Sparsity and Structured Matrices

CS6787 Lecture 13 — Fall 2019

Final project guidelines

Abstract — Due in a Week

- Should be like an abstract for a paper
  - Something that could (but doesn't need to) be the abstract for your final report.

- Typical length: about 6–8 sentences
  - Can be longer or shorter
  - But it's an advertisement/summary for your paper, not the paper itself

- Should fairly summarize your results, but does not have to be complete
  - E.g. you can say something like "Our method improves throughput by X% over an existing method" if you haven't run the experiment yet.

Things to cover in an abstract

- Setting/Motivation
  - What is the setting of the paper? Why should we care about it?
- Problem Statement/Scope
  - Within this setting, what problem did you set out to solve in the paper?
- Approach/Methodology
  - What did you do to solve the problem? What techniques did you develop?
- Results
  - What did you observe? How did your techniques perform?
- Conclusion:
  - What should we take away from your results? Why should we care about them?

An Example: HOGWILD!

Stochastic Gradient Descent (SGD) is a popular algorithm that can achieve state-of-the-art performance on a variety of machine learning tasks. Several researchers have recently proposed schemes to parallelize SGD, but all require performance-destroying memory locking and synchronization. This work aims to show using novel theoretical analysis, algorithms, and implementation that SGD can be implemented without any locking. We present an update scheme called HOGWILD! which allows processors access to shared memory with the possibility of overwriting each other's work. We show that when the associated optimization problem is sparse, meaning most gradient updates only modify small parts of the decision variable, then HOGWILD! achieves a nearly optimal rate of convergence. We demonstrate experimentally that HOGWILD! outperforms alternative schemes that use locking by an order of magnitude.

Setting/Motivation

Stochastic Gradient Descent (SGD) is a popular algorithm that can achieve state-of-the-art performance on a variety of machine learning tasks. Several researchers have recently proposed schemes to parallelize SGD, but all require performance-destroying memory locking and synchronization. This work aims to show using novel theoretical analysis, algorithms, and implementation that SGD can be implemented without any locking. We present an update scheme called HOGWILD! which allows processors access to shared memory with the possibility of overwriting each other's work. We show that when the associated optimization problem is sparse, meaning most gradient updates only modify small parts of the decision variable, then HOGWILD! achieves a nearly optimal rate of convergence. We demonstrate experimentally that HOGWILD! outperforms alternative schemes that use locking by an order of magnitude.

Problem Statement/Scope

Stochastic Gradient Descent (SGD) is a popular algorithm that can achieve state-of-the-art performance on a variety of machine learning tasks. Several researchers have recently proposed schemes to parallelize SGD, but all require performance-destroying memory locking and synchronization. This work aims to show using novel theoretical analysis, algorithms, and implementation that SGD can be implemented without any locking. We present an update scheme called HOGWILD! which allows processors access to shared memory with the possibility of overwriting each other's work. We show that when the associated optimization problem is sparse, meaning most gradient updates only modify small parts of the decision variable, then HOGWILD! achieves a nearly optimal rate of convergence. We demonstrate experimentally that HOGWILD! outperforms alternative schemes that use locking by an order of magnitude.

Approach/Methodology

Stochastic Gradient Descent (SGD) is a popular algorithm that can achieve state-of-the-art performance on a variety of machine learning tasks. Several researchers have recently proposed schemes to parallelize SGD, but all require performance-destroying memory locking and synchronization. This work aims to show using novel theoretical analysis, algorithms, and implementation that SGD can be implemented without any locking. We present an update scheme called HOGWILD! which allows processors access to shared memory with the possibility of overwriting each other's work. We show that when the associated optimization problem is sparse, meaning most gradient updates only modify small parts of the decision variable, then HOGWILD! achieves a nearly optimal rate of convergence. We demonstrate experimentally that HOGWILD! outperforms alternative schemes that use locking by an order of magnitude.

Results

Stochastic Gradient Descent (SGD) is a popular algorithm that can achieve state-of-the-art performance on a variety of machine learning tasks. Several researchers have recently proposed schemes to parallelize SGD, but all require performance-destroying memory locking and synchronization. This work aims to show using novel theoretical analysis, algorithms, and implementation that SGD can be implemented without any locking. We present an update scheme called HOGWILD! which allows processors access to shared memory with the possibility of overwriting each other's work. We show that when the associated optimization problem is sparse, meaning most gradient updates only modify small parts of the decision variable, then HOGWILD! achieves a nearly optimal rate of convergence. We demonstrate experimentally that HOGWILD! outperforms alternative schemes that use locking by an order of magnitude.

