AMSC Project Report

Forward and Reverse-mode Automatic Differentiation with Applications

Giuseppe Galardi, Pietro Beghetto, Zhaohui Song

June 21, 2025

Contents

1 Introduction

The computation of derivatives is a crucial aspect of many modern scientific applications. For example, most techniques employed in the context of Machine Learning/Deep learning require derivatives in order to train a specific model. Automatic differentiation is a computational technique for efficiently and accurately computing derivatives of functions defined by computer programs. Unlike symbolic differentiation or finite difference methods, automatic differentiation takes advantage of the fact that any computer program, regardless of complexity, executes a sequence of elementary arithmetic operations and elementary functions for which exact derivatives are known.

2 Numerical Methodology

2.1 Introduction

The fundamental principle underlying automatic differentiation is the chain rule. For a composite function $f = f_n \circ f_{n-1} \circ \ldots \circ f_1$, the derivative is computed as:

$$\frac{df}{dx} = \frac{df_n}{df_{n-1}} \cdot \frac{df_{n-1}}{df_{n-2}} \cdot \ldots \cdot \frac{df_1}{dx} \tag{1}$$

This decomposition allows an algorithm to compute exact derivatives (up to machine precision) applying the chain rule to the elementary operations that constitute the given function. To better understand how automatic differentiation works, it is useful to view a composite function as a computational graph. Each node in the graph represents an intermediate calculation and the directed edges represent the dependencies of these calculations. Now, by applying the chain rule to this computational graph we can compute the partial derivatives of our function (the reverse-mode paragraph will explain this in better details).

Depending on how the chain rule is evaluated, we have different possible implementation of automatic differentiation, among these, Forward Mode and Reverse Mode are the most widely used. **Forward mode** evaluates the single terms in the chain rule from right to left, the same method that we usually employ when we compute derivatives by hand. **Reverse Mode**, on the other end, evaluates the terms from left to right.

The two methods have different advantages and disadvantages that will be explained in the following paragraphs.

2.2 Forward-Mode Automatic Differentiation

A possible way to implement Forward-Mode automatic differentiation is by using dual numbers. A dual number is an object of the form $a + b\epsilon$, where $a \in \mathbb{R}$ is called the real part, $b \in \mathbb{R}$ is called the dual part, and ϵ is an object satisfying $\epsilon^2 = 0$.

These numbers follow the following arithmetic rules:

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon \tag{2}$$

$$(a + b\epsilon) - (c + d\epsilon) = (a - c) + (b - d)\epsilon \tag{3}$$

$$(a + b\epsilon) \cdot (c + d\epsilon) = ac + (ad + bc)\epsilon \tag{4}$$

$$(a + b\epsilon)/(c + d\epsilon) = a/c + ((bc - ad)/c^2)\epsilon \tag{5}$$

Now, if we write a smooth function $f : \mathbb{R} \to \mathbb{R}$ via its Taylor expansion and we evaluate it in $x = a + b\epsilon$ we get:

$$f(a + b\epsilon) = f(a) + bf'(a)\epsilon \tag{6}$$

That is, by evaluating a function on a dual number $a+b\epsilon$ we get, in turn, a dual number whose real part is the function evaluated in a and whose dual part is the derivative of the function evaluated in a, times b.

For functions $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$ the same principle applies, but now we have:

$$\mathbf{f}(\mathbf{a} + \mathbf{b}\epsilon) = \mathbf{f}(\mathbf{a}) + (\mathbf{J}_\mathbf{f}(\mathbf{a}) \cdot \mathbf{b})\epsilon \tag{7}$$

That is, we get both the function and the jacobian-vector product (i.e. the directional derivatives along $\mathbf{b}$ of each output function) evaluated in $\mathbf{a}$.

Starting from the previous equations, we can formulate an algorithm that initializes each input variable of a generic composite function as a dual number. The real part of each dual number is set to the value of the corresponding input variable, while the dual part is set to the component of the direction along which we wish to compute the directional derivative of the output functions. If we are interested in computing a single partial derivative, the dual parts are set to zero for all input variables except the one of interest, for which the dual part is set to one. The algorithm then propagates these dual numbers through the sequence of elementary functions that make up the composite function, ultimately yielding the final result along with its directional derivatives.

Due to the fact that to obtain the gradient we should modify each time the vector $\mathbf{b}$, forward mode is preferred, at least theoretically, in those cases where the number of outputs is greater than the number of inputs ($m >> n$). From a practical point of view, forward mode could perform better than reverse mode even with many input variables compared to the number of outputs. This is due to the fact that in reverse mode we need to store all the operations done during the forward pass, adding a non-negligible overhead to the computational time.

2.3 Reverse-Mode Automatic Differentiation

Reverse-mode automatic differentiation, unlike forward mode, which computes derivatives from inputs to outputs, starts from the output and propagates gradients backward to the inputs.

