0-1-deep-network-structure.out.md

Introduction to Deep Learning

This lecture provides a rapid overview of deep learning, covering the essential concepts and components that constitute a deep network. It aims to help students assess their understanding and identify areas for further study.

What is a Deep Network?

- **Definition**: A deep network is fundamentally a large, differentiable function, often represented as f(x) .
- **Structure**: Composed of layers of simple functions, forming a computation graph. These layers are interconnected in a specific manner to facilitate learning.
- **Learning**: The network learns by adjusting parameters theta using gradient descent.

Types of Layers in Deep Networks

1. **Linear Layers**:
   - **Matrix Multiplication**: Basic form of linear transformation.
   - **Convolution**: A type of matrix multiplication applied over inputs like images.
2. **Non-linear Layers**:
   - **Normalization Layers**: Ensure the network operates on the right scale, aiding in training.
   - **Activation Functions**: Introduce non-linearity, allowing the network to learn complex patterns.
   - **Advanced Layers**: Include attention mechanisms and pooling, which manage information flow and gathering.

Building Blocks of Deep Networks

Popular Blocks

1. **Multi-Layer Perceptron (MLP)**:
   - Consists of alternating linear and activation layers.
   - Known as a universal function approximator but prone to overfitting if too large.
2. **Multi-Head Attention**:
   - Combines linear projections with attention computations to facilitate information exchange between inputs.
3. **Transformer Block**:
   - Combines attention mechanisms with MLPs.
   - Includes residual connections to enhance training.
4. **Convolutional Block**:
   - Replaces linear layers with convolutions.
   - Often includes normalization and residual connections.

Types of Deep Networks

1. **Convolutional Networks**:
   - Efficient for image-like inputs with fixed spatial dimensions.
   - Transform images into abstract representations, often reducing size while increasing channel information.
2. **Transformer Networks**:
   - Popular for handling unordered inputs in a set format, such as text.
   - Utilize positional embeddings to incorporate order and structure in inputs like images or text.

Functionality of Deep Networks

- **Input and Output**: Deep networks process real-valued vectors, essential for training via gradient descent.
- **Handling Categorical Data**:
  - **Inputs**: Convert categorical data into embeddings or one-hot encodings.
  - **Outputs**: Predict probabilities over categories, enabling the modeling of non-real-valued outputs.

Summary

- A deep network is a large, differentiable function, primarily convolutional or transformer-based.
- Inputs and outputs are real values, with categorical data being transformed using embeddings and classifiers.
- Understanding these components and their interactions is crucial for effectively building and training deep learning models.

This overview serves as a foundational guide to the key elements of deep learning, providing the basis for further exploration and study in the field.

0-2-training-deep-networks.out.md

Lecture Notes: Training Deep Networks

Welcome to part two of the introduction to deep networks. In this session, we will focus on how to train deep networks effectively.

Overview

A deep network can be conceptualized as a parametric function f(x) , where the parameters are typically stored within linear layers of this complex, nested function.

Key Components of Training Deep Networks

Training deep networks involves three primary components:

1. **Optimizer and Training Objective (Loss Function)**
2. **Dataset**
3. **Architecture**

Architecture

- Typically involves models like Transformers or Convolutional Neural Networks (CNNs).
- The architecture's weights are what we aim to train, focusing on parameters of matrix multiplications or convolutions.

Dataset

- Comprises collections of data such as images with labels, text, or audio.
- The dataset serves as a benchmark to check the model's performance.

Loss Function

- Determines how well the network predictions align with the actual data.
- Two main types: Supervised vs. Unsupervised learning and Generative vs. Discriminative modeling.
- In most cases, involves comparing network output to data labels.

Types of Loss Functions

1. **Regression Losses** (for continuous labels)
   - L1 Loss
   - L2 Loss (Mean Squared Error)
   - Choice between L1 and L2 is not critical for most applications unless precision is crucial.
2. **Classification Losses** (for discrete labels)
   - Binary Classification: True or False
   - Multi-class Classification: Identifying among multiple categories (e.g., dog, cat, etc.)
3. **Embedding Losses**
   - Used to determine if data pairs match (e.g., text and image matching).
   - Typically reduced to specialized versions of cross-entropy classification losses.