Conclusion/Takeaway

Stochastic Gradient Descent (SGD) is a popular algorithm that can achieve state-of-the-art performance on a variety of machine learning tasks. Several researchers have recently proposed schemes to parallelize SGD, but all require performance-destroying memory locking and synchronization. This work aims to show using novel theoretical analysis, algorithms, and implementation that SGD can be implemented without any locking. We present an update scheme called HOGWILD! which allows processors access to shared memory with the possibility of overwriting each other's work. We show that when the associated optimization problem is sparse, meaning most gradient updates only modify small parts of the decision variable, then HOGWILD! achieves a nearly optimal rate of convergence. We demonstrate experimentally that HOGWILD! outperforms alternative schemes that use locking by an order of magnitude.

Another Example: Deep Compression

Neural networks are both computationally intensive and memory intensive, making them difficult to deploy on embedded systems with limited hardware resources. To address this limitation, we introduce "deep compression", a three stage pipeline: pruning, trained quantization and Huffman coding, that work together to reduce the storage requirement of neural networks by 35× to 49× without affecting their accuracy. Our method first prunes the network by learning only the important connections. Next, we quantize the weights to enforce weight sharing, finally, we apply Huffman coding. After the first two steps we retrain the network to fine tune the remaining connections and the quantized centroids. Pruning, reduces the number of connections by 9× to 13×; Quantization then reduces the number of bits that represent each connection from 32 to 5. On the ImageNet dataset, our method reduced the storage required by AlexNet by 35×, from 240MB to 6.9MB, without loss of accuracy. Our method reduced the size of VGG-16 by 49× from 552MB to 11.3MB, again with no loss of accuracy. This allows fitting the model into on-chip SRAM cache rather than off-chip DRAM memory. Our compression method also facilitates the use of complex neural networks in mobile applications where application size and download bandwidth are constrained. Benchmarked on CPU, GPU and mobile GPU, compressed network has 3× to 4× layerwise speedup and 3× to 7× better energy efficiency.

Setting/Motivation

Neural networks are both computationally intensive and memory intensive, making them difficult to deploy on embedded systems with limited hardware resources. To address this limitation, we introduce "deep compression", a three stage pipeline: pruning, trained quantization and Huffman coding, that work together to reduce the storage requirement of neural networks by 35× to 49× without affecting their accuracy. Our method first prunes the network by learning only the important connections. Next, we quantize the weights to enforce weight sharing, finally, we apply Huffman coding. After the first two steps we retrain the network to fine tune the remaining connections and the quantized centroids. Pruning, reduces the number of connections by 9× to 13×; Quantization then reduces the number of bits that represent each connection from 32 to 5. On the ImageNet dataset, our method reduced the storage required by AlexNet by 35×, from 240MB to 6.9MB, without loss of accuracy. Our method reduced the size of VGG-16 by 49× from 552MB to 11.3MB, again with no loss of accuracy. This allows fitting the model into on-chip SRAM cache rather than off-chip DRAM memory. Our compression method also facilitates the use of complex neural networks in mobile applications where application size and download bandwidth are constrained. Benchmarked on CPU, GPU and mobile GPU, compressed network has 3× to 4× layerwise speedup and 3× to 7× better energy efficiency.

Problem Statement/Scope

Neural networks are both computationally intensive and memory intensive, making them difficult to deploy on embedded systems with limited hardware resources. To address this limitation, we introduce "deep compression", a three stage pipeline: pruning, trained quantization and Huffman coding, that work together to reduce the storage requirement of neural networks by 35× to 49× without affecting their accuracy. Our method first prunes the network by learning only the important connections. Next, we quantize the weights to enforce weight sharing, finally, we apply Huffman coding. After the first two steps we retrain the network to fine tune the remaining connections and the quantized centroids. Pruning, reduces the number of connections by 9× to 13×; Quantization then reduces the number of bits that represent each connection from 32 to 5. On the ImageNet dataset, our method reduced the storage required by AlexNet by 35×, from 240MB to 6.9MB, without loss of accuracy. Our method reduced the size of VGG-16 by 49× from 552MB to 11.3MB, again with no loss of accuracy. This allows fitting the model into on-chip SRAM cache rather than off-chip DRAM memory. Our compression method also facilitates the use of complex neural networks in mobile applications where application size and download bandwidth are constrained. Benchmarked on CPU, GPU and mobile GPU, compressed network has 3× to 4× layerwise speedup and 3× to 7× better energy efficiency.

