# TABLE OF CONTENTS

[ACKNOWLEDGEMENT 1](#_Toc162904714)

[TABLE OF CONTENTS 1](#_Toc162904715)

[LIST OF TABLES, DRAWINGS, GRAPHICS 2](#_Toc162904716)

[CHAPTER 1: Understanding and Implementing automatic differentiation (Autograd) in Deep Learning 3](#_Toc162904717)

[1.1 What is Automatic differentiation (Autograd) 3](#_Toc162904718)

[1.2 How to compute the gradients 3](#_Toc162904719)

[1.3 Automatic differentiation mechanism 5](#_Toc162904720)

[1.4 Why Automatic differentiation significance in training deep learning models 5](#_Toc162904721)

[1.5 Compare and contrast forward mode and reverse mode automatic differentiation 6](#_Toc162904722)

[1.6 Forward mode and reverse mode automatic differentiation advantages and disadvantages. 7](#_Toc162904723)

[CHAPTER 2: BLEU 10](#_Toc162904724)

# LIST OF TABLES, DRAWINGS, GRAPHICS

[Figure 1: d is output of function f(x,y) = x + y 4](#_Toc162901662)

[Figure 2: Local Gradients 4](#_Toc162901663)

[Figure 3: Backpropagation in a Computational Graph 5](#_Toc162901664)

# CHAPTER 1: Understanding and Implementing automatic differentiation (Autograd) in Deep Learning

## What is Automatic differentiation (Autograd)

Automatic differentiation (Autograd) stands as a pivotal technique within machine learning, particularly for the training of deep learning models. It operates by automatically computing the derivatives (gradients) of mathematical expressions or functions concerning their inputs. This process of differentiation is fundamental for optimizing intricate models through gradient-based optimization algorithms like stochastic gradient descent (SGD), Adam, and others.

## How to compute the gradients

Each node of the computation graph, with the exception of leaf nodes, can be considered as a function which takes some inputs and produces an output. Consider the node of the graph which produces variable d from w4 c and w3 b

. Therefore we can write,

*D* = *f*(*w3* ​*b*, *w4* ​*c*)

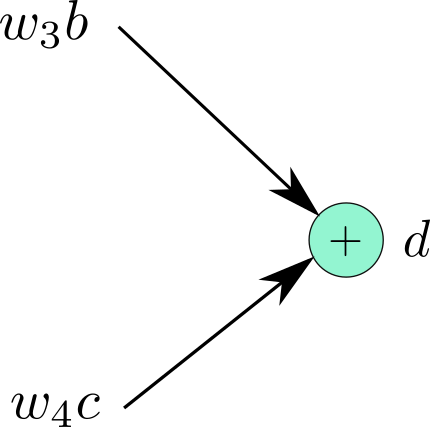


Figure 1: d is output of function f(x,y) = x + y

Now, we can easily compute the gradient of the f with respect to it's inputs,  and  (which are both 1). Now, **label the edges coming into the nodes** with their respective gradients like the following image.

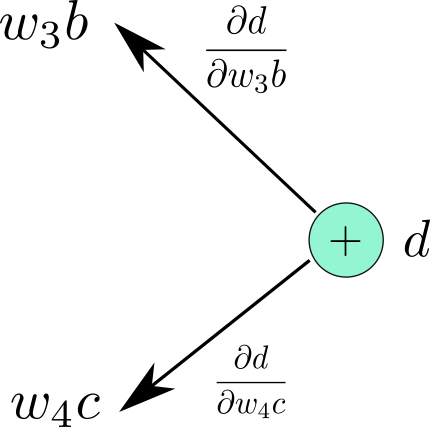


Figure 2: Local Gradients

We do it for the entire graph. The graph looks like this.

