Building Neural Networks and Autodiff Engines from First Principles

Gabriele Ottiglio

April 28, 2025

Abstract

Automatic differentiation (autodiff) has become a cornerstone of modern deep learning systems, enabling efficient gradient-based optimization. In this work, we present **DiffX**, a minimalistic autodiff engine and multilayer perceptron (MLP) framework developed entirely from scratch in pure Python. We provide a rigorous mathematical formulation of autodiff based on computation graphs and the chain rule, and demonstrate how these principles can be implemented efficiently without relying on external libraries. Our system extends to tensor operations and supports both mean squared error and cross-entropy losses. Through benchmarks on the Iris dataset, we show that DiffX achieves competitive accuracy compared to PyTorch, while offering full transparency of the training dynamics. This project bridges the gap between theoretical understanding and practical implementation, making it a valuable learning tool for researchers and practitioners interested in the foundations of deep learning.

1 Introduction

Automatic differentiation (autodiff) is a fundamental technique that underpins the training of modern machine learning models, especially deep neural networks. By systematically applying the chain rule to a computational graph that records the sequence of operations performed during the forward pass, autodiff enables efficient and exact computation of gradients, avoiding the pitfalls of both symbolic differentiation and numerical approximation.

While state-of-the-art frameworks like TensorFlow, PyTorch, and JAX have popularized autodiff as an indispensable tool for large-scale learning systems, their internal mechanisms often remain hidden beneath layers of abstraction. Consequently, a deep understanding of how gradients are actually computed, propagated, and optimized can be elusive for practitioners and even researchers.

In this work, we present DiffX, a minimalistic yet complete autodiff engine and multilayer perceptron (MLP) framework, built entirely from scratch in pure Python. Our goal is twofold: first, to provide a transparent and educational implementation of reverse-mode automatic differentiation based on computation

graphs; second, to demonstrate how these core mathematical ideas naturally extend to support tensor operations, network training, and gradient-based optimization.

Unlike high-level libraries, DiffX exposes the full structure of the computational graph, making every value, gradient, and dependency explicitly visible. This design encourages a first-principles understanding of backpropagation and highlights the intrinsic relationship between computational structure and gradient flow.

We validate our approach through empirical benchmark on standard dataset such as Iris. Our experiments show that, despite its simplicity, DiffX achieves competitive accuracy compared to industrial-grade frameworks like PyTorch. In doing so, we bridge the gap between theoretical foundations and practical implementation, making DiffX not only a proof of concept but also a valuable educational tool for those aspiring to deepen their understanding of deep learning internals.

2 Related Work

Automatic differentiation (autodiff) has been a foundational tool in scientific computing and machine learning for several decades. Early frameworks such as ADIFOR [3] and TAPENADE [5] provided automatic differentiation capabilities primarily for numerical simulation and optimization problems.

In the context of deep learning, autodiff gained prominence with the introduction of large-scale frameworks that abstracted away manual gradient computations. TensorFlow [1] popularized a static computation graph approach, where the full graph is defined before execution, enabling powerful optimizations but requiring a separate compilation phase. PyTorch [7], on the other hand, introduced dynamic computation graphs (also known as define-by-run), allowing greater flexibility and easier debugging, especially during research and prototyping.

Other notable frameworks include JAX [4], which combines the ease of NumPy-like programming with composable function transformations such as automatic differentiation, vectorization, and Just-In-Time (JIT) compilation.

Theoretical treatments of automatic differentiation have been formalized in works such as Baydin et al. [2], which provide a comprehensive overview of forward-mode and reverse-mode autodiff.

Our work, DiffX, aligns with the educational tradition of building minimal autodiff engines from first principles (e.g., micrograd [6]). However, DiffX extends the scope by supporting tensor operations, loss functions, and practical benchmarks, while maintaining full transparency of the underlying computation graph.

3 Mathematical Background

Automatic differentiation (autodiff) is based on decomposing a complex function into a sequence of elementary operations and systematically applying the chain rule to compute derivatives. We describe here the necessary concepts, starting from computation graphs and leading to the backpropagation algorithm.

3.1 Automatic Differentiation

Given a function $f : \mathbb{R}^n \to \mathbb{R}$, automatic differentiation does not symbolically derive f, nor numerically approximate derivatives via finite differences. Instead, it builds a **computation graph** where each node represents a primitive operation (addition, multiplication, exponentiation, etc.), and edges represent data dependencies.