Approach/Methodology

Neural networks are both computationally intensive and memory intensive, making them difficult to deploy on embedded systems with limited hardware resources. To address this limitation, we introduce "deep compression", a three stage pipeline: pruning, trained quantization and Huffman coding, that work together to reduce the storage requirement of neural networks by 35× to 49× without affecting their accuracy. Our method first prunes the network by learning only the important connections. Next, we quantize the weights to enforce weight sharing, finally, we apply Huffman coding. After the first two steps we retrain the network to fine tune the remaining connections and the quantized centroids. Pruning, reduces the number of connections by 9× to 13×; Quantization then reduces the number of bits that represent each connection from 32 to 5. On the ImageNet dataset, our method reduced the storage required by AlexNet by 35×, from 240MB to 6.9MB, without loss of accuracy. Our method reduced the size of VGG-16 by 49× from 552MB to 11.3MB, again with no loss of accuracy. This allows fitting the model into on-chip SRAM cache rather than off-chip DRAM memory. Our compression method also facilitates the use of complex neural networks in mobile applications where application size and download bandwidth are constrained. Benchmarked on CPU, GPU and mobile GPU, compressed network has 3× to 4× layerwise speedup and 3× to 7× better energy efficiency.

Results

Neural networks are both computationally intensive and memory intensive, making them difficult to deploy on embedded systems with limited hardware resources. To address this limitation, we introduce "deep compression", a three stage pipeline: pruning, trained quantization and Huffman coding, that work together to reduce the storage requirement of neural networks by 35× to 49× without affecting their accuracy. Our method first prunes the network by learning only the important connections. Next, we quantize the weights to enforce weight sharing, finally, we apply Huffman coding. After the first two steps we retrain the network to fine tune the remaining connections and the quantized centroids. Pruning, reduces the number of connections by 9× to 13×; Quantization then reduces the number of bits that represent each connection from 32 to 5. On the ImageNet dataset, our method reduced the storage required by AlexNet by 35×, from 240MB to 6.9MB, without loss of accuracy. Our method reduced the size of VGG-16 by 49× from 552MB to 11.3MB, again with no loss of accuracy. This allows fitting the model into on-chip SRAM cache rather than off-chip DRAM memory. Our compression method also facilitates the use of complex neural networks in mobile applications where application size and download bandwidth are constrained. Benchmarked on CPU, GPU and mobile GPU, compressed network has 3× to 4× layerwise speedup and 3× to 7× better energy efficiency.

Conclusion/Takeaway

Neural networks are both computationally intensive and memory intensive, making them difficult to deploy on embedded systems with limited hardware resources. To address this limitation, we introduce "deep compression", a three stage pipeline: pruning, trained quantization and Huffman coding, that work together to reduce the storage requirement of neural networks by 35× to 49× without affecting their accuracy. Our method first prunes the network by learning only the important connections. Next, we quantize the weights to enforce weight sharing, finally, we apply Huffman coding. After the first two steps we retrain the network to fine tune the remaining connections and the quantized centroids. Pruning, reduces the number of connections by 9× to 13×; Quantization then reduces the number of bits that represent each connection from 32 to 5. On the ImageNet dataset, our method reduced the storage required by AlexNet by 35×, from 240MB to 6.9MB, without loss of accuracy. Our method reduced the size of VGG-16 by 49× from 552MB to 11.3MB, again with no loss of accuracy. This allows fitting the model into on-chip SRAM cache rather than off-chip DRAM memory. Our compression method also facilitates the use of complex neural networks in mobile applications where application size and download bandwidth are constrained. Benchmarked on CPU, GPU and mobile GPU, compressed network has 3× to 4× layerwise speedup and 3× to 7× better energy efficiency.

Abstract Swap — Next Monday

- Goal: to learn how to write a great abstract
  - Secondary goal: to see what other students' projects are
- Submit your abstracts **by 5:00 PM**
  - So that I can print them out and put them into a slide deck before class
- We will have an in-class feedback activity
  - Like the project idea discussion activity
  - May overflow into Wednesday

Questions?

