.shape[0]


def nll_loss(input, target):
    """
    Computes the negative log likelihood loss between input and target
     tensors.

    Args:
        input: The input tensor.
        target: The target tensor.

    Returns:
        The negative log likelihood loss.
    """
    return -(input * target).sum() / target.shape[0]


def mse_loss(input, target):
    """
```

```
    Computes the mean squared error (MSE) loss between input and
    target tensors.

    Args:
        input: The input tensor.
        target: The target tensor.

    Returns:
        The mean squared error loss.
    """
    return ((input - target) ** 2).mean()


def huber_loss(input, target, delta=1.0):
    """
    Computes the Huber loss between input and target tensors.

    Args:
        input: The input tensor.
        target: The target tensor.
        delta: The threshold for the absolute error.

    Returns:
        The Huber loss.
    """
    error = input - target
    abs_error = abs(error)
    quadratic = 0.5 * (error ** 2)
    linear = delta * (abs_error - 0.5 * delta)
    return np.where(abs_error <= delta, quadratic, linear).mean()
```

2. **Training Model** learning (determining) good values for all the weights and the bias from labeled examples

```
import autograd.torch.tensor as Tensor
import autograd.torch.optim as SGD
import autograd.functiona as F

for k in range(num_epochs):
    accuracy = 0
    train_loss = 0
    for batch in range(num_batches):
        inputs = autograd.Tensor(X_train[batch * batch_size:(batch +
    1) * batch_size])
        labels = autograd.Tensor(np.eye(10)[y_train[batch * batch_size
    :(batch + 1) * batch_size]])

        # Forward
        preds = model(inputs)
        loss = F.nll_loss(preds, labels)

        # Backward
        optimizer.zero_grad()
        loss.backward()

        # Update (SGD)
        optimizer.step()

        accuracy += int(np.count_nonzero(np.argmax(preds.data, axis
    =-1) == np.argmax(labels.data, axis=-1)))
        train_loss += loss.data.item()
```