Training Objective

- The objective is to minimize the loss function L .
- The loss function takes as input the network output f_theta(x) and a label.
- We often take the expectation over the entire dataset to evaluate the fit of model parameters to the data.

Gradient Descent

- We use gradient descent to minimize the loss.
- Since the network is a computation graph of simple operations, automated differentiation can compute the gradient.
- This gradient is used by the optimizer to adjust the model's weights.

Optimizer: Adam and AdamW

- Adam and AdamW are widely used optimizers for training deep networks.
- These optimizers track momentum terms (both first and second) and adjust weights accordingly.
- Adam adds parameters to the model, doubling the number of model parameters plus the gradient values.
- This can become problematic for very large models due to increased memory requirements.

Steps to Train Deep Networks

1. **Develop the Architecture**: Use convolutional or transformer-based models.
2. **Select a Dataset**: Gather input-output pairs for training.
3. **Optimize and Fit Weights**: Use loss functions and optimizers to adjust the model weights to fit the dataset.

In summary, training a deep network involves a systematic approach to selecting models, datasets, and utilizing optimizers and loss functions to iteratively improve model performance.

1-1-gpu.out.md

Deep Learning and GPU Architecture

Understanding the hardware architecture that modern deep learning models run on is crucial for optimizing efficiency and performance. This lecture provides an overview of GPU architecture, focusing on its implications for deep learning.

Graphics Processing Unit (GPU)

- **Definition**: GPU stands for Graphics Processing Unit, now often referred to as a General Purpose Processing Unit due to its expanded role in computing.

- **Influence**: GPUs have significantly shaped what is possible in deep learning, influencing both feasible architectures and market demand.

GPU Architecture

High-Level Overview

- **TPU (Tensor Processing Unit)**: A massively parallel processor.
- **Example: H100 GPU**:
  - Contains ~130 streaming processing units.
  - Each unit has 128 cores for float 32 precision.
  - Features 80 GB of RAM, separated by two layers of cache.

Cache and Memory

- **L2 Cache**: Shared between processing units.
- **L1 Cache**: Each streaming processor has additional L1 caches.
- **Shared Memory**: Each streaming processor has about 200 kB of shared memory.

Processing Units

- **Streaming Processor**: Contains multiple cores and shared caches.
- **Warps**: Blocks of processors within a streaming processor.
  - Each warp shares a scheduler, dispatch unit, and set of registers.
  - All cores in a warp execute the same command simultaneously.

Memory Structure

- **Shared Memory**: Fast access but limited in size.
- **Global Memory**: Slower access; includes the main memory separate from processors.

Multi-GPU and Data Centers

- **Nodes**: A single node can contain multiple GPUs.
- **Data Centers**: Consist of multiple nodes with thousands of GPUs.
  - Power consumption for large data centers can approach 0.42 gigawatts.

Computational Considerations

Constraints

- **Memory Bandwidth**: The primary bottleneck in GPU computation.
- **Computation vs. Memory Access**: GPUs can perform a large number of computations, but accessing memory is slow in comparison.

Programming with CUDA

- **CUDA Programming**: Used to write programs for GPUs.
- **Kernel Functions**: These are the core of CUDA programs, executed by all GPU threads.
- **Thread and Block IDs**: Used to identify the specific core and warp executing the function.

Practical Example: Max Pooling

Problem Statement

- Compute the maximum value in a sliding window across a sequence of numbers, analogous to a max pooling operation in deep learning.

Python Implementation

- **Brute Force**: O(n * w) complexity where n is sequence length and w is window size.
- **Heap Optimization**: Reduces complexity to O(n * log(w)) by using a heap data structure.

GPU Implementation

- **CUDA Kernel**: Parallelizes the computation by distributing across GPU cores.
- **Memory Optimization**: Reduces memory accesses by using shared memory within streaming processors.

Performance

- GPU implementations significantly outperform Python versions due to parallelization and efficient memory access.

