**VIETNAM GENERAL CONFEDERATION OF LABOR**

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**

Logo

Description automatically generated

**MID-TERM REPORT**

**DEEP LEARNING**

***Instructor: LÊ ANH CƯỜNG***

***Student: NGUYỄN DUY ANH***

***LA GIA HIỆP***

***Student ID: 521K0126***

***521K0133***

***Class: 21K50301***

**HO CHI MINH CITY, 2023**

**VIETNAM GENERAL CONFEDERATION OF LABOR**

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**

Logo

Description automatically generated

**MID-TERM REPORT**

**DEEP LEARNING**

***Instructor: LÊ ANH CƯỜNG***

***Student: NGUYỄN DUY ANH***

***LA GIA HIỆP***

***Student ID:521K0126***

***521K0133***

***Class: 21K50301***

**HO CHI MINH CITY, 2023**

# Acknowledgement

We express our sincere gratitude to the teachers of Ton Duc Thang University in Ho Chi Minh City and the Department of Information Technology for their invaluable guidance and support during my academic journey.

We are especially grateful to Mr.Cường for his direct guidance in helping us complete our mid term report.We would also like to extend our appreciation to all our friends from the 21K50301 class, as well as my other friends, brothers, and sisters who have provided constant encouragement and companionship throughout this period.

Above all, we are deeply grateful to our parents for their unwavering dedication and guidance in nurturing and educating us, as well as their unwavering support and care, which has given us the peace of mind to focus on my studies.

Once again, we express our sincere gratitude to all those who have supported us on this journey.

**Table of Contents**

[Acknowledgement 1](#_heading=h.30j0zll)

[**QUESTION 1: 3**](#_heading=)

[1. Introduction: 3](#_heading=h.ihxyk0xf8dnu)

[2. Theoretical Component: 3](#_heading=h.f9cuak47dunu)

# 

# QUESTION 1:

Understanding and Implementing automatic differentiation (Autograd) in Deep Learning in the three following aspects:

## 1. Introduction:

Imagine building a model to predict house prices based on size and location. A simple linear function might suffice if the relationship between these factors and price is relatively linear. However, real-world scenarios often involve intricate, non-linear relationships. Neighborhood quality, hidden market trends, and various other factors can significantly influence price. Here's where simple linear functions fall short.

Deep learning models address this challenge by employing complex functions, typically built upon layers of interconnected artificial neurons. These functions, often referred to as activation functions, can learn intricate, non-linear patterns from data, making them ideal for tasks like:

* Image Recognition: Capturing the subtle variations in shapes, textures, and lighting conditions within an image.
* Natural Language Processing (NLP): Understanding the nuances of human language, including sarcasm, sentiment, and context.
* Speech Recognition: Distinguishing subtle differences in sounds and accents within spoken language.

In all these cases, the complex functions within deep learning models allow them to learn intricate patterns that would be impossible for simpler models. But manually calculating the derivatives (gradients) of these functions with respect to their inputs (weights and biases) would be a nightmare! This is where AD comes in.

## 2. Theoretical Component:

1.1 Automatic Differentiation (AD):

Automatic Differentiation (AD) is a method used to compute derivatives in a unique way. It interprets a computer program differently by adding derivative calculations to the standard computations. The key behind AD lies in applying the principle of algorithmic differentiation, which essentially leverages the chain rule of calculus (Chain Rule allows us to compute the derivative of a composite function by iteratively applying the derivative of each inner function. AND utilizes this rule to automatically compute the derivative of the entire program with respect to its inputs)

1.1.1 How does it work ?

All numerical computations are essentially combinations of a finite set of basic operations. The derivatives of these basic operations are known, and AD combines these derivatives using the chain rule to compute the derivative of the overall function. These functions include binary arithmetic, unary sign switch, and transcendental (functions such as exponential, logarithm, and trigonometric functions.)

1.1.2 Representation and Efficiency

In AD we can represent an algorithm as an evaluation trace of elementary operations. Where we use intermediate variables:

such that for are the input variables,

, are the intermediate variables,

are the output variables.