```
25
26      scheduler.step()
27
28      accuracy /= X_train.shape[0]
29      train_loss /= X_train.shape[0]
30      train_accuracies.append(accuracy)
31      train_losses.append(train_loss)
32
33      with autograd.no_grad():
34          preds_t = model(inputs_t)
35          loss_t = F.nll_loss(preds_t, labels_t).data.item()
36
37      accuracy_t = int(np.count_nonzero(np.argmax(preds_t.data, axis=-1)
         == np.argmax(labels_t.data, axis=-1))) / X_test.shape[0]
38      test_accuracies.append(accuracy_t)
39      test_losses.append(loss_t)
40
41      print(f"Epoch {k+1} loss {train_loss:.6f}, accuracy {accuracy *
        100:.6f}%  test loss {loss_t:.6f}, test accuracy {accuracy_t *
        100:.6f}% lr {optimizer.lr:.6f}")
```

Graph Update weights(DAG): After the model finished training we can get the Graph Update weights which are based on the computational Graph process where the update wights have been calculated through the Back-Propagation algorithm following 4.2

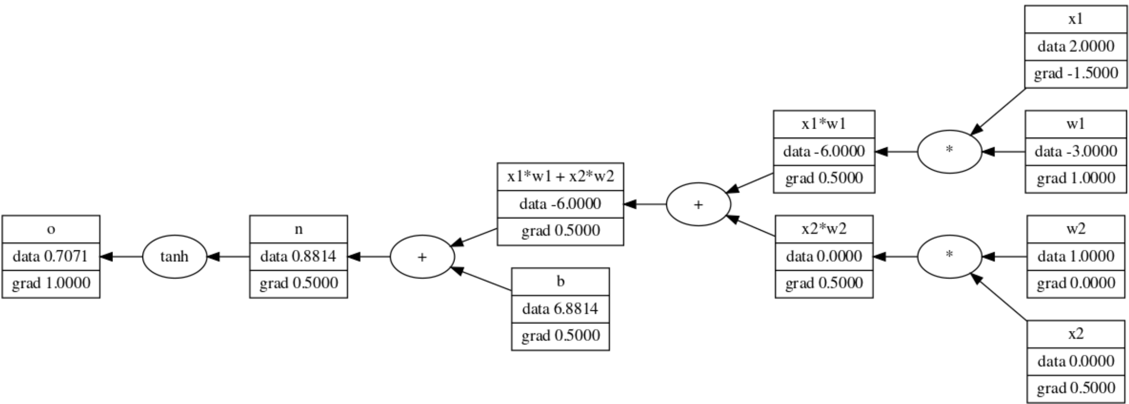

Figure 4.2: Graph Update weights(DAG) Update weights

4.2 Data collection

To demonstrate the capabilities of our Nano-AutoGrad micro-Framework in approximating functions and solving basic problems, we will implement a pipeline using two distinct datasets: MNIST digit recognition and the make_moons dataset from scikit-learn

1. **MNIST Digit Recognition Dataset:** The MNIST dataset 4.3is widely used as a benchmark dataset for image classification tasks. It comprises a collection of 28x28 grayscale images depicting handwritten digits ranging from 0 to 9. The objective is to train a model capable of accurately classifying these digits.

2. **make_moons Dataset:** The make_moons dataset 4.4 is a synthetic dataset provided by the *scikit−learn* library. It is commonly employed for binary classification problems. The dataset consists of 2D points arranged in the shape of two interleav-

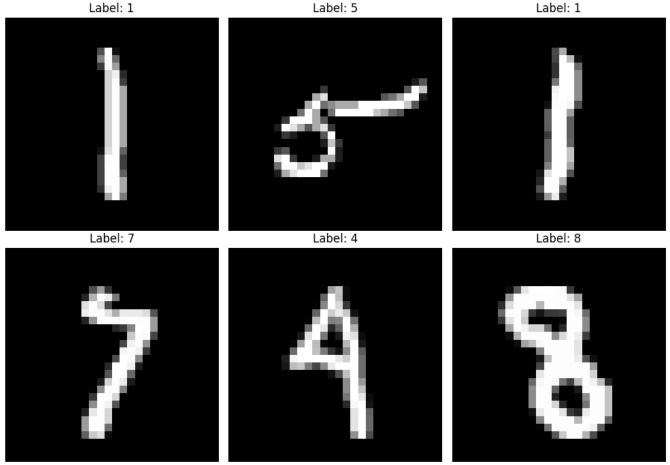

Figure 4.3: Samples Data Minist Digit

ing half-moons. The goal is to learn a model capable of effectively separating the points into their respective classes.

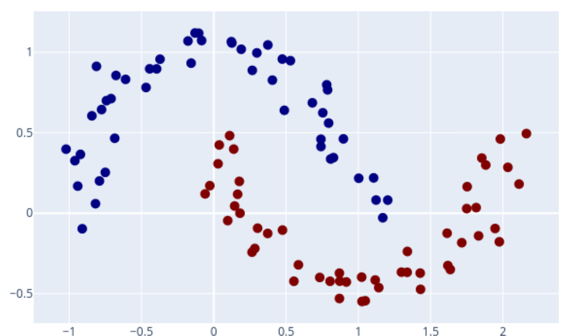

Figure 4.4: Generating Data from make_moons Distribution Function

5 Chapter Experiment

One of the main motivations for developing and deploying AI applications is the desire to provide user-friendly models for making predictions. In our technology stack, we have utilized Gradio, an open-source Python library, to create machine learning and data science demos and web applications 5.1.

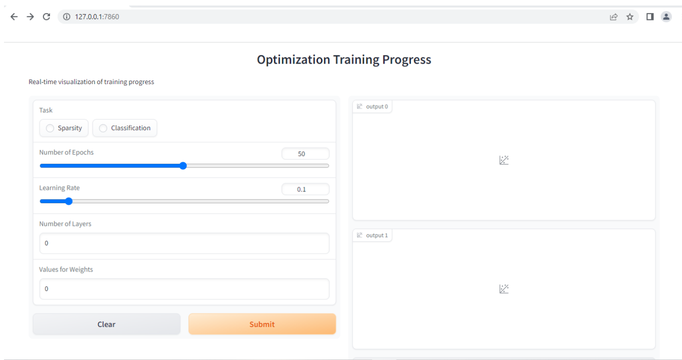

Figure 5.1: our web application interface

Through our application, we have demonstrated that training models and fine-tuning hyperparameters 5.2 can greatly impact their performance.

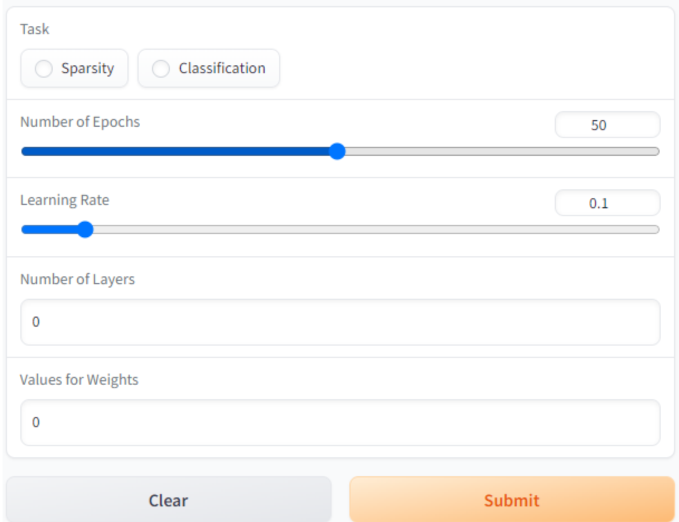

Figure 5.2: Hyper-Parameters Tuning Setting and Specify Task

1. In the experiment Results

1. **Minist Classification Experiment**

Figure 5.3: Training Step Plot of the Loss Function and Accuracy

The figure in Figure 5.3 shows the training step plot of the loss function and accuracy on the Minist dataset. The loss function represents the difference between the predicted and actual values, while the accuracy measures the model's ability to correctly classify the digits. The accuracy can be computed using the following equation:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \times 100\%$$

Figure 5.4: Output Prediction of Digits

Figure 5.4 displays the output predictions of the model on the Minist dataset. Each digit image is classified into its corresponding label, representing the model's ability to recognize and classify handwritten digits.

2. **Sparsity Distribution Data Points 2D**

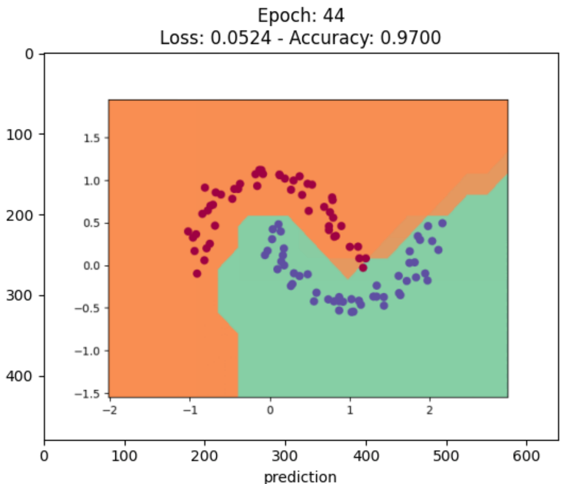