Key Takeaways

1. **Memory Access is Key**: Efficient memory management is crucial for optimizing GPU performance.
2. **Parallelization**: GPUs excel at parallel computation but require careful structuring of operations to maximize efficiency.
3. **Physical Limits**: Current technology is nearing its peak in terms of power consumption and scaling capabilities.

Understanding these principles allows developers to design more efficient deep learning models and effectively utilize GPU resources.

1-2-training-models.out.md

Lecture Notes: Evolution and Challenges in Training Deep Networks

Overview

This lecture discusses the evolution of training deep networks, the role of hardware advancements, the transition from CPU to GPU, and the ongoing challenges in scaling models. It also covers the shift in focus from algorithmic innovations to systems engineering and memory management in modern deep learning.

1. Historical Context

Pre-2012: CPU Era

- **Hardware**: Machine learning models primarily used CPUs.
- **Algorithms**: Focus on efficient algorithms due to limited processing power.
- **Optimization**: Relied heavily on theories from convex optimization.
- **Feature Engineering**: Required extensive human engineering, especially in areas like computer vision.

2012 Onwards: The GPU Revolution

- **Breakthrough**: Deep networks trained on GPUs began to surpass traditional models.
- **Shift**: Moved from hand-engineered features to data-driven approaches leveraging GPU compute.
- **Data**: Explosion of large datasets, especially in computer vision.
- **Optimization Algorithms**: Introduction of Adam and improved versions of stochastic gradient descent.

2. Phases of Deep Learning Evolution

2012-2018: GPU-Based Deep Networks

- **Compute Shift**: Transition from CPU to GPU-based computations.
- **Feature Learning**: Models began learning features directly from data.
- **Optimization Improvements**: Development and maturation of optimization algorithms.
- **Limitations**: Primarily GPU compute and innovative ideas.

2019-2022: Scaling and Complexity

- **Larger Datasets**: Utilization of vast internet-crawled datasets.
- **Multi-GPU Models**: Emergence of models requiring multiple GPUs.
- **Attention Models**: Rise due to their scalability with data and compute.
- **New Limitations**: Memory constraints became prominent as models grew.

2023 and Beyond: Multi-Node Systems

- **Massive Models**: Training models with 8 to 400 billion parameters.
- **Infrastructure**: Necessity of multi-node systems, sometimes involving thousands of GPUs.
- **Data**: Shift from task-specific datasets to internet-scale data.
- **Architecture Stability**: Few fundamental algorithmic breakthroughs; focus on architectural tweaks and scaling.

3. Current Challenges and Focus

Systems and Memory Management

- **Resource Limitation**: GPU memory is a significant bottleneck.
- **Cost**: Training large models is expensive, driven by hardware and memory requirements.
- **Scaling Benefits**: Larger models consistently perform better.
- **Memory Usage**: Weights, gradients, and optimizer states consume significant memory.

Innovations in Training Efficiency

- **Memory Optimization Techniques**:
  - Distributed Training
  - Redundancy Reduction
  - Low-Rank Approximation
  - Quantization
  - Checkpointing
  - Flash Attention
- **Efficiency Goal**: Reduce memory usage from 16x the number of parameters to potentially below the memory used by the model's weights in floating-point representation.

4. Conclusion

The landscape of deep learning has shifted dramatically from algorithmic innovation to systems engineering. The primary challenges today are centered around efficiently managing resources to train ever-larger models. The lecture concludes with a focus on memory efficiency techniques that enable the training of massive models on current hardware architectures.

1-3-precision-training.out.md

Lecture Notes: Mixed Precision Training

Introduction to Mixed Precision Training

- **Objective**: Reduce memory usage and speed up model training in deep learning.
- **Memory Requirement Without Optimizations**:
  - For a model with `n` parameters using Adam optimizer:
    - Memory needed: `16n` bytes.
  - Example: A model with 8 billion parameters requires a GPU with unprecedented memory capacity.

Understanding Floating Point Values

- **Floating Point Components**:
  - **Sign**: 1 bit indicating positive or negative.
  - **Exponent**: Determines the scale of the number.
  - **Fraction**: Determines precision within a given exponent range.
- **Range and Precision**:
  - Floating point numbers can represent a large range (e.g., from approximately `3e38` to `1.7e-38`).
  - Allows representation of both very large and very small numbers with high precision.