2.3.1 Governing Equations

The fundamental principle of reverse-mode AD is based on the application of the chain rule on a computational graph. For a composite function $y = f(x) = f_n \circ f_{n-1} \circ \ldots \circ f_1(x)$, we define intermediate variables v_i such that:

$$v_0 = x \text{ (input variable)} \tag{8}$$
$$v_i = f_i(v_{i-1}) \text{ for } i = 1, \ldots, n \tag{9}$$
$$y = v_n \text{ (output variable)} \tag{10}$$

Now, define the adjoint of v_i as $\bar{v}_i = \frac{\partial y}{\partial v_i}$, that is, the adjoint of v_i is the partial derivative of the output variable y with respect to the variable v_i. The adjoint variables are computed as:

$$\bar{v}_n = \frac{\partial y}{\partial v_n} = 1 \text{ (seed value)} \tag{11}$$
$$\bar{v}_{i-1} = \bar{v}_i \cdot \frac{\partial v_i}{\partial v_{i-1}} \text{ for } i = n, \ldots, 1 \tag{12}$$

Note that the derivatives $\frac{\partial v_i}{\partial v_{i-1}}$ are simple to compute (for example, if $v_i = \log v_{i-1}$ then $\frac{\partial v_i}{\partial v_{i-1}} = \frac{1}{v_{i-1}}$). So, by propagating these adjoints and by computing these simple derivatives we can compute the actual derivative of interest.

For functions $f : \mathbb{R}^n \to \mathbb{R}^m$ the same rule applies but now for each path in the computational graph that goes from a fixed output to a fixed input we need to sum up all the derivatives computed along these paths. From an implementation perspective, the adjoints are computed starting from an output node and by going backward along a topological ordered list of all the nodes of the computational graph. Using a topological ordered list is necessary because we must be sure that all the paths that go from an output node to a specific node have been explored before being able to backpropagate the adjoint of this node to previous nodes in the computational graph.

Unlike forward mode, where we need to run the algorithm for each component of the gradient, here with a single pass we get all its components. So, theoretically, this algorithm works best for functions with many input variables with respect to the number of outputs ($n >> m$). But, as was introduced before, we need to store all the computations that are performed during the forward evaluation of the computational graph in order to compute the adjoints during the backward pass, and this introduces a lot of accesses to main memory that slow down the algorithm.

In our implementation an arena allocator is employed in order to reduce the total number of calls to *new*/*malloc* and to improve the number of cache hits during the backward pass.

3 Code Structure and Architecture

3.1 Overview

The codebase is organized as a modular library, composed mainly of header files for the core automatic differentiation library. The automatic differentiation modules make extensive use of templates to support different numeric types (double, float, etc.). In particular, this could allow for future extensions such as the implementation of higher-order derivatives for both DualVar(s) and Var(s).

3.2 Main Modules and Components

3.2.1 Core Automatic Differentiation Library

Forward Mode Implementation (`include/autodiff/forward/`)

- `DualVar.hpp`: Core dual number implementation for forward-mode automatic differentiation. Contains all function overloads.
- `ForwardDifferentiator.hpp`: High-level interface for computing derivatives, gradients and jacobians using forward mode
- `CudaSupport.hpp`: CUDA compatibility macros for GPU acceleration
- `ForwardEigenSupport.hpp`: Specializes the NumTraits class to provide Eigen with information on the DualVar class. This allows the use of the DualVar type in Eigen matrices and vectors.

Reverse Mode Implementation (`include/autodiff/reverse/`)