Figure 5.5: Output of Approximation of Sparsity Between 2D Data Point Samples

Figure 5.5 illustrates the output of the model's approximation of sparsity between 2D data point samples. The model identifies and represents the sparsity patterns in the data, allowing for a better understanding and analysis of the distribution.

These visualizations and results demonstrate the performance and capabilities of the Nano-AutoGrad micro-framework on different datasets, including the Minist dataset for digit recognition and the sparsity distribution of 2D data points.

2. In our conclusion from the experiment following

1. **Conclusion 1:** Each specific problem may require a different model design that can effectively handle the data and extract relevant features. It is crucial to choose a model architecture that is well-suited for the task at hand in order to achieve optimal performance.

2. **Conclusion 2:** Hyperparameters play a critical role in model performance. By carefully tuning these hyperparameters, we can improve the model's performance and achieve better results. Researchers have explored various techniques, such as genetic algorithms or random search, to automate the process of hyperparameter tuning and optimize model performance.

6 Chapter Discussion

In this chapter, we will delve into the discussions and key findings related to the Nano-AutoGrad micro-framework. We will analyze its capabilities, limitations, and implications based on our experiments and study. Additionally, we will explore potential use cases, challenges encountered during its implementation, and the significance of our findings.

6.1 Capabilities and Applications of Nano-AutoGrad

Here, we will discuss the capabilities and potential applications of the Nano-AutoGrad micro-framework. We will highlight its usefulness in function approximation, solving binary classification problems, and handling sparsity-related challenges. Furthermore, we will explore its potential for educational purposes and its ability to build deep neural networks.

6.2 Limitations and Challenges

While Nano-AutoGrad shows promise, it is important to acknowledge its limitations and challenges. Some of the limitations and challenges encountered during our study include:

- Limited scalability: Nano-AutoGrad performance may degrade when dealing with large-scale datasets or complex models due to its simplistic implementation.
- Lack of support for advanced operations: The micro-framework may not support advanced mathematical operations or complex neural network architectures, limiting its versatility.
- Efficiency trade-offs: Due to its emphasis on simplicity and educational purposes, Nano-AutoGrad may not be optimized for computational efficiency, leading to potential performance trade-offs.

6.3 Future Directions

Despite its limitations, Nano-AutoGrad opens up avenues for future study and improvements. Some suggestions for future work include:

- Extension of operations and functionalities: Expanding the micro-framework to support a wider range of mathematical operations, advanced neural network architectures, and additional functionalities. leverage their extensive libraries and optimizations.
- Real-world applications and case studies: Applying Nano-AutoGrad to real-world problems and evaluating its performance, robustness, and scalability in various domains.

7 Chapter Conclusion

In this chapter, we will provide a concise summary of the discussions and conclusions drawn from our exploration of the Nano-AutoGrad micro-framework.

7.1 Summary of Findings

Summarize the key findings and insights obtained through our analysis and experiments with Nano-AutoGrad. Emphasize the strengths, limitations, and potential applications of the micro-framework.

7.2 Contributions