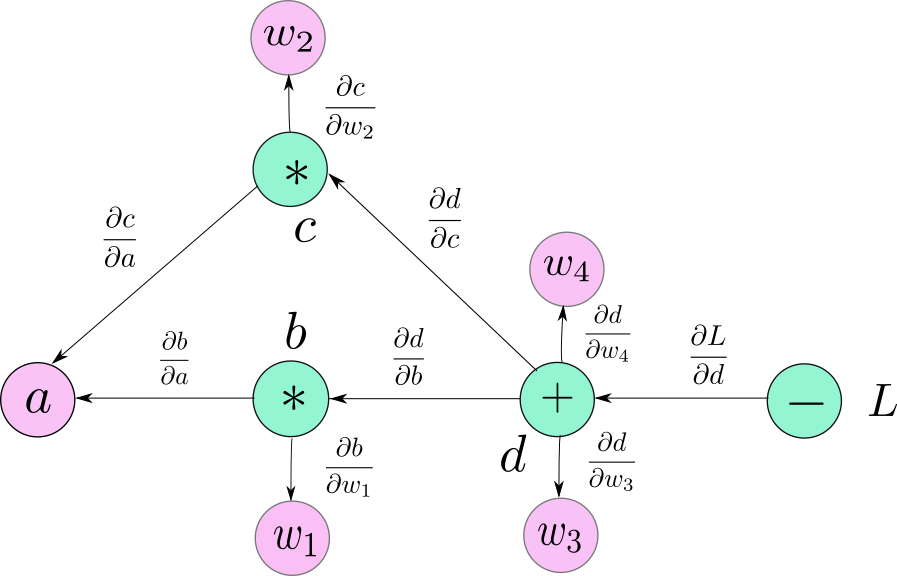


Figure 3: Backpropagation in a Computational Graph

Following we describe the algorithm for computing derivative of any node in this graph with respect to the loss,

L . Let's say we want to compute the derivative,

* 1. We first trace down all possible paths from d to w4.
  2. There is only one such path.
  3. We multiply all the edges along this path.

If you see, the product is precisely the same expression we derived using chain rule. If there is more than one path to a variable from L then, we multiply the edges along each path and then add them together. For example, is computed as

## Automatic differentiation mechanism

**Divide into allowed bases**: AD splits each complex function into a series of allowed bases such as addition, subtraction, multiplication, division and basic functions such as exponential, logarithm, trigonometry. Each of these fundamental properties has a previously known derivative.

**Forward and reverse modes**: AD performs analysis using two main modes: forward (forward) mode and backward (reverse) mode. In progress mode, derivatives are calculated as the function is evaluated in the start-to-finish direction. In backward mode, the derivatives are calculated from last to first, making it more efficient in calculating the moral function of multidimensional functions with many variables.

**Accuracy and efficiency**: AD calculates derivatives with numerical precision and efficiency, helping to optimize deep learning models quickly and accurately.

## Why Automatic differentiation significance in training deep learning models

**Efficient Gradient Calculation**: Deep learning models often involve millions of parameters, and optimizing these parameters requires calculating the gradient of the loss function for each parameter. Autograd automates this process, efficiently calculating gradients using techniques such as automatic differentiation in inversion mode. This automation significantly reduces the computational burden and complexity associated with manually creating and deploying gradients.

**Backpropagation implementation**: Backpropagation is the main algorithm used to train deep neural networks. It involves calculating the gradient of the loss function with respect to the model parameters and updating these parameters in a direction that minimizes the loss. Autograd integrates seamlessly with backpropagation algorithms, allowing efficient gradient computation during both forward and backward propagation across network layers.

**Flexibility in Model Architecture**: Autograd allows flexibility in designing complex neural network architectures. Can test many different architectures, activation functions, loss functions, and optimization techniques without having to manually derive gradients for each specific configuration. This flexibility encourages innovation and the discovery of new model architectures tailored to specific tasks or datasets.

**Dynamic computational graph support**: Some deep learning frameworks, such as PyTorch, support dynamic computational graphs where the graph structure can change at runtime. Autograd is well suited for processing dynamic graphs, allowing the construction of models with variable input sizes, conditional execution paths, and dynamic architectures such as recurrent neural networks (RNN) and recursive neural network (RNN).