Application in Deep Learning

- **Weight and Gradient Representation**:
  - Weights and gradients in neural networks often fit comfortably within `Float32` precision.
  - Most values are distributed around zero, with few outliers.

Exploring Smaller Floating Point Representations

- **Float16**:
  - Composed of 5 bits for the exponent and 10 bits for the fraction.
  - Provides sufficient precision for many deep learning tasks but may struggle with extreme values.
- **Bfloat16**:
  - Developed by Google Brain for deep learning.
  - Allows a larger range due to a larger exponent, but with less precision in the fraction.
  - Preferred for training deep networks to reduce memory usage and computation time.

Advantages of Using Bfloat16

- **Memory and Speed Benefits**:
  - Reduces memory usage by half compared to `Float32`.
  - Allows training of larger models on existing GPUs.
  - Potential for faster computation due to smaller data size.
- **Matrix Multiplication on GPUs**:
  - Uses block form to reduce memory reads from main memory.
  - More `bfloat16` values fit into shared memory, allowing larger blocks and faster computation.

Implementing Mixed Precision Training

- **Mixed Precision Training Techniques**:
  - Original method: Keep weights in `Float32`, perform forward and backward passes in `Float16`.
  - Modern approach: Train directly with `bfloat16`, reducing memory and communication overhead.
- **Practical Considerations**:
  - Use of gradient scalers to manage underflows and maintain stability.
  - Certain operations (e.g., normalizations) may require full precision to avoid issues.

Conclusion

- **Benefits of Mixed Precision**:
  - Reduces memory requirements and speeds up computation.
  - Particularly useful for large models constrained by memory limits.
- **Future Directions**:
  - Further optimizations and techniques needed to fully leverage mixed precision in extremely large models without compromising accuracy or stability.

By adopting mixed precision training, deep learning practitioners can effectively handle larger models on existing hardware, enabling more complex and capable AI systems.

2-1-distributed-training.out.md

Lecture Notes: Memory Requirements and Parallelism in Deep Learning

Memory Requirements of Deep Networks

- **Memory Consumption**:
  - Large deep networks have immense memory requirements due to:
    - Storing four copies of weights: original weights, gradients, first and second moments.
    - In floating-point precision, this amounts to about 16n bytes, excluding intermediate computation and activation memory.
  - Mixed precision can reduce the requirement to about 8n, but this is the limit.
- **Scale of Modern Models**:
  - Models can have 400 billion parameters or more.
  - Large language models in academia range from 8 billion to 70 billion parameters.

Challenges of Training Large Models

- **Resource Limitations**:
  - Even smaller models like an 8 billion parameter model cannot fit on a single GPU.
  - Solution: Use multiple GPUs to distribute the workload and memory.

Parallelism Strategies

Data Parallelism

- **Concept**:
  - Keep one copy of the model on each GPU.
  - Split the dataset into chunks processed independently by each GPU.
  - Synchronize gradients post-processing.
- **Advantages**:
  - Allows parallel data processing.
  - Close to linear speedup with a small number of GPUs.
- **Challenges**:
  - Initial versions required synchronization with a central server, slowing down the process.
  - Communication overhead due to gradient and weight transfers.
  - Synchronization could be slow due to networking and uneven computation among GPUs.
- **Modern Data Parallelism**:
  - Synchronization is achieved using "all-reduce" operations, reducing communication overhead.
  - No central server; each GPU has its optimizer.
  - Gradients are synchronized before optimizer steps.

Model Parallelism

- **Pipeline Parallelism**:
  - Split the model across multiple GPUs in a pipeline.
  - Forward and backward passes are processed sequentially across GPUs.
  - Introduction of micro-batches can reduce idle time ("bubbles").
- **Tensor Parallelism**:
  - Split each layer across multiple GPUs.
  - Less common due to complexity in implementation.
- **Memory Efficiency**:
  - Allows models to be split across GPUs, reducing per-GPU memory requirements.
  - Facilitates training of very large models.

