https://science.utm.my/procscimath
Volume 24 (2024) 81-89

Introduction to Automatic Differentiation and Neural Differentiation Equation

Mohamad Azrin Syafiq Halim, Yeak Su Hoe
Department of Mathematical Sciences, Faculty of Science,
Universiti Teknologi Malaysia, 81310 Johor Bahru, Johor, Malaysia.

Abstract
This study explores various methods computers used to calculate derivatives, crucial for mathematical and computational modelling, with a focus on their application in machine learning. Derivatives are essential in optimization and gradient-based algorithms, key in developing and refining machine learning models. The research categorizes derivative computation into four main methods: manual derivation and coding, numerical differentiation via finite difference approximations, symbolic differentiation, and automatic differentiation (AD). Traditionally, manual derivation has required individuals to compute and code derivatives themselves, a time-consuming, labour-intensive, and error-prone process, especially as models become more complex. Numerical differentiation offers a simpler approach by approximating derivatives using finite difference methods, but it often sacrifices accuracy and can be numerically unstable. Symbolic differentiation manipulates mathematical expressions symbolically to derive analytical expressions for derivatives, excelling with complex functions but often resulting in cumbersome expressions difficult to interpret or implement algorithmically, especially with functions that have conditional statements or loops. In contrast, AD leverages the chain rule of calculus to propagate derivatives through a computational graph, generating derivative computations alongside function evaluations during code execution, facilitating efficient and accurate computation of derivatives, aligning well with modern programming languages and libraries, and is ideal for gradient-based optimization in machine learning. Although AD is often referred to as "automatic," it computes derivatives numerically during code execution rather than performing symbolic manipulation of expressions, emphasizing its practical implementation in computational frameworks. This research highlights AD's pivotal role in modern machine learning, particularly in neural networks and backward propagation algorithms, underscoring its transformative potential in enhancing the efficiency, accuracy, and scalability of machine learning models, marking the evolution from manual differentiation to automated symbolic and automatic differentiation techniques.

Keywords: Automatic Differentiation; Neural Ordinary Differential Equations; Back Propagation; Neural Network

Introduction

Derivative computation within computer programs plays a fundamental role in various scientific and engineering applications, particularly in the fields of machine learning and numerical simulations. This research explores the intricate methods of derivative computation, categorizing them into four primary approaches: manual derivation and coding, numerical differentiation using finite difference approximations, symbolic differentiation, and automatic differentiation (AD), also known as algorithmic differentiation.

Traditionally, machine learning methods have relied on derivative evaluations for optimization procedures, often requiring manual derivation when introducing new models. However, manual differentiation is both time-consuming and prone to errors. Numerical differentiation, while simpler, often proves to be inaccurate and lacks scalability in machine learning applications. Symbolic differentiation

addresses some of these issues but tends to produce complex expressions that can limit algorithmic control flow.

Automatic differentiation has emerged as a powerful technique, interpreting computer programs by integrating derivative values into the variable domain and redefining operators to propagate derivatives according to the chain rule of differential calculus. Despite its widespread adoption in various domains, the term "automatic" in AD can be misleading. It specifically refers to a family of techniques that compute derivatives by accumulating values during code execution, resulting in numerical derivative evaluations rather than symbolic derivatives.

In the context of sophisticated automobile simulations, traditional methods for manually or numerically computing derivatives are often computationally expensive, error-prone, and imprecise, especially in high-dimensional scenarios. This research aims to enhance the computational efficiency and numerical stability of derivative calculations by implementing automatic differentiation approaches tailored to the specific demands of these simulations.

The objectives of this study are to explore the relevance of automatic differentiation in automotive applications, explain the concepts and approaches related to AD, examine its application in deep learning and optimization, generate Python code for its implementation, and identify its importance and efficacy in deep learning tasks.

This research focuses on dual numbers, forward mode differentiation, and reverse mode differentiation, with implementations carried out using Python programming. By doing so, the study contributes to a better understanding of the practical applications of AD and enriches knowledge on various implementation options in different contexts.

