Enhancing LLM Training Efficiency with Clad for Backpropagation

Rohan Timmaraju, June 2025
Compiler Research Group

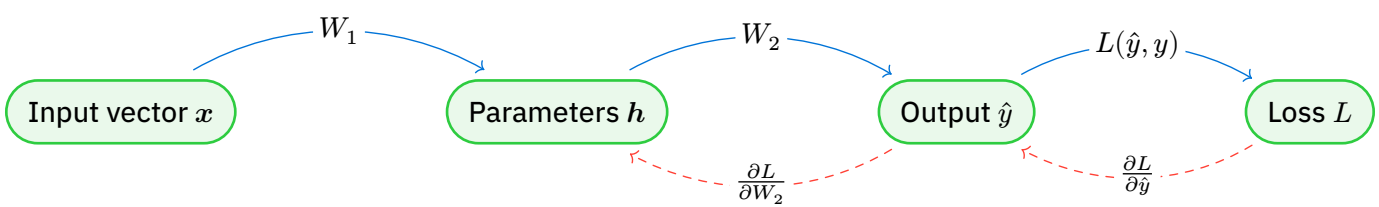
Introduction

The Challenge of LLM Training

- Large Language Models (LLMs) like GPT, Claude, Llama, etc. are groundbreaking, but incredibly computationally expensive to train.
- Python frameworks (PyTorch, TensorFlow, etc.) dominate:
  - Pros: Flexibility, rich ecosystem, ease of use.
  - Cons: Performance overhead (interpreter, dynamic computation graphs), memory usage
- Need for efficient solutions, especially to integrate with HPC/C++ environments.

Core Idea: Compiler-level AD for Backpropagation

- Neural networks use reverse mode automatic differentiation to compute the gradients of model parameters wrt. to a loss function – this is called backpropagation.
  - These gradients are then used to update model parameters during the training loop
- Idea: construct the neural network in C++ so that we can use `clad::gradient` to statically generate the backpropagation code at compile time.
- Goal: allow for compiler optimizations across the entire computation graph and training loop to improve performance



Technical Overview

`cladtorch`: A Custom C++ Tensor Library

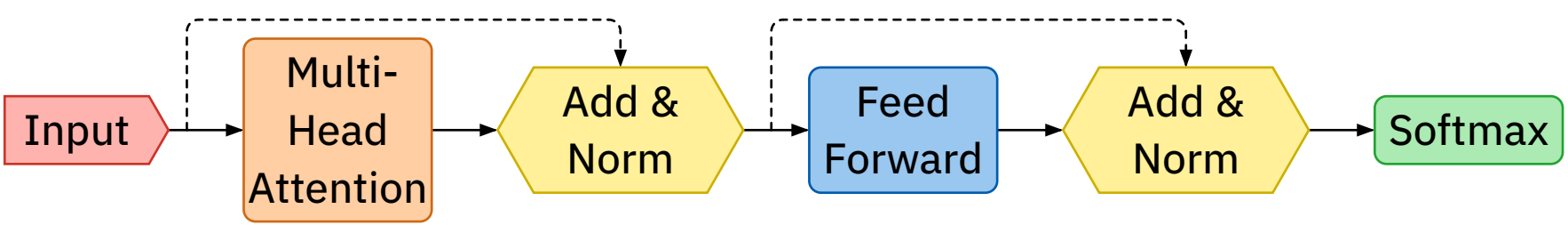
- Neural networks are composed of large sequences of tensor operations (matrix multiplication, convolution, activation functions, etc.)
- We need a C++ tensor library to express these operations in a way that integrates with Clad to differentiate them.
- Design Goals:
  - Familiar API for PyTorch users
  - Transparency to clad – should be able to differentiate all operations/kernels
  - Efficiency and performance (minimal dynamic execution overhead)
- Key Question/Challenge: Does it need to be a special case in Clad? Or can we improve the existing `clad::gradient` implementation enough to support it?

Main Workflow

```
using cladtorch::Tensor;
struct NeuralNetwork {
  Layer1 l1; Layer2 l2;
  Tensor forward(const Tensor &input) const {
    softmax(l1.forward(input) + l2.forward(input));
  }
};
float nn_loss(const NeuralNetwork &nn, Tensor &input, Tensor &output) {
  return cross_entropy_loss(nn.forward(input), output);
}
for ([input, output] : training_data) {
  NeuralNetwork nn, d_nn = {0}; // gradients are accumulated in d_nn
  auto grad = clad::gradient(nn_loss, "0");
  grad.execute(nn, input, output, &d_nn);
  nn.update_weights(d_nn, learning_rate);
}
```

Project Timeline

- Phase 1: Implement `cladtorch` tensor library
  - Develop an efficient C++ tensor library.
  - Integrate with Clad for automatic differentiation
    - Either by extending Clad's existing functionality or by implementing custom logic for Tensor operations
- Phase 2: Implement a Machine Learning library on top of `cladtorch`
  - Implement common neural network layers (e.g., linear, convolutional, activation functions), loss functions (e.g., cross-entropy, MSE), and operations.
  - Implement key LLM layers (e.g., attention, transformer blocks)



Project Timeline (cont.)

- Phase 3: Implement the GPT-2 LLM architecture
  - Implement the GPT-2 architecture using the `cladtorch` library for training
  - Train the model on a small dataset to validate functionality
  - Ensure correctness of backpropagation and gradient accumulation
- Phase 4: Benchmarking, Optimization, and Extension
  - Benchmark the performance of `cladtorch` and the ML library against manual implementations, PyTorch, and other frameworks.
  - Optimize performance based on results, potentially integrating OpenMP etc.
  - Potentially extend to other architectures (e.g. Llama, Mistral) and tasks (e.g. fine-tuning, inference).
  - Extend ML library with additional features, e.g. for Physics-based ML tasks, solver-in-the-loop training, etc.

Summary

Goals and Impact

Goals:

- Develop a C++ tensor library that integrates with Clad for efficient differentiation.
- Implement an ML library on top of `cladtorch` for training neural networks.
- C++ GPT-2 architecture implementation via Clad.
- Performance benchmarks (speed, memory) and detailed documentation.

Impact:

- Enable efficient LLM training in C++ & HPC environments.
- Provide a foundation for future research/efforts in compiler-based ML optimization.
- Insights for Clad’s usability in ML and LLM training.

Thank You