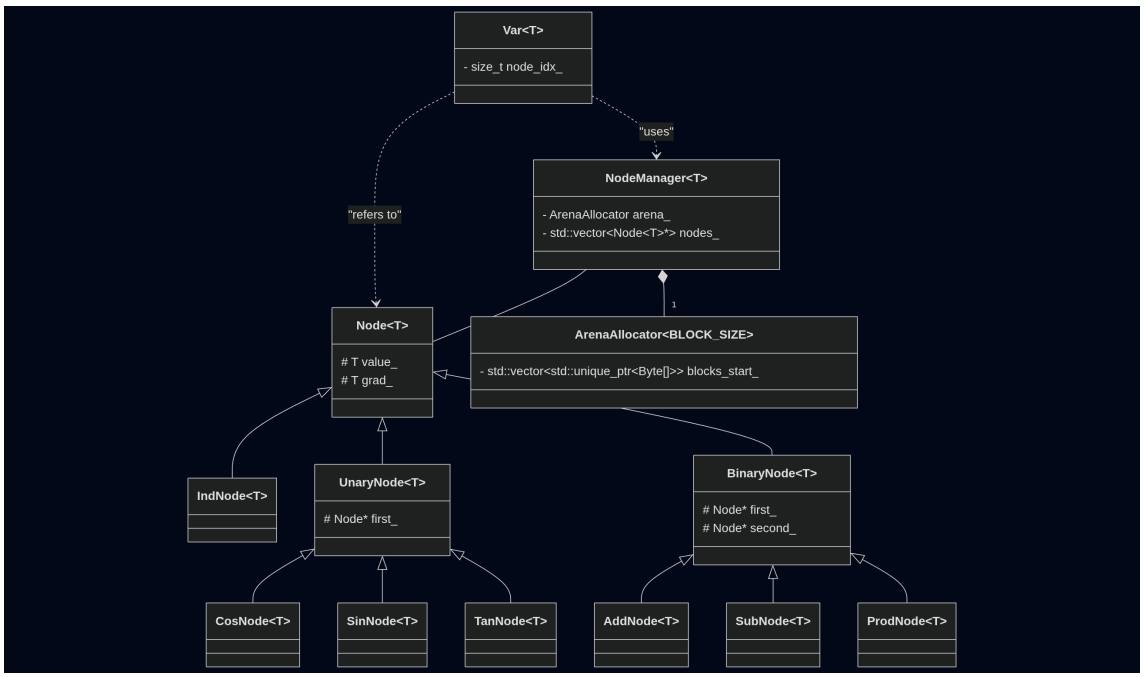

Figure 1: UML class diagram for Reverse Mode (only shows the data members for each class)

- `Var.hpp`: Contains the **Var** class and all its operator overloads. This class is designed to be used by the user as if it were a simple type, such as double or float. Internally, Var relies on the NodeManager class to handle the creation and management of nodes within the computational graph. This design cleanly separates memory management from the operations performed on Var instances. Instead of holding a direct pointer, the Var class stores an index that references its corresponding node in the computational graph. Special care is taken to ensure that a default-constructed Var object points to a dedicated dummy node, preventing invalid references and potential errors.

- `NodeManager.hpp`: Contains the definition of the **NodeManager** helper class, responsible for storing all nodes created during the forward pass in topological order. This class provides methods to create various types of nodes and to access their data members, such as value and grad. Nodes are allocated using an Arena Allocator to minimize the number of dynamic allocations (which can negatively impact performance) and to keep as many nodes as possible in contiguous memory regions, improving cache efficiency. The node creation methods are templated, eliminating the need to write a custom factory function for each new Node subclass and supporting easy extensibility.

- `ArenaAllocator.hpp`: This file contains the definition of the **ArenaAllocator** class which helps managing the memory allocations for all the nodes of the computational graph. This class exposes an *alloc* method that takes care of the actual allocations while respecting the alignment constraints of each type. Even tough some operations performed by this method require the manipulation of raw pointers, care has been taken to use smart pointers (*std::unique_ptr*) where useful. Given that this arena

allocator is used in a context in which many memory allocations must be done repeatedly, a *clear* method is provided. This method does not release any memory that was allocated but it simply resets the arena allocator to an initial state where the previously allocated blocks can be reused (special care has been taken to avoid undefined behaviors while doing so, see comment [1] in the NodeManager.hpp file).

- `Node.hpp`: Contains the declarations of the abstract base class **Node** and of all the classes that derive from it (**IndNode**, **UnaryNode**, **BinaryNode**). The composite pattern is employed to represent the relationship between these types of classes.

- `Functions.hpp`: Contains the definition of the actual classes that derive from UnaryNode and BinaryNode. Each of these classes specializes the backward method of the Node class to allow the propagation of derivative information along the computational graph. Some examples are the classes **CosNode**, **SinNode**, etc.

- `ReverseEigenSupport.hpp`: This file provides a specialization of the NumTraits struct from the Eigen library to supply Eigen with the necessary type information for the Var class. It also customizes specific Eigen functions to ensure compatibility with Var, enabling its use within Eigen matrices and vectors. The main limitation of this approach is that it disables Eigen's automatic vectorization, as supporting vectorized operations for Var would require specializing several internal Eigen classes (an approach that is tightly coupled to Eigen's internal implementation, which may change across versions).

- `ReverseUtility.hpp`: Contains the free functions *gradient* and *jacobian* that simplify the computation of these quantities via reverse-mode automatic differentiation.

3.2.2 Machine Learning Models

(`include/examples/ml/models/`)

- `IModel.h`: The General Interface of a model that can approximate a function or set of data.

- `LinearModel.h`: A linear model that contains 2 parameters, it will approximate any function of the form ax + b.

- `LinearModelParallel.h`: A parallel implementation of the linear model, in which the linear model is updated using an accumulated gradient from a batch of data.

- `NeuralModel.h`: A simple neural model that contains 2 MLPs.

- `NeuralModelOptimized.h`: An optimized neural model that uses span to access the parameters, all parameters are within a contiguous memory, which is more cache-friendly.