Project Report Expectations

Formatting

- Report should be at **least four pages**, not including references
- Report should use **ICML 2019 style** or a similar style
    - This is mostly to be fair about length
- Report should be **structured appropriately**
    - For example: abstract, introduction, related work, main results, experiments
    - Correctly formatted references page

Content — Overview

- You should have **implemented a machine learning system**
  - This entails writing some code
  - You should have some **code to submit** along with the report
    - Either as a supplemental file, or as a link to a repository
- You should have used **a technique we discussed in the course**
  - And it should be clear from the report which one you used
- You should have run empirical evaluations of your method
  - **Systems metric**: e.g. throughput, wall-clock time, memory usage
  - **Statistical metric**: e.g. accuracy, F1 score
- Your work should **correspond to the proposal**

Content — Conceptual

- The report should **summarize the problem** you are trying to solve
  - Explain why your approach is a good idea or interesting to study
  - Thesis statement clearly and concisely states the purpose of the report
- The report should fairly acknowledge **previous work**
  - And relate it to what you did
- The report should be **clear and well-written**
  - Avoid grammar/spelling/punctuation issues that make the text difficult to read.
- The report should **demonstrate knowledge/understanding** of the chosen technique beyond what we discussed in class

Content — Technical

- The main section of the report should **explain what you did**
  - And **why** you did it!
- Someone should be able to **reproduce your results** from the report
- The paper should be technically sound
  - Any **claims should be supported** by theoretical analysis or experimental results
- Evaluate both the **strengths and weaknesses** of the work

Content — Experimental

- The experiments should involve a **fair comparison**
  - In terms of systems & statistical performance, among two or more methods
  - Will need **more** than the bare minimum described in the proposal
- The report should **explain the experimental results**
  - Why did this happen? Was it what you expected? What does this tell us?
- The results should be **properly formatted**
  - At least one figure with a title and properly labeled axes
  - Present things graphically whenever possible

Content — Impact

- The report should **discuss the impact** of the results
    - What does this tell us about how we should design systems in the future?

- The report should gesture at possibilities for **future work**

Questions?

Sparsity and Structured Matrices

CS6787 Lecture 13 — Fall 2019

Sparsity Basics

- A sparse matrix has most of its entries zero
  - The fraction of nonzero entries is called the **density**
- One way to make linear algebras operations faster
  - **Why does this help?**
- But, it’s not that simple
  - There are many **pros** to sparse computing for ML systems
  - But there are also a lot of **cons**

With Sparsity, Storage Matters!

- Unlike dense matrices, **many different ways** to store a sparse matrix
  - COO — coordinate list
  - CSR — compressed sparse row
  - CSC — compressed sparse column

- **What are the advantages and disadvantages of these?**

Demo

General rule of thumb for performance

- For **fixed vector dimension**
  - As density decreases, cost of computations **goes down**
  - But only starts being better than dense at around 10% for many operations
  - Can improve this a bit with specialized hardware accelerators
- For **fixed size of data** (measured in bytes)
  - As density decreases, cost of computations **goes up**
  - In the limit of extreme sparsity, you start using techniques from databases
    - Or from graph computation

Where do we find sparsity in ML?

- In **input** training sets
  - Many real-world phenomena are sparse. **Examples?**
- In **models** that we learn
  - Particularly when we use **L1 regression**
  - Also sometimes want to **impose sparsity** on our models a priori
  - Intuition: sparse models **less prone to overfitting**
- In **intermediate values** used during computation
  - For example, the output of a **ReLU** activation function is typically sparse

Two Strategies for Leveraging Sparsity in Data

- Use **sparse linear algebra/sparse computations**
  - Hopefully this will run faster
  - You probably already know about this

- Use an **embedding**
  - Map the sparse input data onto a **lower-dimensional dense feature vector**
  - For example, with random kernel features
  - For example, with the first layer of a deep neural network
  - For example, **word2vec**

Johnson–Lindenstrauss Transform

- One popular **general embedding** you've already seen

- Recall: given **0 < ε < 1**, **m** points in $\mathbf{R}^{\mathbf{D}}$, there is a matrix **A** such that

$$(1-\epsilon)\|x-y\|^2 \leq \|Ax - Ay\|^2 \leq (1+\epsilon)\|x-y\|^2$$

where $A \in \mathbb{R}^{d \times D}$ and $d \approx 8\epsilon^{-2}\log(m)$

- We can use this to project sparse vectors onto a smaller dense space
  - Then use **fast dense arithmetic**

Sparsity on Hardware

- The **CPU usually has the most to gain** from going sparse
  - Because it has large caches that support random access
- But **GPUs can also benefit** from sparse computation
  - For example, NVIDIA has a **cuSPARSE** sparse matrix library
- If sparsity pattern is predictable, we can design **specialized hardware**
  - But I have not seen this used in production systems yet

More Complex Questions

Sparsity: Storage Matters — Episode 2

- Attack of the Clones!
  - **Should we store multiple copies of our sparse thing in different formats?**
- **What precision to use for the indices?**
- **Should we use blocking?**
- **Should we use heterogeneous formats with dense sub-blocks?**
- These questions can affect performance by orders of magnitude!

Questions?

Structured Matrices

A whiteboard talk