DASlab

HARVARD School of Engineering and Applied Sciences

CS265 Systems Project

Activation Checkpointing: Trading off compute for memory in deep neural network training

---

Introduction

The systems project for CS265 is designed to provide hands-on experience on the state-of-the-art systems for deep learning. It includes understanding the system architecture of modern deep learning frameworks, analyzing the compute memory trade-offs involved in training deep learning models and implementing an algorithm that navigates this trade-off. Systems projects will be done individually, each student is required to work on their own. This is a focused project that should not necessarily result in many lines of code (like the CS165 project), but will exercise your understanding of modern deep learning systems.

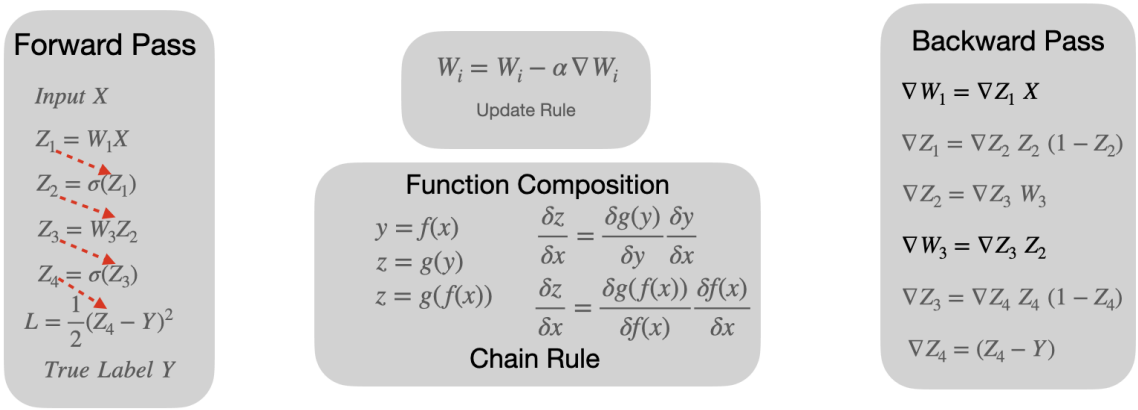
The goal of this project is to implement an activation checkpointing algorithm in PyTorch. The project is structured into three stages: the first stage involves creating a profiler to gather performance metrics during the training process, the second stage involves implementing an algorithm that determines which activations to checkpoint based on the profiler's statistics, and the final stage requires modifying the execution strategy to implement the decisions made by the algorithm.

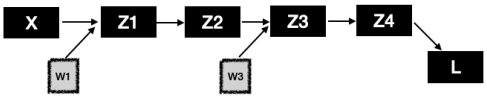
Students that finish the systems project quickly and want to work on research will be able to do so by exploring open research topics directly on top of their project.

Background

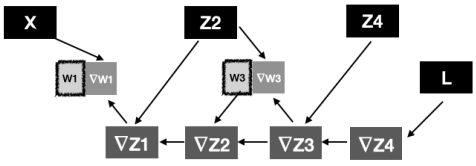
A typical neural network is a function that consists of multiple operations and is defined by a set of learnable parameters, known as weights (W1, W3). The example below illustrates a simple neural network composed of two linear operators followed by sigmoid activation functions and a mean squared error loss function. During a single training iteration, the neural network is provided with an input sample (X) and a target value/label (Y). The forward pass processes the input by passing it through various functions to generate a prediction. The difference between the prediction and the true value/label is referred to as the loss (L). The output of one function serves as the input for another and is called an activation/intermediate tensor/feature map (Z1, Z2, Z3, and Z4). These activations are stored in GPU memory during the forward pass and are used by the backward pass to calculate the weight gradients (∇W1, ∇W3).

The size of these activations is determined by the size of the input and weights, while the number of activations is determined by the depth of the network (or the number of operations in the network). The weights and their corresponding gradients typically reside in GPU memory throughout the lifetime of the iteration. The activations are stored when they are first generated (in forward pass) and are freed after their last use (in backward pass).