- `NeuralModelOpenmp.h`: A parallel implementation of a neural model where the computations are done on different threads, each thread calculates the forward gradients independently(each time over a batch). More specifically, we divide the whole dataset into multiple meta-batches. For example, we have 103 data points, we have 3 threads available, and each time we will work on 3 * 5 data points. With 3 threads working on each mini-batch independently. In this case, a meta-batch contains 3 mini-batches. For each mini-batch, during the loss calculation, a parallel reduction is done on each mini-batch.

3.2.3 Optimization Algorithms

(`include/examples/ml/optimizer/`)

- `Optimizer.h`: A general optimizer interface.
- `SGD.h`: A basic implementation of stochastic gradient descent with a predefined learning rate.
- `SGDWithMomentum.h`: Similar to SGD, but each time we calculate the gradient towards a certain direction, a momentum is accumulated in the gradient updates, and each time going in the same direction, the convergence will be accelerated.
- `Adam.h`: We tried to implement Adam by hand, Adam works with double momentum techniques; one momentum mt is similar to the SGD with momentum, the second momentum is vt, which scales the learning rate for each parameter based on the magnitude of recent gradients. Parameters with consistently large gradients will have a larger vt, leading to smaller effective learning rates, preventing overshooting. Parameters with small or sparse gradients will have a smaller vt, leading to larger effective learning rates, helping them make meaningful progress. Since mt and vt are initialized to zero, they would be biased towards zero, especially during the early training steps. Adam applies a bias correction to these moment estimates to counteract this.

3.2.4 Newton Solver

(`include/examples/newton/`)

- `Newton.hpp`: Implementation of Newton's method for root finding and optimization. Supports both forward and reverse modes for the Jacobian computation.
- `Jacobian.hpp`: Jacobian class used by the Newton solver. The hierarchical strcuture of the code, with a base `Jacobian` class and specialized implementations (`ForwardJacobian`, `ReverseJacobian`, `CudaJacobian`), provides extensibility for future differentiation strategies such as manual Jacobian computation or hybrid approaches.
- `JacobianTraits.hpp`: Type aliases used by the Jacobian/Newton classes.

4 Numerical Experiments and Results

4.1 Forward Mode

4.1.1 Tests `test/{dualvar_test, forward_utility_test}.cpp`

- **Test description:** Unit testing of `DualVar` class operations (arithmetic, transcendental functions, comparison operators) and forward-mode utility functions (derivative, gradient, Jacobian)
- **Input parameters:** Test polynomials $f(x) = 12x^2 + 3x + 4$, multivariate function $f(x, y) = x^2+2xy+y^2+3x+4y+5$, vector-valued function $\mathbf{f}(x, y) = [x^2+y, x+y^2]^T$; Test edge cases with small/large numbers (10^{-15}, 10^{15}).

- **Result verification:** All derivatives were validated against hand-computed analytical solutions.
- **Results:** All unit tests pass with numerical tolerance $\epsilon = 10^{-15}$. DualVar arithmetic operations demonstrate exact derivative computation and transcendental functions maintain machine precision accuracy.

4.1.2 Benchmarks

- **Benchmark description:** Jacobian computation using forward mode automatic differentiation with comparison of CPU vs CUDA implementations for varying problem sizes. Two test runs have been performed, one on functions with fixed input size and varying output, and the other on functions with fixed output size and varying input. Dimensions were increased until the device memory was filled, or the compilation failed.
- **Input parameters:** Function input sizes varied from 2 to 50, with a fixed output dimension of 512. On the second test run, output sizes varied from 2 to 1024, with a fixed input size of 50. Three independent runs were performed for each dimensional configuration.
- **Results:** CUDA implementation demonstrates significant speedup for almost all matrix sizes, with the lowest speedup (7.98×) observed with a 50×2 Jacobian. Memory was the bottleneck for very large matrices, especially for large input dimensions since each thread needs a local copy of the input vector. Forward mode exhibits linear scaling with input variables. Performance graphs show CUDA acceleration factors across different problem scales.