AD is a powerful tool that can compute derivatives not only for closed-from expression but also for algorithms that use control flow structures such as branching, loops, recursion and procedure calls. This gives AD a significant advantage over symbolic differentiation, which has limitations in handling such complex structures. This is because AD can turn any numeric code into a numeric evaluation trace with specific values of the input, intermediate and output variables.

1.1.3 Why is AD significant in training Deep Learning Models ?

In deep learning, it's crucial for training models by calculating gradients for the loss function concerning the model's parameters (weights and biases). These gradients are used by optimization algorithms to update the parameters and improve the model's performance. In which, AD can handle complex loss functions by effortlessly compute gradients of loss function, without manual intervention making it a prominent technique.

1.2. Forward mode:

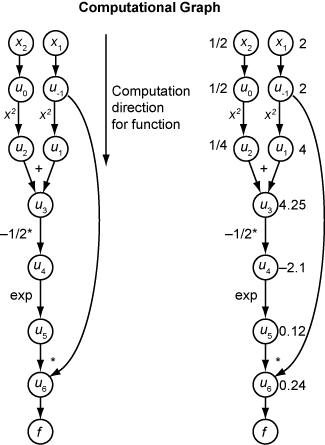
This technique calculates the derivative of the function with respect to the inputs by recording all the operations performed during the program’s execution (think addition, multiplication, applying activation functions). And by utilizing these recorded operations to compute the gradients in a separate pass. This separate pass leverages the chain rule to propagate the derivative information forward through the computational graph

* Computes derivatives along with the function evaluation (forward pass).
* Efficient for calculating many outputs' derivatives w.r.t a single input.
* Less memory efficient for deep models where gradients need to be propagated through many layers.
* May struggle with certain operations that involve control flow (like conditional statements of loops) or function calls where the function itself is not explicitly defined

Consider the problem of evaluating this function and its gradient

AD works at particular points. In this case, take =2,

The following computational graph encodes the calculation of the function f(x)



| Intermediate Variables | Expressions | Values |
| --- | --- | --- |
|  |  | 2 |
|  |  | 0.5 |
|  |  | 4 |
|  |  | 0.25 |
|  | + | 4.25 |
|  |  | -2.1 |
|  |  | 0.12 |
|  | \* | 0.24 |

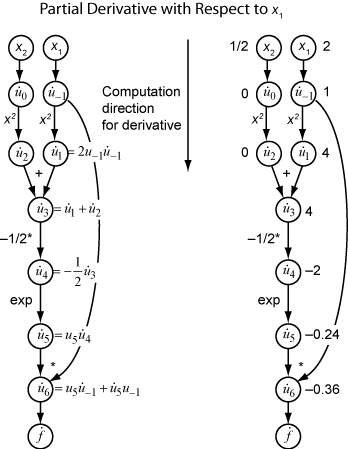
Table 1: Evaluation Trace of

In forward mode, we follow the same graph as the function, moving in the same direction. But here’s the twist, you adjust the calculations based on basic rules of differentiation (which tell you how things change). As you move along the graph, you keep track of the derivative (rate of change) of each part of the function.

To make things easier, we fill in the value of the derivative for each subexpression (denoted as ()) as you go. Think of it like solving a puzzle, you piece together these small derivatives to get the overall gradient. We do this twice: once for each independent variable (like ()) in the function.

The whole process is like applying the chain rule repeatedly. In this case, if you want to find the derivative of () with respect to , you expand it using the derivatives of each (). These () values come from evaluating the function). And as you traverse the graph from top to bottom, all the necessary () values become available

Let represent the derivative of the expression with respect to . Using the evaluated values of the from the function evaluation, you compute the partial derivative of with respect to as shown in the following figure. You can see that all the values of the become available as you traverse the graph from top to bottom



| Intermediate Variables | Expressions | Values | Derivatives |
| --- | --- | --- | --- |
|  |  | 2 | 1 |
|  |  | 0.5 | 0 |
|  |  | 4 | 4 |
|  |  | 0.25 | 0 |
|  | + | 4.25 | 4 |
|  |  | -2.1 | -2 |
|  |  | 0.12 | -0.24 |
|  |  | 0.24 | -0.36 |

Table 2: Evaluation Trace of with derivatives with respect to . Take =2,

0.12 + 2\* 0.12\*(-2) = 0.36

Now to compute the partial derivative with respect to , you traverse the similar computational graph. Therefore, when you compute the gradient of the function, the number of graph traversals is the same as the number of variables. This process is too slow for typical DL applications, which have thousands of millions of variables

1.2.1 Dual Numbers:

In forward-mode AD, a common approach for representing both the value and its derivatives is through dual numbers. It works by simultaneously evaluating the function and its derivatives at a specific input point.