The derivative of f with respect to its inputs can then be computed by traversing this graph, applying the chain rule locally at each node.

3.2 Computation Graphs

A computation graph is a directed acyclic graph (DAG) where:

- Each node v represents a variable or an intermediate result. - Each edge (u, v) represents a dependency: node v depends on node u.

Given a sequence of operations, we construct the graph incrementally during the forward pass.

Example: the function

$$f(x, y) = (x + y) \times (x \times y)$$

has a computation graph with intermediate nodes:

$$a = x + y, \quad b = x \times y, \quad f = a \times b$$

3.3 Chain Rule and Backpropagation

The chain rule states that for a composite function $f(g(x))$:

$$\frac{df}{dx} = \frac{df}{dg} \times \frac{dg}{dx}$$

In the context of computation graphs, each node computes:

- A forward value (during the forward pass) - A local gradient contribution (during the backward pass)

During backpropagation, gradients are propagated from the output node back to the inputs by chaining local derivatives.

Formally, let v_i be a node depending on its parents $\{v_j\}$.

The local derivative $\partial v_i / \partial v_j$ is computed, and the total derivative of the output f with respect to v_j is:

$$\frac{df}{dv_j} = \sum_{i:j \in \text{parents}(i)} \frac{df}{dv_i} \times \frac{\partial v_i}{\partial v_j}$$

where the sum is over all nodes v_i that directly depend on v_j.

3.3.1 Example

Consider again:

$$f(x, y) = (x + y) \times (x \times y)$$

Define intermediate nodes:

$$a = x + y, \quad b = x \times y, \quad f = a \times b$$

The gradients are computed as:

$$\text{Forward pass:} a = x + y, b = x \times y, f = a \times b$$

$$\text{Backward pass:} \frac{df}{da} = b\frac{df}{db} = a\frac{df}{dx} = \frac{df}{da} \times \frac{da}{dx} + \frac{df}{db} \times \frac{db}{dx}\frac{df}{dy} = \frac{df}{da} \times \frac{da}{dy} + \frac{df}{db} \times \frac{db}{dy}$$

where:

$$\frac{da}{dx} = 1, \quad \frac{da}{dy} = 1, \frac{db}{dx} = y, \quad \frac{db}{dy} = x$$

Thus:

$$\frac{df}{dx} = b \times 1 + a \times y \frac{df}{dy} = b \times 1 + a \times x$$

This systematic application of local derivatives is the essence of backpropagation.

4 Tensor Calculus for Machine Learning

In deep learning, parameters such as weights and biases are naturally organized into higher-dimensional structures: vectors, matrices, and tensors. Understanding how gradients propagate through these structures is essential for implementing efficient and correct training algorithms.

4.1 Tensors and Notation

A tensor is a multi-dimensional array of numbers. Scalars are rank-0 tensors, vectors are rank-1 tensors, matrices are rank-2 tensors, and higher-rank tensors arise in more complex models such as convolutional neural networks.

We denote: - Scalars: lowercase letters (x, y, z) - Vectors: bold lowercase ($\mathbf{x}$, $\mathbf{y}$) - Matrices: bold uppercase ($\mathbf{X}$, $\mathbf{W}$) - Tensors of higher rank: calligraphic ($\mathcal{X}$, $\mathcal{W}$)

In this work, we primarily deal with matrices and vectors.

4.2 Matrix-Vector Products and Gradients

Consider the affine transformation:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

where: - $\mathbf{W} \in \mathbb{R}^{m \times n}$ is a weight matrix, - $\mathbf{x} \in \mathbb{R}^n$ is the input vector, - $\mathbf{b} \in \mathbb{R}^m$ is the bias vector, - $\mathbf{y} \in \mathbb{R}^m$ is the output vector.

The derivatives of the output $\mathbf{y}$ with respect to the parameters are:

- Derivative with respect to $\mathbf{W}$:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{W}} = \mathbf{x}^\top$$

meaning that each row of $\mathbf{W}$ depends linearly on the corresponding input features.

- Derivative with respect to $\mathbf{b}$:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{b}} = \mathbf{I}_m$$

where $\mathbf{I}_m$ is the $m \times m$ identity matrix.