**Ease of development** : Autograd simplifies development by automatically calculating gradients, reducing the amount of boilerplate code needed to implement gradient-based optimization algorithms. This allows researchers and developers to focus more on high-level design, testing, and hyperparameter tuning, rather than getting bogged down in low-level implementation details.

## Compare and contrast forward mode and reverse mode automatic differentiation

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Forward Mode AD** | **Reverse Mode AD** |
| **Computational Efficiency** | Efficient for functions with few inputs and many outputs | Efficient for functions with many inputs and few outputs |
| **Main Operation** | Propagates derivatives from inputs to outputs | Propagates derivatives from outputs to inputs |
| **Memory Usage** | Low memory usage | High memory usage |
| **Complexity** | Typically simpler to implement | Typically more complex to implement |
| **Use Cases** | Useful for functions with a large number of inputs | Useful for functions with a large number of outputs |
| **Accumulation of Derivatives** | Derivatives are accumulated as the computation progresses | Derivatives are accumulated backwards from outputs to inputs |
| **Application Examples** | Neural networks, where input dimensions are typically large | Gradient-based optimization algorithms such as gradient descent |

## Forward mode and reverse mode automatic differentiation advantages and disadvantages.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Forward Mode AD** | **Reverse Mode AD** |
| Advantages | Suitable for functions with a small number of inputs | Efficient for functions with many inputs and few outputs |
|  | Efficient for calculating derivatives with respect to each input variable separately | Memory-efficient for functions with many outputs |
|  | Simple implementation | Well-suited for gradient-based optimization algorithms |
|  |  | Enables efficient computation of gradients for optimization algorithms such as backpropagation in neural networks |
| Disadvantages | Inefficient for functions with many inputs | Higher memory usage |
|  | Cumbersome for functions with a large number of outputs | More complex to implement |
|  | Accumulates numerical errors when computing higher-order derivatives | May require additional memory for storing gradients during backpropagation |
|  | Requires storage of intermediate values for each input |  |

1. **Explain the role of computational graphs in automatic differentiation and backpropagation.**
2. **Representation of Computation:** Computational graphs provide a specific and structured way to represent the computations performed during the forward pass of a neural network. Each node in the graph represents an operation or a variable, and the edges represent the flow of data or dependencies between them. This representation helps in visualizing and understanding the sequence of operations involved in computing the network's output.
3. **Efficient Gradient Computation:** During backpropagation, gradients are computed by traversing the computational graph in reverse order. The graph structure allows for an efficient propagation of gradients backward through the network by applying the chain rule of calculus at each operation node. This process enables the computation of gradients with respect to each parameter in the network, which is crucial for updating the parameters during training via optimization algorithms like gradient descent**.**
4. **Integration of Partial Derivatives:** In the backward pass, gradients of the loss function with respect to the output of the network are computed first. Then, these gradients are recursively propagated backward through the graph to compute the gradients with respect to the intermediate variables and parameters. The computational graph facilitates the integration of partial derivatives at each node, allowing for the systematic computation of gradients across the entire network.
5. **Memory Efficiency:** Computational graphs enable memory-efficient computation of gradients by storing only the necessary intermediate values required for gradient computation. By retaining only the information needed for the backward pass, computational graphs help manage memory consumption during training, particularly in deep neural networks with many layers and parameters.
6. **Optimization Support:** The structured representation provided by computational graphs enables the application of various optimization techniques during training. Techniques like graph pruning or optimization of computational paths can be applied to streamline computation and reduce the computational cost of backpropagation.

CHAPTER 2: BLEU

**What is Bleu score ?**

The BLEU (Bilingual Evaluation Understudy) score is a metric used to evaluate the quality of machine-generated translations against one or more reference translations. It is widely employed in the field of natural language processing and machine translation. The BLEU score quantifies the similarity between the machine-generated text and the human-generated reference texts by measuring the precision of n-grams (contiguous sequences of n items, usually words) in the machine translation compared to reference translations. The BLEU score ranges from 0 to 1, with higher scores indicating better-quality translations. It is based on the premise that the more similar the predicted sentence is to the human target sentence, the better.