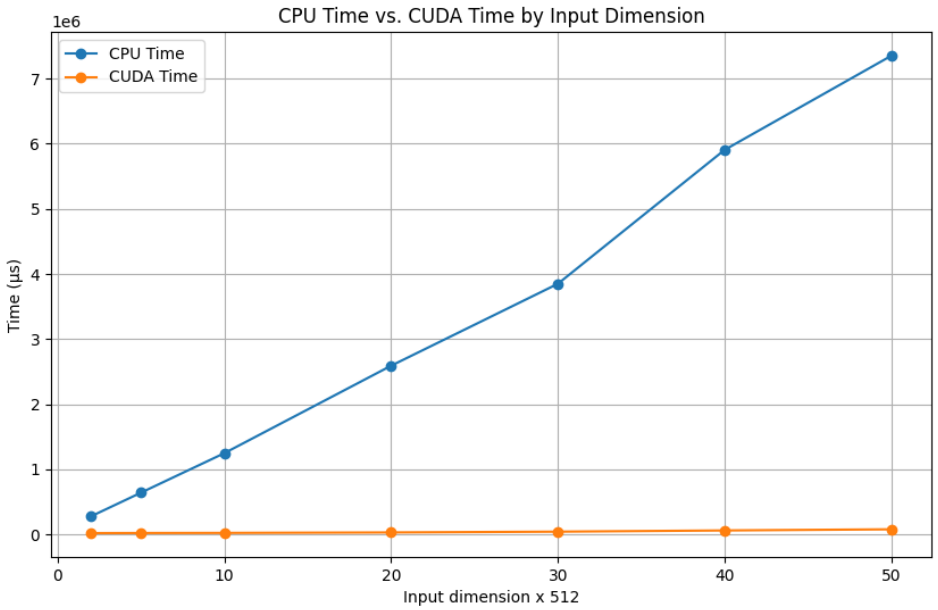

Figure 2: Sample numerical result

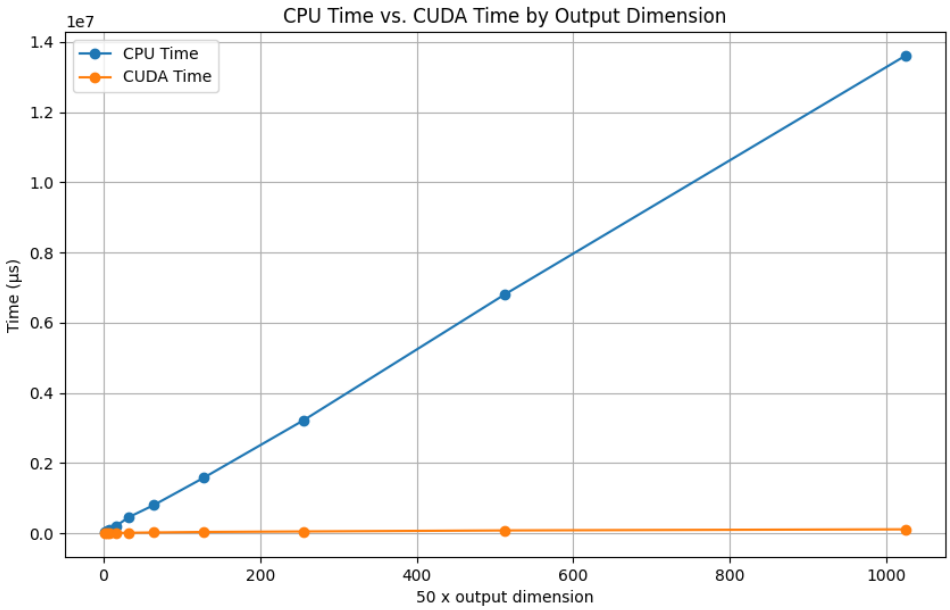

Figure 3: Sample numerical result

4.2 Reverse Mode

4.2.1 Tests `test/{var_test, reverse_utility_test, arena_allocator_test}.cpp`

- **Test description:** Unit testing of the Var class operations (arithmetic, transcendental functions, comparison operators), reverse-mode utility functions (gradient, jacobian) and tests for the arena allocator class.
- **Result verification:** All derivatives were validated against hand-computed analytical solutions.
- **Results:** All units tests on the operators and functions defined on the Var class pass. The tests on the utility functions and the arena allocator pass as well.

5 Autodiff Applications

In this section, we explore different scenarios where the dual variable approach is applied...

We will illustrate how we can use forward-model automatic differentiation to build an example application, such as a neural model for function approximation or a simple Newton solver.

- **Neural models**
  - **Linear Model**: Implementation of 2 parameters to be optimized (w and b).
  - **Parallel Linear Model**: Uses parallel for loops on batches of data.
  - **Neural Model**: A basic 2-layer feed-forward neural network implementation.
  - **Cache-Optimized Neural Model**: Uses std::span to point to a unique, contiguous, flat memory where the parameters are located.
  - **Parallel Neural Model**: Leverages OpenMP threads working on meta-batches, parallelizing mini-batches.
- **Newton Solver** A Newton solver that can utilize both forward mode and reverse mode automatic differentiation to find the roots of non linear functions.

5.1 ML Models

5.1.1 Linear Model and Parallel Linear Model

On 100 datapoints, batch size is 20, SGD learning rate is 0.01, w is 5, b is -2

Test Case	w	b	time in ms
Serial Traning	4.99575	-2.01084	90722
Parallel Training	4.99939	-1.99163	28838

Table 1: Comparison of serial linear and parallel linear results

The speed up is 3.19011

Although there is a speed up, the convergence is diminished, to have the same approximated parameter, the linear model took 200 epochs, whereas the parallel model worked for 1000 epochs, the need of iterations is 5x more..

It seems that in the serial version, each update of the gradient is small, and updating gradients every time keeps the training on track. While updating the gradients in a batch, the effect of "stochastic" gradient descent becomes less stochastic. The convergence is weaker.