Overall, this research aims to unravel the significance of automated differentiation, highlighting its transformative potential in enhancing the efficiency and scalability of machine-learning models and other computational applications

Automatic Differentiation with Symbolic and Numerical Differentiation

Differentiable programming relies on automatic differentiation, a method that accurately calculates function derivatives through a series of algorithmic transformations. [2] points out that AD stands apart from symbolic and numerical differentiation techniques. Symbolic differentiation involves computing function derivatives using symbolic expressions, which can become unwieldy with complex functions despite the potential for simplification.

Numerical differentiation approximates derivatives using finite differences, which is straightforward to implement but can introduce numerical inaccuracies and require many function evaluations, making it computationally expensive [5].

In contrast, automatic differentiation addresses the limitations of both symbolic and numerical approaches by recursively applying the chain rule. This method is both fast and accurate, capable of computing precise derivatives for any differentiable function, regardless of its complexity [1].

Types of Automatic Differentiation

Automatic differentiation offers three modes: forward, backward, and mixed. In forward mode, the derivative of each variable is computed concerning the input and propagated through the sequence of operations. In reverse mode, the derivative is calculated by finding the derivative of the output with respect to each variable and propagating backward through the operations. Mixed mode allows a combination of these two orientations during computation [3].

There are two primary approaches to implementing AD: operator overloading and source code transformation. Operator overloading involves defining additional mathematical operations on custom types that carry derivative information, allowing the chain rule's recursive application to determine a function's derivative. Although easy to implement and widely used in languages supporting operator overloading, it may raise concerns about code optimization, efficiency, and type testability [3].

Source code transformation translates a function's source code into new code that computes its derivative. This method is more efficient than operator overloading as it avoids the frequent invocation of overloaded operators and provides more precise control over numerical computations. However, it is more challenging than operator overloading, requiring an understanding of the semantics of complex programs and the provision of appropriate code for derivative computation [4].

Automatic differentiation, whether in forward, backward, or mixed mode, can be achieved using both operator overloading and source code transformation methods. The choice between them

depends on specific needs, including the function's complexity, differentiation precision, efficiency, programming language, and available tools [4]. For instance, operator overloading may be suitable for small functions and rapid prototyping, while source code transformation may be preferable for complex functions and production-level code.

Neural Differentiation

Neural differentiation, critical in training neural networks, has evolved significantly. The computation of derivatives has been central to the success of gradient-based optimization methods since the advent of backpropagation algorithms. This literature review explores the historical development and methodologies of neural differentiation, including symbolic, numerical, and notably, automatic differentiation. Emphasizing AD's efficiency and precision, it discusses various modes and implementation strategies, including operator overloading and source code transformation. While acknowledging challenges related to computational cost and numerical stability, the review highlights practical applications of neural differentiation in optimizing neural networks for diverse domains, underscoring its ongoing importance in the ever-evolving landscape of machine learning.

Automatic Differentiation and Neural Ode

Automatic Differentiation (AD) enables precise computation of derivatives within a fixed amount of time, a capability that finds applications in various fields. Differentiation, the measure of rate change, is fundamental to optimization techniques and is integral in methods like backpropagation in deep neural networks and the equations of motion in physics. For instance, consider using Newton's method to compute the inverse of a function g, which involves repeated applications as shown in this equation:

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

Here, x_n represents the variable value at iteration n. The process continues until convergence, leveraging derivatives. In machine learning, training parameters often involve minimizing a function called loss. With potentially billions of parameters, efficient derivative computation methods are crucial.

AD frameworks provide a compact programming language within a larger one, allowing for gradient computation as quickly as function evaluation. This is particularly useful in deep learning, where a network with a defined loss function can obtain gradients efficiently.

Dual Numbers

One AD approach involves dual numbers, as described by [1]. Dual numbers can be stored in any data structure, maintaining their dual properties until arithmetic operations are performed. Differentiation proceeds as operations are applied to dual numbers.

In practical terms, when computing a function g's derivative in a programming language, an AD tool can be utilized to add code handling dual operations, allowing simultaneous computation of the function and its derivative. Methods include using libraries, source code modification, or operator overloading, keeping differentiation transparent to the user. Appendix A provides a code implementation of Dual Numbers, including class definitions and arithmetic and differentiation operations.