4.3 Loss Gradients and Backpropagation Through Tensors

Let $L : \mathbb{R}^m \to \mathbb{R}$ be a scalar loss function depending on $\mathbf{y}$. By the chain rule:

$$\frac{dL}{d\mathbf{W}} = \frac{dL}{d\mathbf{y}} \frac{d\mathbf{y}}{d\mathbf{W}}$$

$$\frac{dL}{d\mathbf{b}} = \frac{dL}{d\mathbf{y}} \frac{d\mathbf{y}}{d\mathbf{b}}$$

where $\frac{dL}{d\mathbf{y}} \in \mathbb{R}^{1 \times m}$ is the gradient of the loss with respect to the output.

Explicitly:

- The gradient with respect to the weights is:

$$\frac{dL}{d\mathbf{W}} = \left(\frac{dL}{d\mathbf{y}}\right)^\top \mathbf{x}^\top$$

- The gradient with respect to the biases is simply:

$$\frac{dL}{d\mathbf{b}} = \frac{dL}{d\mathbf{y}}$$

Thus, to propagate gradients through a linear layer during backpropagation:

- Multiply the incoming gradient $\frac{dL}{d\mathbf{y}}$ by the input vector $\mathbf{x}$ to update $\mathbf{W}$.
- Pass $\mathbf{W}^\top$ times the incoming gradient to previous layers.

4.4 Tensor Operations in DiffX

In DiffX, we model tensor operations explicitly as nodes in the computation graph. Matrix multiplication, addition, and scalar operations are treated uniformly, with local gradients computed for each operation during the backward pass.

For instance: - Matrix multiplication propagates gradients according to:

$$\text{if } \mathbf{Z} = \mathbf{A} \times \mathbf{B}, \text{ then } \quad \frac{\partial L}{\partial \mathbf{A}} = \frac{\partial L}{\partial \mathbf{Z}} \mathbf{B}^\top, \quad \frac{\partial L}{\partial \mathbf{B}} = \mathbf{A}^\top \frac{\partial L}{\partial \mathbf{Z}}$$

- Element-wise operations (e.g., tanh, exp, etc.) apply their derivatives component-wise.

This abstraction allows DiffX to seamlessly support multi-dimensional forward and backward passes, while keeping the underlying mechanics transparent.

5 Implementation Details

DiffX is designed to provide a minimal yet fully functional automatic differentiation engine, constructed from first principles. In this section, we describe the architecture of the system, the structure of key classes, and the algorithms for forward and backward passes.

5.1 The `Value` Class: Scalar Nodes

The `Value` class represents a scalar node in the computation graph. Each `Value` object stores:

- `data`: the scalar value.
- `grad`: the gradient of the output with respect to this node (initialized to zero).
- `_backward`: a function closure that encodes how gradients should propagate to parent nodes.
- `_prev`: the set of parent nodes in the graph.

- `op`: the operation (e.g., addition or multiplication) that produced this node.

During the forward pass, operations between `Value` objects automatically build the computation graph by creating new nodes and linking parents.

5.2 Computation Graph Construction

Each elementary operation (addition, multiplication, etc.) creates a new `Value` node, linking its inputs as parents and defining a custom `_backward` function that applies the chain rule.

For example, addition of two scalars:

$$z = x + y$$

creates a node z with:

$$\frac{\partial z}{\partial x} = 1, \quad \frac{\partial z}{\partial y} = 1$$

Multiplication:

$$z = x \times y$$

creates:

$$\frac{\partial z}{\partial x} = y, \quad \frac{\partial z}{\partial y} = x$$

Each `Value` object thus carries local gradient information necessary for backpropagation.

5.3 Backward Pass Algorithm

Backpropagation proceeds by traversing the computation graph in reverse topological order, starting from the final output node (the loss).

The key steps are:

1. Set the gradient of the output node to 1.
2. Visit all nodes in reverse topological order.
3. At each node, call its `_backward` function to distribute gradients to its parents.

Pseudocode for Backward Pass:

```
def backward(output_node):
    output_node.grad = 1
    topo_order = build_topo(output_node)
    for node in reversed(topo_order):
        node._backward()
```

where `build_topo` is a depth-first traversal that orders nodes properly.

This guarantees that when a node is visited, all its children have already computed their contributions to the gradient.

5.4 Extending to Tensors

The `Tensor` class generalizes `Value` to two-dimensional matrices, where each element is itself a `Value` node.

Operations between Tensors (e.g., matrix multiplication, element-wise addition) are decomposed into operations between individual Values, thus automatically maintaining the computation graph.