Comparison of Parallelism Methods

- **Data Parallelism**:
  - High memory use, high throughput.
  - Requires the whole model to fit on each GPU.
- **Model Parallelism**:
  - Lower memory use, potential lower throughput due to bubbles.
  - Splits the model across GPUs, reducing per-GPU memory requirements.

Advanced Techniques for Large Models

- Techniques are being developed to combine low memory consumption with high throughput.
- Used by companies like OpenAI for training massive models like GPT.

Key Takeaways

- **Memory Management**: Efficient memory use is critical in training large models.
- **Parallelism**: Both data and model parallelism strategies have their use cases and limitations.
- **Scalability**: Modern techniques strive to balance memory efficiency with computational speed, enabling the training of extremely large models.

2-2-zero-redundancy.out.md

Lecture Notes: Advanced Data Parallelism and Multi-GPU Training

Overview

This lecture focuses on advanced techniques for data parallelism and multi-GPU training, particularly in settings with large datasets and a significant number of GPUs. The goal is to train models efficiently across thousands of GPUs by combining data parallelism and model parallelism, optimizing memory usage, and employing synchronization strategies.

Key Concepts

Basic Parallelism Recap

1. **Data Parallelism**: Distributing the data across multiple GPUs, each with a copy of the model.
2. **Model Parallelism**: Distributing the model components across multiple GPUs.

Advanced Techniques

- **Zero Redundancy Optimizers (ZeRO)**: These techniques aim to reduce memory consumption by partitioning model states (e.g., optimizer states, gradients, weights) across multiple GPUs.

ZeRO Approaches

ZeRO-1: Optimizer State Partitioning

- **Insight**: Optimizer state (first and second momentum) is only required when stepping the optimizer.
- **Technique**:
  - Store optimizer states across different GPUs instead of duplicating them.
  - Perform forward and backward passes without needing optimizer state.
  - Use a `reduce scatter` operation to gather and sum gradients needed for each GPU's optimizer state.

ZeRO-2: Gradient Partitioning

- **Additional Step**: Partition gradients across GPUs.
- **Technique**:
  - Compute gradients locally, then use `all-reduce` to sum and distribute gradients across GPUs.
  - Only keep gradients relevant to each GPU, discarding others.

ZeRO-3: Full Model Sharding

- **Further Optimization**: Partition weights and optimizer states across GPUs.
- **Technique**:
  - Synchronize weights needed for each forward and backward pass.
  - Communicate and gather model shards dynamically, leading to reduced memory but increased synchronization overhead.

Fully Sharded Data Parallelism

- **Improvement over ZeRO-3**: Implements efficient communication strategies to reduce synchronization overhead.
- **Usage**: Employed in training large models, such as Meta's LLaMA models.

Hybrid Sharding

- **Concept**: Combines fully sharded data parallelism within groups of GPUs with regular data parallelism across groups.
- **Benefit**: Reduces long-distance communication, improving efficiency in large clusters.

Memory Efficiency

- **Goal**: Reduce memory footprint from entire model state to fractions proportional to GPU count.
- **Outcome**: Enables training significantly larger models (50x reduction in required memory).

Challenges and Trade-offs

- **Synchronization Overhead**: Increased communication can become a bottleneck.
- **Implementation Complexity**: Different versions of ZeRO have varying levels of efficiency and complexity.

Conclusion

- **GPU-Rich vs. GPU-Poor**: Strategies differ based on resource availability. GPU-rich environments can leverage extensive parallelism, while GPU-poor environments require more innovative approaches.
- **Next Steps**: Explore techniques for environments with limited GPU resources, focusing on optimizing with fewer GPUs.

This lecture provides a comprehensive understanding of advanced data parallelism techniques, emphasizing the balance between memory usage and synchronization overhead in large-scale model training.

2-3-low-rank-adapters.out.md

Lecture Notes: Low Rank Adapters in Machine Learning

Introduction to Low Rank Adapters

- **Concept Overview**: Low rank adapters are a technique for optimizing fewer weights in large neural networks, allowing for efficient training without needing to modify all weights.
- **Purpose**: Designed to reduce memory usage and computational cost by focusing on a subset of weights, especially useful when resources (like GPUs) are limited.