Another approach involves Taylor's theorem and dual numbers. Assuming g is sufficiently smooth and ϵ is an infinitesimal parameter, a function can be derived as in equation 3.2 based on Taylor's theorem:

$$f(x+\epsilon) = g(x) + \epsilon g'(x) + \epsilon^2 g''(x) + \cdots \tag{2}$$

Since ϵ is very small, ϵ^2 becomes negligible, simplifying $f(x+\epsilon)$ as follows:

$$g(x+\epsilon) = g(x) + \epsilon g'(x) \tag{3}$$

Assuming an augmented object $x+\epsilon$ in the calculation of f, gradients can be extracted from the coefficient of the ϵ term. The addition and multiplication operators for dual numbers are defined as follows:

$$(x + a\epsilon) + (y + b\epsilon) = (x + y) + (a + b)\epsilon \tag{3}$$

$$(x + a\epsilon)(y + b\epsilon) = xy + (xb + ya)\epsilon + (ab)\epsilon 2 = xy + (xb + ya)\epsilon \tag{4}$$

The division of dual numbers can be approached in two ways:

$$\frac{1}{x+y}=\frac{1}{y}-\frac{x}{y^2}+\frac{x^2}{y^3}-\frac{x^3}{y^4}+\cdots$$

which converges when $|x|<|y|$, and

$$\frac{1}{x+y}=\frac{1}{x}-\frac{y}{x^2}+\frac{y^2}{x^3}-\frac{y^3}{x^4}+\cdots$$

which converges when $|y|<|x|$. The division operators can be simplified as shown in the equation:

$$\begin{aligned}\frac{x+a\epsilon}{y+b\epsilon}&=(x+a\epsilon)\left(\frac{1}{y}-\frac{b}{y^2}\epsilon+\cdots\right)=\frac{x}{y}+\epsilon\left(\frac{a}{y}-\frac{bx}{y^2}\right)-\epsilon^2\frac{ab}{y2}+\cdots\\&=\frac{x}{y}+\epsilon\left(\frac{a}{y}-\frac{bx}{y^2}\right)\end{aligned}\tag{5}$$

Polynomials play a significant role in approximating various functions. For example, the exponential function e^x can be approximated as follows:

$$e^x\approx\sum_{k=0}^{n}\frac{1}{k!}x^k$$

where $e^x=1+x+\frac{1}{2}x^2+\frac{1}{6}x^3+\cdots$. The accuracy of the approximation improves with higher k.
The gradient of a monomial x^k with an integer k is $(x^k)'=kx^{k-1}$. This property is used to derive exponential Taylor expansions as shown in the equation:

$$e^{x+a\epsilon}=e^x+a\epsilon e^x+a^2\epsilon^2e^x+\cdots e^x+a\epsilon e^x\tag{6}$$

Similarly, for trigonometric functions such as sine, equation (7) is derived:

$$\sin(x+a\epsilon)=\sin(x)+a\epsilon\cos(x)-a^{2\epsilon^2\sin}(x)+\cdots=\sin(x)+a\epsilon\cos(x)\tag{7}$$

The derivative of $\sqrt{x}$ is:

$$\left(\sqrt{x}\right)'=\left(x^{\frac{1}{2}}\right)'=\frac{1}{2}x^{\frac{1}{2}-1}=\frac{1}{2\sqrt{x}}$$

For the square root of an augmented object, equation (8) is used:

$$(x+a\epsilon)^{\frac{1}{2}}=\sqrt{x}+\frac{a\epsilon}{2\sqrt{x}}-\frac{a^2\epsilon^2}{4x^{\frac{3}{2}}}+\cdots=\sqrt{x}+\frac{a\epsilon}{2\sqrt{x}}\tag{8}$$

The square root is calculated using Newton's method rather than a polynomial approximation. To find $y,\sqrt{y}=x$ implies $x^2-y=0$, and $f(x)=x^2-y$. Using Newton's method in equation (1), the fixed-point iteration for finding the square root is expressed in equation (9):

$$x_{i+1}=x_i-\frac{f(x_i)}{f'(x_i)}=x_i-\frac{x_i^2-y}{2x_i}=\frac{1}{2}\left(x_i+\frac{y}{x_i}\right),\qquad i=0,1,2,\ldots,n\tag{9}$$

The function iterates until convergence to a fixed point, approximating the square root of y.