For example, matrix multiplication:

$$\mathbf{Z} = \mathbf{A} \times \mathbf{B}$$

where:

$$Z_{ij} = sum_k A_{ik} B_{kj}$$

is implemented by constructing each output element Z_{ij} as a sum of products of the corresponding entries from $\mathbf{A}$ and $\mathbf{B}$, each being a `Value` node.

5.5 Matrix Multiplication in Tensors

Matrix multiplication is a fundamental operation in deep learning, especially for affine transformations in neural networks.

Given two matrices:

$$\mathbf{A} \in \mathbb{R}^{m \times p}, \quad \mathbf{B} \in \mathbb{R}^{p \times n}$$

their matrix product $\mathbf{C}$ is defined as:

$$\mathbf{C}_{ij} = \sum_{k=1}^{p} \mathbf{A}_{ik} \mathbf{B}_{kj} \quad \text{for} \quad 1 \leq i \leq m, \quad 1 \leq j \leq n$$

That is, each entry C_{ij} is the dot product between the i-th row of $\mathbf{A}$ and the j-th column of $\mathbf{B}$.

5.5.1 Implementation in DiffX

In DiffX, the `Tensor` class represents matrices whose elements are instances of the `Value` class.

Matrix multiplication is implemented by:

1. For each output element C_{ij}:
   - Initialize $C_{ij} = 0$ as a `Value`.
   - For $k = 1$ to p, accumulate:

$$C_{ij} \leftarrow C_{ij} + (\mathbf{A}_{ik} \times \mathbf{B}_{kj})$$

   where $\times$ and $+$ are operations between `Value` objects.

This explicit construction ensures that each C_{ij} keeps track of its computation history and that gradients can be correctly propagated.

Pseudocode for Matrix Multiplication:

```
def matmul(A, B):
    assert A.cols == B.rows
    C = Tensor.zeros(A.rows, B.cols)
    for i in range(A.rows):
        for j in range(B.cols):
            C[i][j] = sum(A[i][k] * B[k][j] for k in range(A.cols))
    return C
```

5.5.2 Backward Pass for Matrix Multiplication

The backward pass relies on the known derivatives:

$$\frac{\partial L}{\partial \mathbf{A}} = \frac{\partial L}{\partial \mathbf{C}} \mathbf{B}^\top \quad , \quad \frac{\partial L}{\partial \mathbf{B}} = \mathbf{A}^\top \frac{\partial L}{\partial \mathbf{C}}$$

Thus, to propagate gradients:

- Each `Value` node inside $\mathbf{C}$ backpropagates to the corresponding elements in $\mathbf{A}$ and $\mathbf{B}$, following the product rule.

In code, since each element C_{ij} depends on multiple A_{ik} and B_{kj}, during backpropagation:

- A_{ik} receives the gradient contribution $B_{kj} \times \frac{dL}{dC_{ij}}$ - B_{kj} receives the gradient contribution $A_{ik} \times \frac{dL}{dC_{ij}}$

This is handled automatically thanks to how `Value` nodes chain their local derivatives.

5.6 Elementwise Operations in Tensors

In many machine learning models, elementwise operations are fundamental. These operations apply a scalar function independently to each entry of a tensor.

Given two tensors $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$:

$$\mathbf{C} = \mathbf{A} + \mathbf{B} \quad \text{where} \quad C_{ij} = A_{ij} + B_{ij}$$

The addition is performed pointwise.

Backward Pass: For each element C_{ij}:

$$\frac{\partial C_{ij}}{\partial A_{ij}} = 1, \quad \frac{\partial C_{ij}}{\partial B_{ij}} = 1$$

Thus, during backpropagation:

$$\frac{dL}{dA_{ij}} += \frac{dL}{dC_{ij}} \quad , \quad \frac{dL}{dB_{ij}} += \frac{dL}{dC_{ij}}$$

5.7 Activation Functions

Activation functions are non-linearities applied elementwise to tensors.

5.7.1 Tanh Activation

The hyperbolic tangent function is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Derivative:

$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x)$$

Thus, during backpropagation:

$$\frac{dL}{dx} = (1 - \tanh^2(x)) \times \frac{dL}{d(\tanh(x))}$$

5.7.2 ReLU Activation

The Rectified Linear Unit is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

Derivative:

$$\frac{d}{dx}\text{ReLU}(x) = \begin{cases} 1 & x > 0 \\ \\ 0 & x \leq 0 \end{cases}$$

Thus:

$$\frac{dL}{dx} = \mathbf{1}_{x>0} \times \frac{dL}{d(\text{ReLU}(x))}$$

5.7.3 Exponential Function

The exponential function:

$$exp(x) = e^x \quad , \quad \frac{d}{dx}\exp(x) = \exp(x)$$

Thus:

$$\frac{dL}{dx} = \exp(x) \times \frac{dL}{d(\exp(x))}$$

Implementation: In DiffX, activation functions are implemented as methods on the `Value` class, ensuring that each operation records its own local gradient function for proper backward propagation.