Highlight the contributions made by our research to the field of deep learning and education. Discuss the novelty and significance of Nano-AutoGrad as a lightweight autograd engine for function approximation

7.3 Final Remarks

Conclude the Bachelor/project by reflecting on the overall significance of Nano-AutoGrad. Discuss its potential impact on the education of neural networks, its role in understanding autograd engines, and its implications for developing more efficient and accessible deep learning frameworks.

Bibliography

[1] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[2] Minsky Marvin and A Papert Seymour. Perceptrons. *Cambridge, MA: MIT Press*, 6:318–362, 1969.

[3] Venkata Sai Sandeep Yendamuri. *Comparison of numerical properties comparing Automated Derivatives (Autograd) and explicit derivatives (Gradients) for Prototype-based models*. PhD thesis, 2022.

[4] Taylor Arnold and Lauren Tilton. Depth in deep learning: knowledgeable, layered, and impenetrable, 2020.

[5] Rene Y Choi, Aaron S Coyner, Jayashree Kalpathy-Cramer, Michael F Chiang, and J Peter Campbell. Introduction to machine learning, neural networks, and deep learning. *Translational Vision Science & Technology*, 9(2):14–14, 2020.

[6] Daniel A Roberts, Sho Yaida, and Boris Hanin. *The principles of deep learning theory*. Cambridge University Press, 2022.

[7] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[8] Brian Guenter. Efficient symbolic differentiation for graphics applications. In *ACM SIGGRAPH 2007 Papers*, pages 108–es. 2007.

[9] Timothy P Lillicrap, Adam Santoro, Luke Marris, Colin J Akerman, and Geoffrey Hinton. Backpropagation and the brain. *Nature Reviews Neuroscience*, 21(6):335–346, 2020.

[10] Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. Pytorch. *Programming with TensorFlow: Solution for Edge Computing Applications*, pages 87–104, 2021.

[11] G Palm. Warren mcculloch and walter pitts: A logical. In *Brain Theory: Proceedings of the First Trieste Meeting on Brain Theory October 1-4, 1984*, page 229. Springer, 1986.

[12] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[13] Sam Wiseman, Sumit Chopra, Marc'Aurelio Ranzato, Arthur Szlam, Ruoyu Sun, Soumith Chintala, and Nicolas Vasilache. Training language models using target-propagation. *arXiv preprint arXiv:1702.04770*, 2017.

[14] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. In *International Conference on Machine Learning*, pages 3145–3153. PMLR, 2017.

[15] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *Towards Data Sci*, 6(12):310–316, 2017.

[16] Arpan Biswas and Vadim Shapiro. Approximate distance fields with non-vanishing gradients. *Graphical Models*, 66(3):133–159, 2004.

[17] Martin Riedmiller and A Lernen. Multi layer perceptron. *Machine Learning Lab Special Lecture, University of Freiburg*, pages 7–24, 2014.

[18] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[19] Yulong Lu and Jianfeng Lu. A universal approximation theorem of deep neural networks for expressing probability distributions. *Advances in Neural Information Processing Systems*, 33:3094–3105, 2020.

[20] Nikhil Ketkar and Nikhil Ketkar. Stochastic gradient descent. *Deep learning with Python: A hands-on introduction*, pages 113–132, 2017.

[21] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

[22] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research*, 18:1–43, 2018.

[23] Hengshuai Yao, Dong-lai Zhu, Bei Jiang, and Peng Yu. Negative log likelihood ratio loss for deep neural network classification. In *Proceedings of the Future Technologies Conference (FTC) 2019: Volume 1*, pages 276–282. Springer, 2020.

[24] Rene Y Choi, Aaron S Coyner, Jayashree Kalpathy-Cramer, Michael F Chiang, and J Peter Campbell. Introduction to machine learning, neural networks, and deep learning. *Translational Vision Science & Technology*, 9(2):14–14, 2020.