A dual number is an extension of a real number where an additional component represents the derivative of a function evaluated at the original number. It’s written as:

* a is the dual number.
* x is the real number (original variable).
* ϵ is a small fictitious number (often set to 1).
* is the derivative of some function a with respect to another variable x.

Again in forward-mode AD, the input variables become dual numbers, carrying both their value and the derivative (initialized to zero) through the computation graph, accumulating derivative information at each step until the final output holds the function's value and its derivative.

1.3 Reverse mode (backpropagation):

This technique is the more commonly used mode in deep learning. It calculates the derivative of the function with respect to the outputs (typically the loss function). Similar to forward-mode, it records all the operations performed during the program’s execution, building the computational graph. But unlike forward-mode, it then back propagates through the computational graph (traverses the cg backward). During backpropagation, it uses the recorded information about the operations performed and the chain rule to calculate how much that operation contributed to the final error(loss). This information is crucial to compute the gradients for all the weights and biases in the model. Finally updating the weights and biases of the model in the direction that minimizes the loss. Making it more efficient but can be computationally expensive for very deep models

The theory behind reverse mode is also based on the chain rule, along with associated adjoint variables denoted with an overbar. The adjoint variable for is:

The computational graph tracks how each variable influences others. There exists an adjoint version of each variable. When a variable, like affects other variables (and in this case), the arrows leaving it in the graph show how much it contributes to the adjoint variables by its term in the chain rule. There’s this equation that captures this relationship in the graph

In this calculation, with =and =, we have:

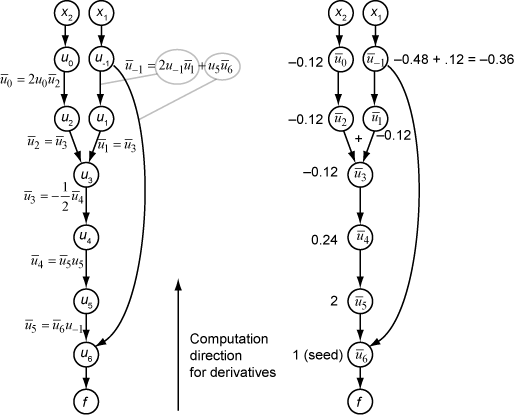
+

Now visualize the computation graph as a maze. First, the program goes through the maze from start to finish (forward traversal), calculating the values of temporary markers (intermediate variables) along the way. Then, it starts back at the end (reverse traversal) with a special clue (seed value ) about how much changing the final result would matter. By following the connections in the maze backwards, it can calculate how much changing each temporary marker would affect the final result in just one pass. Making the Reverse mode approach much faster than going through the maze twice (forward mode) because it reuses the information from the first pass.

Again with this function from the forward mode:

Takes =2,

We again rely on the values which come from evaluating the function on the original computation graph. In the right portion of the figure, the computed values of the adjoint variables appear next to the adjoint variable names, using the formula from the left.



The final gradient values appear as and

| Intermediate Variables | Values | Adjoint |
| --- | --- | --- |
|  | 2 | -0.36 |
|  | 0.5 | -0.12 |
|  | 4 | -0.12 |
|  | 0.25 | -0.12 |
|  | 4.25 | -0.12 |
|  | -2.1 | 0.24 |
|  | 0.12 | 2 |
|  | 0.24 | 1 |

1.6 Comparison of Forward and Backward Mode AD

Advantages of Forward Mode:

* Efficient for Sparse Outputs: When only a few elements contribute to the loss, forward mode can be more efficient by focusing calculations on those elements.
* Can Calculate Jacobians: Forward mode can directly calculate the entire Jacobian matrix, which might be useful for specific tasks like sensitivity analysis.