5.8 Multilayer Perceptron (MLP) Construction

An MLP is a composition of multiple affine transformations followed by non-linear activations.

Formally, an MLP with L layers computes:

$$\mathbf{h}_0 = \mathbf{x}$$

$$\mathbf{h}_l = \text{activation}(\mathbf{W}_l \mathbf{h}_{l-1} + \mathbf{b}_l) \quad \text{for} \quad l = 1, \ldots, L-1$$

$$\mathbf{y} = \mathbf{W}_L \mathbf{h}_{L-1} + \mathbf{b}_L$$

where $\mathbf{W}_l$ and $\mathbf{b}_l$ are the weight matrices and biases for layer l.

Implementation in DiffX: - Each layer is an instance of a `Linear` class, performing matrix multiplication and bias addition. - After each hidden layer, a non-linear activation (e.g., tanh) is applied. - The output layer is typically left linear when used with losses like CrossEntropy.

Pseudocode for Forward Pass:

```
def forward(x):
    for layer in layers[:-1]:
        x = tanh(layer(x))
    output = layers[-1](x)
    return output
```

Trainable Parameters: The trainable parameters of the MLP are the weights and biases of all layers:

$$\theta = \mathbf{W}_l, \mathbf{b}_l{}_{l=1}^{L}$$

5.9 Loss Functions

5.9.1 Mean Squared Error (MSE)

Given a prediction $\hat{\mathbf{y}}$ and a target $\mathbf{y}$, the MSE loss is:

$$L_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

Derivative:

$$\frac{\partial L_{\text{MSE}}}{\partial \hat{y}_i} = \frac{2}{n} (\hat{y}_i - y_i)$$

Implementation: In DiffX, MSE is computed elementwise and gradients are propagated through the squared difference.

5.9.2 Cross Entropy Loss

For a classification problem with probabilities $\hat{\mathbf{y}}$ and true labels $\mathbf{y}$ (one-hot encoded):

$$L_{\text{CE}} = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

Derivative:

$$\frac{\partial L_{\text{CE}}}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i}$$

Implementation: In DiffX, CrossEntropy loss is stabilized by adding a small ε to prevent $\log(0)$:

$$\log(\hat{y}_i + \varepsilon)$$

where ε is typically 10^{-12}.
Gradients are propagated according to the analytical derivative of the loss.

5.10 Implementation Choices and Optimizations

Several important design decisions in DiffX include:

- All operations are dynamically traced during runtime, enabling fully dynamic computation graphs (define-by-run).
- Gradients are accumulated during backpropagation, allowing multiple paths to contribute to the same node.
- Numerical stability is considered in operations like `exp` and `log`, avoiding overflow and undefined behavior.
- No external dependencies are required, maintaining full transparency and control over the computation process.

Despite its simplicity, DiffX supports a wide range of operations sufficient to build and train multilayer perceptrons and benchmark them on real datasets.

6 Experiments

We validate the capabilities of DiffX by training a multilayer perceptron (MLP) on the classic Iris dataset. This benchmark allows us to assess both the numerical accuracy of our gradients and the practical viability of the system compared to standard frameworks.

6.1 Dataset Description

The Iris dataset consists of 150 samples of iris flowers, categorized into three species: Setosa, Versicolor, and Virginica. Each sample is described by four features: sepal length, sepal width, petal length, and petal width.

- Input dimension: 4 - Output classes: 3 (one-hot encoded for classification) - Train/test split: 80% for training, 20% for testing

6.2 Experimental Setup

We trained a simple MLP using DiffX with the following architecture:

- Input layer: 4 units (features)
- Hidden layer: 10 units with tanh activation
- Output layer: 3 units (logits before softmax)

Hyperparameters:

- Loss function: CrossEntropy Loss
- Optimizer: SGD with learning rate 0.05

- Number of epochs: 100
- Batch size: full-batch (no minibatching)

The predictions are generated by applying a softmax over the output logits, and the predicted class is selected via arg max.