5.1.2 Neural model

Training completed in 853 ms
params = [-1.34201, 1.19089, 1.37329, 1.34737, 0.0274887, 1.41561, -2.79584, 2.82683, 2.67177, -1.01097]

Test Case	x	prediction	ground truth
1	-1	-5.99216	-6
2	-0.5	-3.49937	-3.5
3	0	-1.0009	-1
4	0.5	1.49875	1.5
5	1	3.99789	4

Table 2: Comparison of serial linear and parallel linear results

5.1.3 Optimizers training a 2-layers neural model

Optimizer	Training Time (μs)	w	b
SGD	81433	5.00112	-2.01021
Adam	78419	4.99872	-2.00963
SGDWithMomentum	74071	5.00293	-2.01459

Table 3: Training performance and parameters of different optimizers

Generally, the adam optimizer works better.

5.1.4 Activation Function

We observed that to approximate a sinusoidal function, tanh activation function is much better than the RELU activation function.

5.1.5 Linear model and Neural Model

This is a comparison of how different models approximate the same function, between a simple linear function and a multi-parameter neural model with non-linear activation functions.
Adam lr = 0.03
epochs = 1000
batch size = 16
Hidden size of the neural model is 8. There are a total of 3*8 + 1 parameters.

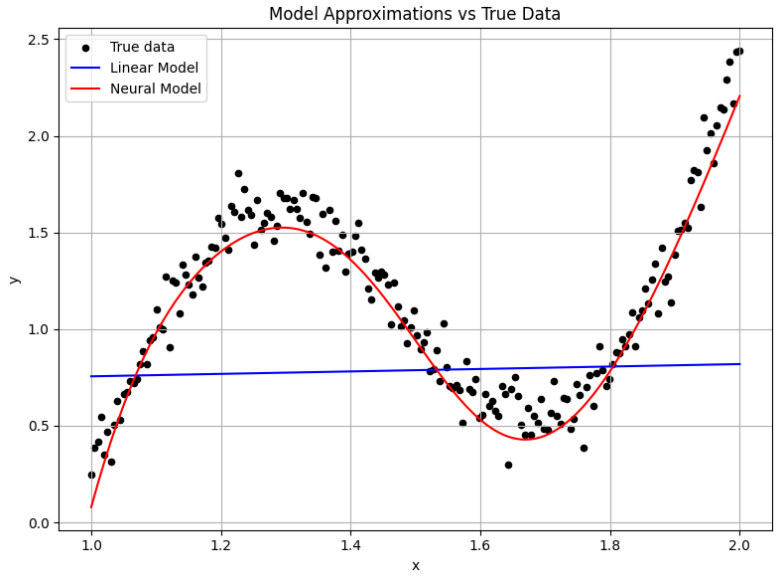

Figure 4: linear function and a neural function

5.1.6 Neural network model with different hidden sizes

There is an illustration of how different sizes of parameters affect the approximation. The total parameters of these 2 models with 2 layers are $3 * hidden\_size + 1$.
int N = 200; // Number of data points
adam lr = 0.03
std::vector $< int >$ hidden layer sizes = $1, 2, 4, 8, 16$;
epochs = 1000