Memory Usage in Training Large Models

- **Memory Consumption**:
  - **Weights, Gradients, and Momentum**: These are dependent on the number of parameters optimized.
  - **Activations**: Intermediate outputs required for backpropagation.
- **Challenges**: Training large models without optimization can demand substantial memory, posing challenges for those with limited computational resources.

Reducing Memory Usage

- **Primary Focus**: The session focuses on reducing memory usage associated with gradients and momentum, not model weights (which is covered in the next segment).

Strategies for Memory Reduction

1. **Training Fewer Parameters**:
   - **Frozen Parameters**: Freeze most parameters and only optimize a subset, reducing the need for gradients and momentum storage.
   - **Fine-Tuning Classifiers with Frozen Backbone**: Common practice where only the last layer(s) are optimized for new outputs while the rest of the network remains unchanged.
2. **Fine-Tuning Inputs and Outputs**:
   - **Adapters for Input Layers**: Adjust how inputs are processed, which can be more expressive and involves some backpropagation through the entire network.
   - **Applications**: Particularly useful for large language models and vision-language models.
3. **Training Intermediate Layers**:
   - **Challenges with Random Subset Selection**: Randomly selecting a subset of weights to optimize usually doesn't work due to the interdependencies of weights.
   - **Low Rank Adapters**: Introduce low rank matrices to adjust weights without extensive memory usage.

Low Rank Adapters: Implementation

- **Matrix Representation**:
  - Original weight matrices are split into two smaller matrices (A and B) with reduced ranks.
  - The rank parameter (r) is significantly smaller than the original dimensions, reducing memory while still enabling meaningful updates.
- **Initialization**:
  - Matrix A initialized randomly; Matrix B initialized to zero to prevent initial noisy updates.
  - This setup ensures that gradients can be computed effectively from the start.

Practical Implementation

- **Mathematical Formulation**:
  - Incorporate low rank matrices into linear layers, adjusting the output using a scaling factor to maintain consistent training dynamics.
- **Coding with PyTorch**:
  - Create a new linear layer class (`LoraLinear`) that inherits from the standard linear layer but includes additional linear transformations for the low rank adapters.
  - Replace standard linear layers in models with `LoraLinear` to integrate low rank adapters seamlessly.

Advantages and Limitations

- **Advantages**:
  - Significant reduction in memory usage for gradients and momentum terms.
  - Enables training larger models with constrained resources.
  - Retains the ability to adapt and learn new computations with previous pre-trained knowledge.
- **Limitations**:
  - Requires access to a pre-trained model for initial weights.
  - Memory is still required to store the original model weights.

Conclusion

Low rank adapters offer an efficient method for optimizing large neural networks by focusing on specific subsets of parameters. By reducing memory requirements for gradients and momentum, they allow for scalable training processes that are particularly beneficial in environments with restricted computational resources. However, reliance on a pre-trained model is necessary, highlighting the importance of accessible model weights.

2-4-quantization.out.md

Lecture Notes: Quantization in Machine Learning

Overview

Quantization is a method used in machine learning to reduce the size of model weights, primarily focusing on decreasing memory usage and computational requirements without significantly sacrificing model performance. This lecture covers various aspects and techniques of quantization, including floating-point formats, integer quantization, and optimizations for training.

Key Concepts

1. **Memory Challenges with Large Models:**
   - Large models require significant memory for storing weights, gradients, and momenta.
   - For inference alone, models like Llama (70 billion parameters) can demand 2 to 4 TPUs.
   - GPUs often lack sufficient memory to handle such large models, necessitating innovative solutions.
2. **Floating-Point Formats:**
   - `Float32` vs. `BFloat16`: Transitioning from 32-bit to 16-bit floats reduces memory usage.
   - Precision and truncation issues arise in `BFloat16`, important for representing fine-grained differences.
   - `Float8` and `Float4` are discussed but deemed impractical due to significant precision loss.
3. **Integer Quantization:**
   - **Scale and Affine Quantization:**
     - Scale quantization maps weights to a range of integers using a maximum value.
     - Affine quantization uses both minimum and maximum values, offering finer granularity.
   - **Block-Wise Quantization:**