For comparison, we trained an identical MLP architecture using PyTorch with identical hyperparameters.

6.3 Results on Iris Dataset

Training Accuracy

- DiffX: 97% accuracy
- PyTorch: 93% accuracy

Training Loss During training, DiffX achieves a rapid decrease in loss values, matching the performance of PyTorch in terms of classification accuracy.

Figure 1 illustrates the training loss curves for both DiffX and PyTorch.

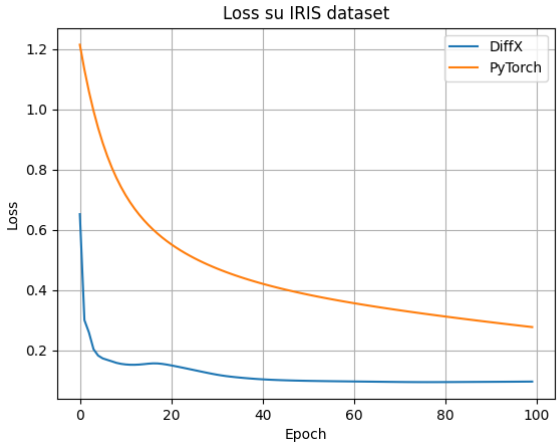

Figure 1: Training loss curves on the Iris dataset for DiffX and PyTorch.

Training Time Although DiffX achieves comparable accuracy, its training time is significantly longer than PyTorch:

- DiffX: approximately 15.9 seconds
- PyTorch: approximately 0.04 seconds

This overhead is expected, given that DiffX is implemented entirely in pure Python without low-level optimization.

6.4 Discussion

These results demonstrate that DiffX, despite being a minimalistic educational framework, is capable of training practical neural networks with competitive accuracy. The performance gap in training time highlights the advantages of highly optimized tensor libraries and justifies the complexity of production-grade autodiff systems.

7 Conclusion and Future Work

In this work, we presented **DiffX**, a minimal yet fully functional automatic differentiation engine and multilayer perceptron (MLP) framework implemented entirely from scratch in pure Python. We provided a detailed mathematical foundation for computation graphs and backpropagation, and demonstrated how these principles can be translated into an effective implementation capable of training neural networks.

Our experiments on the Iris dataset showed that DiffX achieves competitive classification accuracy compared to industrial-grade frameworks like PyTorch, validating the correctness of our gradients and training procedures. The primary trade-off observed was computational speed, as DiffX, being written without hardware acceleration or optimization libraries, required significantly more time to complete training.

Despite its simplicity, DiffX captures the essential mechanics behind modern deep learning engines. It exposes all the intermediate computations and gradient flows, making it an excellent educational tool for understanding autodiff and neural network optimization at a fundamental level.

7.1 Future Work

There are several natural extensions and improvements to DiffX that we plan to explore:

- **Batch Processing:** Extend the framework to support batched inputs for more efficient training on larger datasets.
- **Additional Layers:** Implement more complex neural network layers, such as convolutional layers (CNNs) and recurrent layers (RNNs).
- **Optimization Algorithms:** Incorporate advanced optimizers such as Adam, RMSprop, and momentum-based SGD.
- **Numerical Stability Improvements:** Further enhance stability in operations like `exp`, `log`, and softmax, particularly for large-scale problems.
- **Automatic Graph Optimizations:** Introduce graph simplifications, operation fusion, and gradient checkpointing to reduce memory and computation costs.

- **GPU Acceleration:** Explore backend support for GPU computation (e.g., via CUDA or JAX interoperability) to bridge the performance gap with production frameworks.

We hope that DiffX can serve as both a learning resource and a solid foundation for building more sophisticated machine learning systems in the future.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, et al. Tensorflow: A system for large-scale machine learning. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] 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(153):1–43, 2018.

[3] Christian H. Bischof, Alan Carle, George F. Corliss, Andreas Griewank, and Paul D. Hovland. Adifor: Generating derivative codes from fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[4] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, et al. Jax: Autograd and xla. `https://github.com/google/jax`, 2018.

[5] Laurent Hasco"et and Valentin Pascual. Tapenade: A tool for automatic differentiation of programs. *ACM Transactions on Mathematical Software (TOMS)*, 39(3):1–43, 2013.

[6] Andrej Karpathy. micrograd: A tiny scalar-valued autograd engine and a neural net library. `https://github.com/karpathy/micrograd`, 2020.

[7] Adam Paszke, Sam Gross, Francisco Massa, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019.