Forward Mode Differentiation

In both forward and reverse mode AD algorithms, a function is represented as a computational graph or Wengert list, where each node v corresponds to an intermediate computation. The ultimate derivative is obtained by chaining together intermediate values. [1] illustrate forward mode AD with the function g in equation (10):

$$y=f(x_1,x_2)=x_1x_2-\cos(x_2)\tag{10}$$

The derivative g' is evaluated at $x_1=2\ and\ x_2=3$.
In order to calculate $\frac{\partial y}{\partial x_2}$, a derivative is assigned to each intermediate variable vi:

$$\dot{v}_1=\frac{\partial v_i}{\partial x_2}$$

Forward mode AD tracks intermediary computations at each node. The following calculations are:

$$\begin{gathered}v_1=x_1=2,\\v_2=x_2=3,\\v_3=v_1\cdot v_2=2\cdot3=6,\\v_4=\cos(v_2)=\cos(3)\\v_5=v_3-v_4=6-\cos(3)\approx5.00\end{gathered}$$

Differentiating each equation v with respect to x_2:

$$\dot{v}_1 = \frac{\partial x_1}{\partial x_2} = 0,$$
$$\dot{v}_2 = \frac{\partial x_2}{\partial x_2} = 1,$$
$$\dot{v}_3 = \frac{\partial (v_1 * v_2)}{\partial v_2} = \dot{v}_1(v_2) + \dot{v}_2(v_1) = 0 + 2 * 1 = 2,$$
$$\dot{v}_4 = \frac{\partial \cos(v_2)}{\partial v_2} = -sin(v_2) = -\sin(3),$$
$$\dot{v}_5 = \frac{\partial (v_3 - v_4)}{\partial v_2} = \dot{v}_3 - \dot{v}_4 = 2 + \sin(3) \approx 2.05,$$

This demonstrates forward mode AD's tracking of intermediary computations. However, it becomes impractical for high-dimensional input data in deep learning.

Forward mode AD handles vector-valued functions $g: \mathbb{R}^n \rightarrow \mathbb{R}^m$, using the Jacobian matrix:

$$J_f = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Forward mode scales well with the number of outputs mmm, suitable for generative applications. It also provides a matrix-free approach for Jacobian-vector products (JVPs):

$$J_f \cdot v = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$$

By initializing $\dot{x} = v$, JVPs can be computed in a single pass. For $g: \mathbb{R}^n \rightarrow \mathbb{R}$, the gradient:

$$\nabla g = \left[\frac{\partial y}{\partial x_1}, \cdots, \frac{\partial y}{\partial x_n}\right]$$

requires n evaluations in forward mode.

Reverse Mode Differentiation

Reverse mode differentiation, or backpropagation, is preferred when the number of inputs n is much larger than the number of outputs mmm, common in deep learning tasks.

For function $y = f(x_1, x_2) = x_1 x_2 - \sin(x_2)$

$$v_1 = x = 2,$$
$$v_2 = y = 3,$$
$$v_3 = v_1 \cdot v_2 = 2 \cdot 3 = 6,$$
$$v_4 = sin\, v_2 = sin\, 3,$$
$$v_5 = v_3 - v_4 = 6 - sin3 \approx 5.95$$

In reverse mode, each intermediate variable v_i is accompanied by an adjoint $\bar{v}_i$:

$$\bar{v}_5 = \bar{y} = 1,$$
$$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4} = -1,$$
$$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3} = 1 * \frac{\partial v_3 - v_4}{\partial v_3} = 1,$$
$$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2} + \bar{v}_3 \frac{\partial v_3}{\partial v_2} = -1 * \partial \sin \frac{v_2}{\partial v_2} + 1 * \frac{\partial v_1 v_2}{v_2} = 2 - cos3 \approx 1,$$
$$\bar{v}_1 = \bar{v}_3 \frac{\partial v_3}{\partial v_1} = 1 * \frac{\partial v_1 v_2}{v_1} = 1 * v_2 = 3$$