    - Addresses the issue of outlier weights by computing quantization per block, reducing the impact of extreme values.
4. **Advanced Techniques:**
    - **Double Quantization:** Quantizes the quantization constants themselves, optimizing memory usage.
    - **8-Bit Adam Optimizer:** Reduces memory requirements by quantizing optimizer states.
    - **Stochastic Rounding:** Aids in reducing truncation errors during training by probabilistically rounding values.
5. **Practical Considerations:**
    - **Training vs. Inference:**
        - Quantization is more commonly applied post-training for inference to save memory.
        - Training with quantized weights requires careful handling to maintain performance.
    - **Lower Bound of Quantization:**
        - Theoretical limits suggest large language models can store about two bits of information per parameter under ideal conditions.
        - In practice, models are often quantized to 4-6 bits without significant performance loss.

Applications

- **Memory Reduction:** Quantization allows storing large models in reduced memory spaces, e.g., storing a 70 billion parameter model in approximately 35 GB of memory.
- **Inference Optimization:** Enables running inference on more modest hardware without compromising model performance significantly.
- **Training Efficiency:** Techniques like 8-Bit Adam and stochastic rounding facilitate training with reduced precision, maintaining model accuracy while optimizing resource use.

Conclusion

Quantization serves as a crucial tool in modern machine learning, enabling the deployment of large models on hardware with limited memory resources. By leveraging various quantization techniques and understanding their implications on precision and performance, machine learning practitioners can efficiently manage and deploy large-scale models. The lecture sets the stage for exploring how these models can be further adapted and trained using quantized parameters in future sessions.

3-1-quantized-low-rank.out.md

Lecture Notes on Memory-Efficient Training of Large Models

Introduction

- Discussed strategies for training large models, particularly in environments with abundant GPU resources.
- Explored methods to train smaller models, including:
    - Lora adapters
    - Low-rank adapters
    - Quantization techniques for reducing model size.

Model Distribution and Quantization

- Large models are distributed across many GPUs to handle the size.
- Quantization helps to reduce the memory footprint:
    - Original model weights require 16 bytes per parameter, which is unmanageable for billions of parameters.
    - Inference can reduce this to about half a byte per weight, using integer values between 0 and 15 with non-linear quantization.

  - This reduction is essential but has a lower limit; going below half a byte per parameter risks losing critical information.

Lora Adapters

- Lora adapters allow training of models without altering all parameters:
  - Only 1-5% of additional parameters are trained.
  - Original model weights remain unchanged.
  - Gradient, momentum, and second momentum are required only for the Lora parameters.

QLora: An Advanced Approach

- Combines quantization and Lora adapters:
  - Quantizes the model to 4 bits (or adjustable bit levels).
  - Applies Lora adapters to the quantized model.
- Requirements:
  - Needs a pre-trained model (either already quantized or to be quantized by the user).
  - The adapter itself cannot be trained in low precision, but it's small enough to be manageable.

Mechanism of QLora

- Operations during training remain in floating point:
  - Quantized weights are temporarily converted back to floating point for operations.
  - Lora adapters and gradients are always in floating point.
- Memory efficiency:
  - Only need to store quantized weights and a small Lora adapter.
  - Memory footprint is significantly reduced compared to storing full precision weights.

Memory Calculation

- Memory required for storing the model:
  - Half the memory of the original parameters (n) in quantized form.
  - Additional memory for Lora adapter parameters (m), which is a small fraction (1-5%) of n.

Limitations and Considerations

- QLora is limited to fine-tuning:
  - Not suitable for pre-training as it relies on a pre-trained model.
  - Effectiveness of Lora is task-dependent and may require experimentation.

Future Directions

- Exploration of variants or twists to address limitations of QLora.
- Developments in reducing memory requirements further and improving applicability of Lora adapters.

Conclusion

- QLora represents a significant advancement in memory-efficient model training.
- Offers potential for training large models without needing proportional memory resources.
- Future advancements may address current limitations and expand applicability.