**N-gram**

An n-gram is a contiguous sequence of n items (usually words) in a sentence

1-gram (unigram): "The", "ball", "is", "blue"

2-gram (bigram): “Ball”, “blue ball”, “blue ball”

3 grams (trigram): “Ball”, “blue ball”

4 grams: "Blue Ball"

To calculate the probability of a sentence appearing P( w1 ​,w2 ,…,wn), we will separate the probabilities into products and use the chain rule of probability

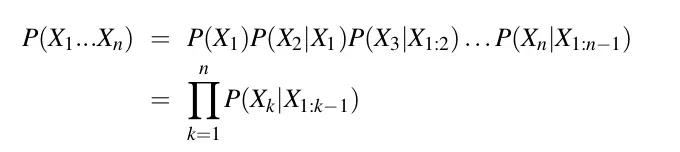


Figure 4: Chain rule of probability

**Precision**

In the context of BLEU, precision measures the proportion of n-grams in the machine translation that also appears in the reference translation(s). It is calculated as the number of overlapping n-grams (predicted sentence) divided by the total number of n-grams in the machine translation (human target sentence)

Example:

Target Sentence: I learn deep learning.

Predicted Sentence: I learned deep learning.

***Precision = Number of correct predicted words / Number of total predicted words***

*Precision = 3 / 4*

*But using Precision is not good enough. There are 2 more cases we need to handle*

**Repetition:**

Repetition refers to the occurrence of the same n-grams multiple times in the machine translation. High repetition can negatively impact the BLEU score because it might suggest lack of diversity or fluency in the translation.

Target Sentence: I learn deep learning.

Predicted Sentence: I I I I.

Precision = 4 / 4 = 1.

**Clipped Precision:**

Clipped precision adjusts the count of each n-gram in the machine translation to match the count of the same n-gram in the reference translation that has the highest frequency. This adjustment helps provide a more accurate assessment of the translation quality, especially when dealing with repetitive phrases or sentences.

Target Sentence 1: I learn a hard major.

Target Sentence 2: I am learning a difficult major.

Predicted Sentence: I I I learn difficult subject.

Now we will perform the following steps:

Compare each word in the predicted sentence with the entire target sentence. If any word matches the target sentence, the prediction is considered correct.

Limit the number of each word to the maximum number of times that word appears in the target sentence. This helps avoid Repetition. We have the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| Word | Matching sentences | Matched Predicted Count | Clipped Count |
| I | Both | 3 | 1 |
| Learn | Target 1 | 1 | 1 |
| Difficult | Target 2 | 1 | 1 |
| Subject | None | 0 | 0 |
| Total |  | 5 | 3 |

***Clipped Precision = Clipped number of correct predicted words / Number of total predicted words***

Clipped Precision = 3 / 6

**Calculating BLEU Score:**

* Calculate the precision for each n-gram up to a certain order (usually up to 4-grams).
* Calculate the geometric mean of these precisions. This is the modified precision.

To compute the geometric mean precision score, we multiply these precision scores together and then take the Nth root, where N is the number of precision scores.

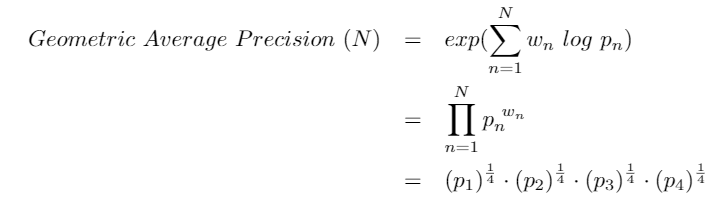
* Calculate the brevity penalty to account for shorter translations.

Brevity Penalty aims to prevent artificially high BLEU scores that could result from generating shorter, incomplete translations.

This penalty factor is then multiplied by the precision scores in the BLEU score calculation to adjust for the brevity of the translation.

**Geometric Average Precision Scores**

Next, we combine the Precision Scores using the formula below. This can be calculated for different N and different weight values. Specifically, we let N = 4 and uniform weights wn = N/4.

****