Both reverse mode and forward mode AD yield the same result, albeit through different computational paths. However, reverse mode AD computes all partial derivatives in a single pass, making it more efficient, especially for functions with numerous inputs. For instance, for a function $g: \mathbb{R}^n \rightarrow \mathbb{R}$,, reverse

mode AD only requires one pass to compute the entire gradient $\nabla g = \left[\frac{\partial y}{\partial x_1}, \ldots, \frac{\partial y}{\partial x_n}\right]$, whereas forward mode AD would necessitate n passes.
Similar to forward mode AD, reverse mode can compute transposed Jacobian-vector products (JVPs) efficiently. By initializing the reverse pass with $\bar{y} = v$, transposed JVPs can be evaluated as shown:

$$J_g^T \cdot u = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$$

Neural Ordinary Differential Equations

Neural Ordinary Differential Equations: Discusses the innovative use of Neural ODEs in modeling dynamic systems within neural networks. The evolution of a state $h(t)$ is governed by:

$$\frac{dh(t)}{dt} = f(h(t), t, \theta)$$

where θ represents the parameters of the neural network. Neural ODEs provide a framework for continuous-time models, allowing for flexible modeling of dynamical systems and leveraging the power of AD for training

Physics in Neural Network

Artificial neural networks have found diverse applications in fields like computer vision and natural language processing. Physics-informed neural networks (PINNs) represent a recent innovation, extending neural network usage to solving partial differential equations (PDEs). This approach leverages the physical properties of PDEs to guide training, making them applicable to various engineering and scientific domains. This study aims to introduce PINNs by solving a first-order ordinary differential equation using PyTorch.

PINNs leverage two fundamental properties of neural networks: their ability to approximate any function and automatic differentiation for efficient computation of derivatives. The core idea is to construct a loss function incorporating the PDE residual and boundary conditions, thereby training the network to approximate the solution effectively.
The loss function is formulated as follows:

$$\mathcal{L} = \mathcal{L}_{DE} + \mathcal{L}_{BC}$$

Where:

$$\mathcal{L}_{DE} = \frac{1}{M}\sum_{j=1}^{M}\left(\frac{df_{NN}}{dt}|_{t_j} - Rf_{NN}(t_j)\left(1 - f_{NN}(t_j)\right)\right)^2$$

$$\mathcal{L}_{BC} = (f_{NN}(t_0) - 0.5)^2 \; with \; t_0 = 0$$

$$\mathcal{L} = \mathcal{L}_{DE} + \mathcal{L}_{BC} = \frac{1}{M}\sum_{j=1}^{M}\left(\frac{df_{NN}}{dt}|_{t_j} - Rf_{NN}(t_j)\left(1 - f_{NN}(t_j)\right)\right)^2 + (f_{NN}(t_0) - 0.5)^2$$

The gradient is given as:

$$\frac{\partial \mathcal{L}}{\partial f_{NN}(t_k)} = \frac{\partial \mathcal{L}}{\partial f_{NN,k}}$$
$$= \frac{1}{M}\sum_{j=1}^{M} 2\left(\frac{df_{NN}}{dt}|_{t_j} - Rf_{NN}(t_j)\left(1 - f_{NN}(t_j)\right)\right)\cdot\left(\frac{\partial}{\partial f_{NN,k}}\frac{df_{NN,k}}{dt} - R \; + 2Rf_{NN,k}\right)$$
$$+ 2(f_{NN}(t_0) - 0.5)\delta_{0,k}$$

Where the Kronecker delta function is given

$$\delta_{j,k} = \begin{cases} 0, & j \neq k \\ 1, & j = k \end{cases}$$

Gradient descent, facilitated by automatic differentiation, minimizes this loss function, effectively training the network to approximate the solution while satisfying the PDE and boundary conditions.

Results of Neural Network

The logistic differential equation, a well-known first-order ordinary differential equation used to model population growth:

$$\frac{df}{dt} = Rf(t)\big(1 - f(t)\big)$$

Here, the function $f(t)$ represents the population growth rate over time t, and the parameter R determines the maximum population growth rate, significantly affecting the shape of the solution. To fully specify the solution of this equation, an initial condition must be imposed, for example, at $t = 0$ such as:

$$f(t = 0) = 0.5$$

The General Equation is:

$$f = \frac{e^{Rt}}{e^{Rt} + 1} = \frac{e^{Rt} \cdot e^{-Rt}}{(e^{Rt} + 1)e^{-Rt}} = \frac{1}{1 + e^{-Rt}}$$

Utilizing Python code, results demonstrate efficient convergence in replicating the analytical solution for a simple differential equation. The Adam optimizer with a learning rate of 0.1 and 100 epochs suffices to achieve near-perfect replication of the analytical result for a specified maximum growth rate, $R = 0.50$.