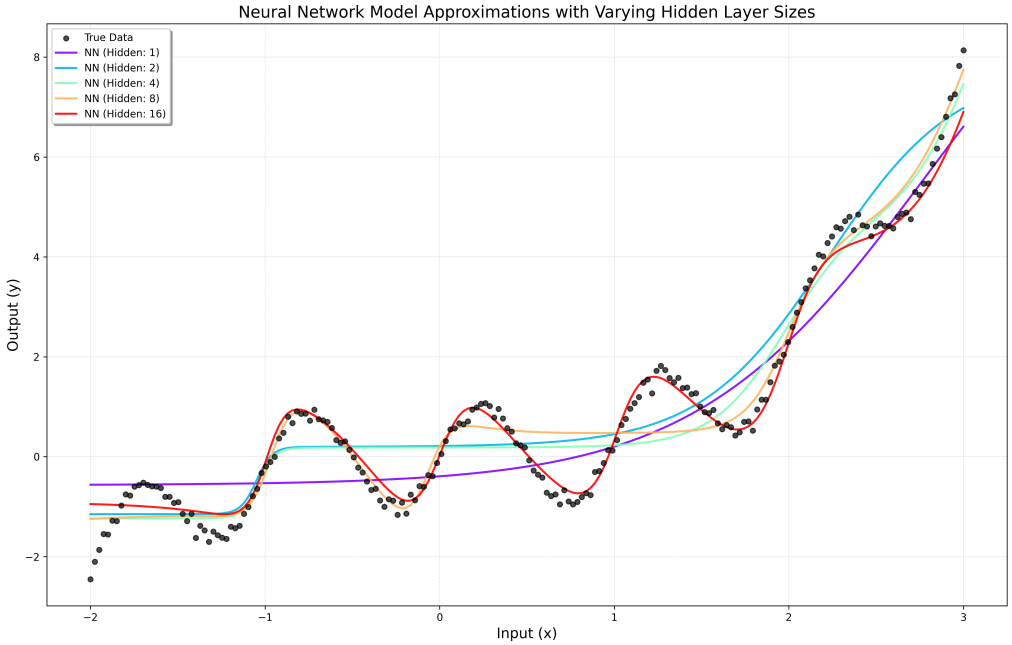

Figure 5: neural models with different sizes

5.1.7 Tests of performance between standard, optimized, and openMP models

const int DATA SIZE is 100; // Fixed data size
const int EPOCHS = 500; // Number of training epochs
const int BATCH SIZE = 5; // Batch size
hidden sizes = 2, 4, 8, 16, 32, 64, 12 // Testing a wider range of hidden sizes
adam lr = 0.03

The performance: As you can see the optimized model is a constant speed-up compared to the serial model, where using more threads gives an exponential speed-up.

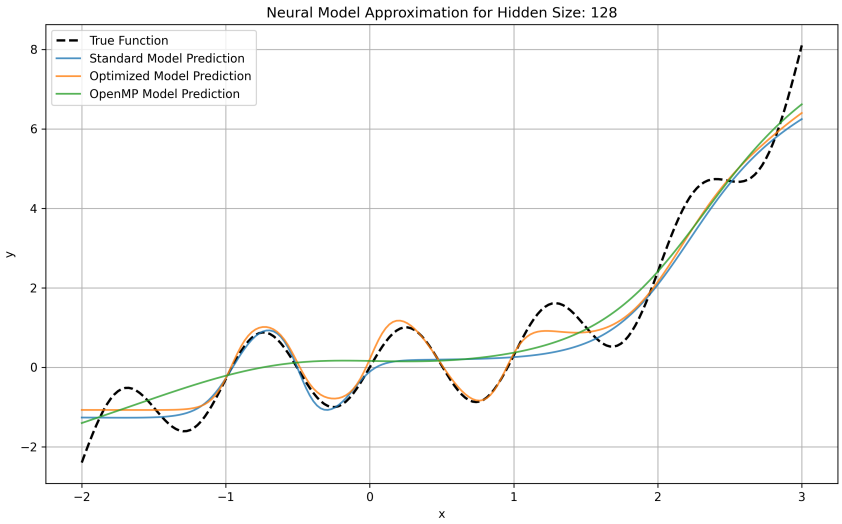

Figure 6: neural accuracy between different accelerated models

Talking about this approximation, we can see that the optimized and standard models are good approximations, but when we talk about AI, too much approximation can lead to overfitting. Whereas the parallelized model is not that approximated, because the averaged gradients somehow destroy the SGD effect, it generally approximates well the trend.