Disadvantages of Forward Mode:

* High Cost for Dense Outputs: The computational cost of storing and manipulating intermediate activations becomes significant for large output vectors.
* Memory Intensive: Storing intermediate activations can lead to high memory usage, especially for deeper networks.
* Complex Implementation: Implementing forward mode requires additional effort to manage intermediate activations and calculations.

Advantages of Backward Mode:

* Efficient for Dense Outputs: Backpropagation is more efficient for dense output problems where most elements contribute to the loss.
* Lower Memory Usage: Backpropagation only stores gradients, which are typically smaller than the full activations.
* Simpler Implementation: Backpropagation leverages the chain rule for efficient gradient calculation, making it easier to implement.

Disadvantages of Backward Mode:

* Not Ideal for Sparse Outputs: Backpropagation can be less efficient for sparse output problems due to unnecessary calculations for elements not contributing to the loss.
* Limited to Gradients: Backpropagation primarily calculates gradients for training neural networks.

1.5 Computational Graphs

Another way to understand AD is through the lens of Directed Acyclic Graphs (DAGs). The computational graph of the program essentially represents a DAG, where:

* Nodes represent the individual operations within the model. These operations can be basic arithmetic (addition, multiplication), applying activation functions (ReLU, sigmoid), or more complex operations specific to deep learning (like convolutional layers in CNNs)
* Edges represent data flowing between these operations. Edges connect nodes, indicating how the output from one operation becomes the input for another

AD algorithms traverse this graph, applying the chain rule at each node to compute the derivative. In backpropagation, for instance, the algorithm starts at the output node (loss function) and propagates backward through the graph, accumulating the derivative contribution from each operation.

Benefits of Using Computational Graphs:

* Visualization: Computational graphs provide a clear visual representation of a potentially complex deep learning model. This can be helpful for debugging, understanding data flow, and analyzing the behavior of the model.
* Efficiency: The structure of the computational graph allows AD algorithms to efficiently apply the chain rule for gradient computation. This is crucial for training large deep learning models with many parameters.
* Flexibility: Computational graphs are not limited to specific deep learning architectures. As long as the model can be represented as a sequence of operations, AD algorithms can leverage the graph for gradient computation.

In essence, computational graphs act as a roadmap for AD algorithms, guiding the recording of operations, the application of the chain rule, and ultimately, the efficient computation of gradients for training deep learning models. Backpropagation, as a specific type of AD, heavily relies on the computational graph for its efficient calculation of gradients with respect to the loss function.

## 2. Practical Implementation (Pytorch)

* Choose a deep learning framework (PyTorch) and implement a simple neural network model from scratch using Autograd capabilities.
* Define a custom loss function and explore how Autograd computes gradients for this function.
* Experiment with different optimization algorithms (e.g., SGD, Adam) and observe their effects on training convergence and performance.

## 3. Experimental Analysis

* Conduct experiments to analyze the efficiency and numerical stability of Autograd-based gradient computations.
* Investigate the impact of batch size, learning rate, and network architecture on gradient computation and training dynamics.

# QUESTION 2

Implement CNN-LSTM models for the problem of Image Captioning with at least two options: with and without using attention mechanisms for the decoding process. Training the models on Colab environments using GPU and CPU only.

References:

<https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html?highlight=parameter>

<https://arxiv.org/pdf/1502.05767.pdf>

<https://www.mathworks.com/help/deeplearning/ug/deep-learning-with-automatic-differentiation-in-matlab.html>

<https://jingnanshi.com/blog/autodiff.html?fbclid=IwAR2Z4XXV6LvJ5-IYXIGq8TiDNLqB8wB8Y6ByBrrhf2qjnvG0Ia28O4oFEPs>

<https://stats.stackexchange.com/questions/224140/step-by-step-example-of-reverse-mode-automatic-differentiation?fbclid=IwAR2vqNRGk_pGdo11Tz4fASVbni5S_QMB1VS9VVGZbdm5YItr1CpAxXZSuXI>