Forward Computational Graph



Backward Computational Graph

Motivation

Training neural networks is a process that demands significant computational resources, both in terms of memory and processing power. Recent trends indicate a consistent and exponential increase in the size of neural networks and the datasets used for their training. However, the memory capacity of GPUs, which are the most commonly used accelerators for training, has only seen linear growth over the past decade. This discrepancy has resulted in accelerator memory becoming a scarce and valuable resource. The peak memory required during training is directly proportional to the size of the model and the mini-batch size. Consequently, the limited memory capacity of GPUs imposes restrictions on the size of the model and/or the mini-batch size that can be used during training.

Problem

In our project, we focus on models whose parameters, gradients, and optimizer states, can be accommodated on a single GPU. To gain insight into the source of the substantial memory footprint during training, we conduct an analysis of peak memory consumption. The primary contributor to peak memory usage, accounting for approximately 70-85%, are the activations. Additionally, it's worth noting that the sequence in which the activations are generated and consumed is reversed, which leads to them lying idle in the memory for a long duration.

Solution Overview

To reduce the peak memory consumption, one of the most popular techniques recently has been “Activation Checkpointing”. Activation checkpointing (AC) addresses the issue by not storing all the activations in the memory during the forward pass. Instead, it stores only a subset of them and recomputes the others during the backward pass as needed. This approach trades off computation time for memory, as some computations need to be performed twice, but it can significantly reduce the memory requirements, enabling the training of larger models or the use of larger mini-batch sizes.

Project

In our project we will implement the algorithm for activation checkpointing in µ-TWO.

1. Inputs: We will experiment with two open source models (one vision and one LLM)
    a. Resnet-152
    b. Bert
2. AI Framework: We will be using PyTorch as our framework for implementation and experimentation.
3. Components to be built:
    a. Computation graph profiler,
    b. Activation checkpointing (AC) algorithm,
    c. Subgraph extractor and rewriter
4. Deliverables (Code Review and Demo) and a document with experimental analysis consisting of:
    a. Computation and memory profiling statistics and static analysis,
    b. Peak memory consumption vs mini-batch size bar graph (w and w/o AC),
    c. Iteration latency vs mini-batch size performance graph (w and w/o AC)

Phase 1: Graph Profiler (3 weeks - 35%)

In the initial phase, our primary task is to construct a comprehensive computational graph. This graph will encapsulate all operations from the forward, backward, and optimizer steps within a single iteration. The nodes within this graph symbolize individual operations, while the edges represent the dependencies between input and output data. The profiler's job is :

1. Collecting data on the computation time and memory usage of each operator when the graph operations are executed in topological order.
2. Categorizing the inputs and outputs of each operation as a parameter, gradient, activation, optimizer state, or other types.
3. Conducting static data analysis on activations by documenting the first and last use of each activation during the forward and backward passes.
4. Generating a peak memory breakdown graph using the collected statistics.

Phase 2: Activation Checkpointing algorithm (2 weeks - 20%)

Using the inputs from the profiler, implement the activation checkpointing algorithm in µ-TWO that decides the subset of the activations to be retained and the subset of the activations to be recomputed.

Phase 3: Graph Extractor and Rewriter (3 weeks - 45%)

Each activation that is chosen to be discarded in the forward pass needs to be recomputed in the backward pass when it is needed for gradient computation. To implement this we will extract the subgraph that computes the activation in the forward pass and then replicate it. This replicated sub-graph will be inserted into the backward pass just before it is required for gradient computation.

Suggested Timeline

Week 1: Familiarize yourself with training a simple model in PyTorch.
Week 2-4: Development of graph profiler.
Week 5-6: Implementing the recomputation algorithm from µ-TWO.
Week 7-9: Implementing the subgraph extractor and rewriter.

Midway Check-in:

1. Phase 1 completed.
2. Document with experimental analysis consisting of deliverables 4(a) and 4(b) [w/o AC].