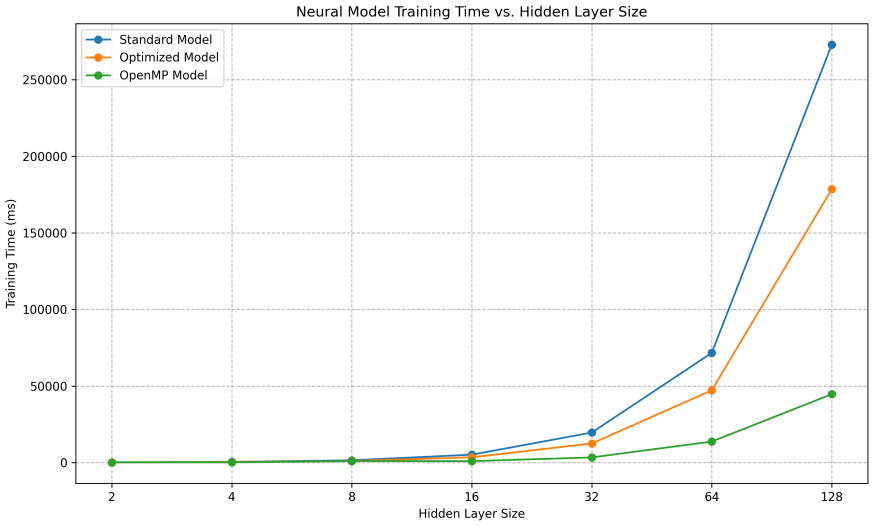

Figure 7: neural performance between different accelerated models

5.1.8 Architecture

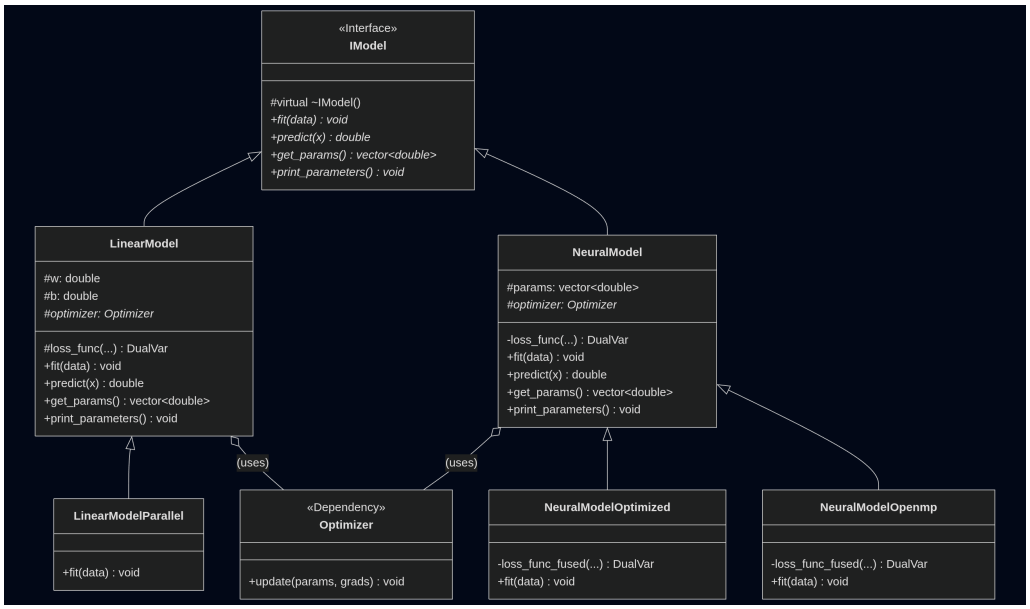

Figure 8: neural model architectures UML

I want to show some advanced implementations that we took into considerations:

- c++20 span to optimize the access to contiguous memory in forward
- rather than using vectors of vectors, we used a single vector to save weights
- use of lambda expressions to calculate the gradients.
- use of open mp to parallelize the calculation

We optimized the calculating based on a perf study, the study showed which part of the code is most time-consuming.

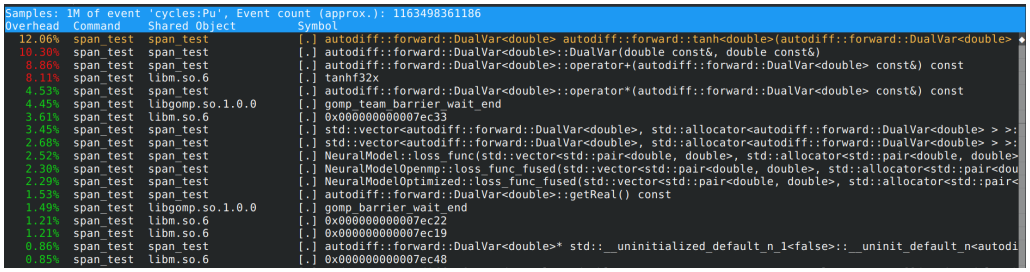

Figure 9: Running process of neural model

As you can see in the processes, the most time-consuming operations are tanh activation function, the creation of dual var, and other operations such as +-*/.
1. The allocation of dualvar are on the stack, this part could be simplified if we didn't create a new dualvar everytime after each operation. we would have saved 10 percent of computing resources
2. The operations of these basic and tanh operations are can't be simplified, so a rational way is to parallelize these long operations, luckily they the operations between each pair of weights and data are independent. So we used openmp to parallelize it. The speed up is worth the overhead.
3. We could have exploited mixed precision operations, we could have used float instead of double to save the memory from some operations. This would make a significant gain on larger neural models.

5.2 Newton Solver Implementation

5.2.1 Algorithm Implementation

The `Newton` class implements the classic Newton-Raphson method for solving $\mathbf{f}(\mathbf{x}) = \mathbf{0}$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{J}_f(\mathbf{x}_k)^{-1}\mathbf{f}(\mathbf{x}_k) \tag{13}$$

The solver includes configurable convergence criteria through `NewtonOpts`, monitoring both step size and residual convergence:

$$\text{Step size:} \quad \|\Delta\mathbf{x}\|_1 < \text{tol} \tag{14}$$

$$\text{Residual:} \quad \|\mathbf{f}(\mathbf{x})\|_1 < \text{tol} \tag{15}$$

5.2.2 Jacobian Computation

1. **Forward-Mode Jacobian** (`ForwardJac`): Uses forward-mode automatic differentiation
2. **Reverse-Mode Jacobian** (`ReverseJac`): Uses reverse-mode automatic differentiation, optimal for systems with many inputs and few outputs
3. **CUDA Jacobian** (`CudaJac`): Uses GPU-accelerated forward mode

Each Jacobian implementation computes both the function evaluation and its Jacobian matrix, then solves the linear system $\mathbf{J}\Delta\mathbf{x} = -\mathbf{f}$ using Eigen's LU decomposition with full pivoting.