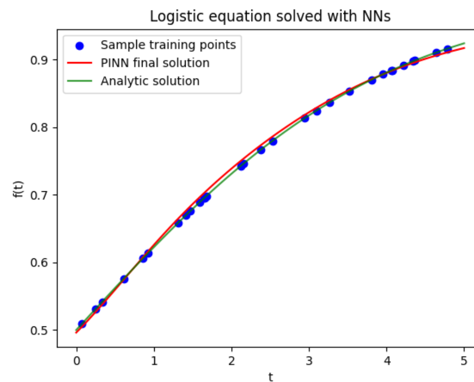

Figure 2 Exact solution with an initial value $f(t = 0) = 0.5$ and at $R = 0.5$

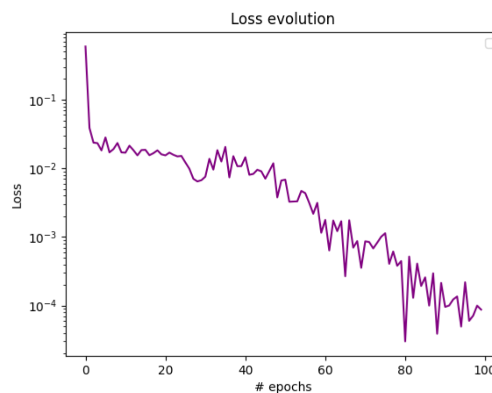

Figure 2 The Loss Graph Over 100 epochs

While this study successfully solves a simple one-dimensional problem, achieving convergence for more complex equations poses challenges. Techniques like domain decomposition and smart weighting of loss contributions are crucial for tackling time-dependent and multidimensional problems effectively.

Conclusion

The exploration of derivative computation within computer programs reveals significant advancements over traditional methods. The study categorizes derivative computation into four primary methods:

manual derivation and coding, numerical differentiation using finite difference approximations, symbolic differentiation, and automatic differentiation (AD).

Manual differentiation, though historically crucial, is labour-intensive and prone to error, especially as model complexity increases. Numerical differentiation offers simplicity but at the cost of accuracy and stability. Symbolic differentiation, while precise, often produces overly complex expressions that can be difficult to interpret and implement.

Automatic differentiation, leveraging the chain rule of calculus, stands out for its efficiency and accuracy. It integrates seamlessly with modern programming languages and libraries, making it particularly suitable for gradient-based optimization in machine learning. AD's ability to compute derivatives numerically during code execution has made it an essential tool in the development of scalable and precise machine-learning models.

The implementation of AD in neural networks, particularly through algorithms like backpropagation, has demonstrated significant potential in improving the efficiency, accuracy, and scalability of machine-learning models. The transition from manual and symbolic differentiation to AD marks a pivotal evolution in computational methods, reflecting a broader trend towards automation and increased computational efficiency in the field of machine learning

Acknowledgement

I am deeply grateful to Allah SWT for His Grace, Blessings, Love, Guidance, and Power, which have allowed me to complete my final year project. Alhamdulillah for the simplicity in this accomplishment. I extend my heartfelt gratitude to my supervisor, Assoc. Prof. Dr. Yeak Su Hoe, whose unwavering patience and invaluable guidance have been indispensable throughout this endeavour, shaping this report significantly. To my beloved parents and family, your unwavering love, encouragement, and support have been my pillars throughout my college journey, and your belief in me has been my greatest motivation. To my friends, thank you for the countless moments of laughter and shared sorrows; your friendship has made this journey memorable and meaningful, and may we continue to support each other in our future endeavours.

References

[1] 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, no. 153 (2018): 1-43.

[2] Charles, C. (2019). "Automatic differentiation: Techniques and applications." Retrieved from [source].

[3] Griewank, A., & Walther, A. (2008). "Evaluating derivatives: principles and techniques of algorithmic differentiation." SIAM.

[4] Revels, J., Lubin, M., & Papamarkou, T. (2018). "Forward-mode automatic differentiation in Julia." arXiv preprint arXiv:1607.07892.

[5] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. "Numerical recipes 3rd edition: The art of scientific computing." Cambridge University Press, 2007.