3-2-low-rank-projections.out.md

Lecture Notes on Efficient Training and Memory Optimization for Large Models

Overview

This lecture discusses the evolution of training large models, focusing on optimizing memory usage and computational efficiency. It covers techniques from basic setups to advanced methods like QLoRA and Galore that significantly reduce memory requirements while maintaining performance.

Key Concepts

Basic Training Setup

1. **Vanilla Training Setup**:
   - Utilizes 16N bytes of memory, where N is the number of parameters.
2. **Memory Constraints**:
   - The goal is to train and run inference on memory-limited setups by minimizing memory usage.

Memory Optimization Techniques

1. **LoRA and Quantized LoRA**:
   - Reduces memory requirements to about half N bytes for original weights.
   - Adapters use only 1 to 5% of the original model's parameters.
   - Cannot compress below two bits or a quarter of a byte due to information theoretic limits.
2. **QLoRA's Disadvantages**:
   - Fine-tuning only; depends on original weights.
   - Task-dependent, requiring different adapter sizes for different tasks.

Understanding Backpropagation and Memory Usage

1. **Forward and Backward Passes**:
   - Forward pass computes transformations and stores activations.
   - Backward pass requires stored activations to compute gradients.
   - Linear layers: Gradient is the outer product of backpropagated data and original input.
   - Non-linear layers: Gradients depend on the original input to the layer.
2. **Memory Management**:
   - PyTorch deletes activations after computing gradients to free memory.
   - Memory usage peaks before calling `optimizer.step()` due to storage needs for weights, gradients, and momentum terms.

Advanced Memory Optimization Techniques

1. **Single GPU Optimization**:
   - Call `optimizer.step()` after each gradient computation to free up memory immediately.
   - Reduces peak memory usage by not retaining the gradient.
2. **Galore**:
   - Assumes gradients are low-rank and optimizes in a lower-dimensional subspace.
   - Projects gradients to a low-rank subspace, performs updates, and projects back.
   - This reduces the memory footprint of the optimizer state.
3. **QGalore**:
   - Further reduces memory by using quantized weights and projections.
   - Introduces stochastic rounding to handle small gradients.

Challenges and Limitations

1. **Single GPU Limitation**:

    - Techniques like Galore are limited to single GPU setups, lacking support for distributed training.
2. **Gradient Accumulation**:
    - Incompatible with approaches that require immediate optimizer steps.
3. **Stability and Performance**:
    - Methods like Galore can be unstable and slower due to SVD computations.
4. **Potential Developments**:
    - Future algorithms may blend techniques from QLoRA and Galore, offering quantized weights with efficient state management for broader applicability.

Conclusion

The lecture highlights the importance of memory-efficient training strategies for large models. It emphasizes ongoing research and development in this area, suggesting that future advancements will likely provide more effective solutions for training large models on constrained hardware.

3-3-checkpointing.out.md

Memory Optimization in Neural Network Training

Introduction

This lecture focuses on optimizing memory usage during neural network training, specifically targeting optimizers and activations. The discussion revolves around strategies to reduce memory requirements proportional to the number of parameters in a model, excluding activations.

Key Concepts

Memory Requirements

- **Optimizers**: The memory required is often proportional to the number of parameters in a model.
- **Activations**: Memory usage is discussed without counting activations.

Gradient Computation

- Gradients in linear layers depend on two inputs: `x` (original input) and `y` (backpropagated output).
- Non-linear layers' gradients inherently depend on the input `x`.

Forward and Backward Passes

- **Forward Pass**: Requires storing the input for each layer to use later in backpropagation.
- **Backward Pass**: Can be memory-intensive due to the need to store activations.

Memory Efficient Strategies

Buffer Reuse

- Use two buffers, `A` and `B`, to store outputs. Reuse these buffers to minimize additional memory requirements.
- PyTorch manages memory efficiently using `torch.no_grad`, which does not store activations for backpropagation.

Activation Checkpointing

- **Recompute Activations**: Instead of storing activations, recompute them during the backward pass.
- **Efficiency Trade-off**: While memory-efficient, it requires more computation, roughly quadratic in complexity concerning the number of layers.

- **Egg Drop Analogy**: Similar to the egg drop problem, efficient recomputation strategies can reduce computation from quadratic to manageable levels.

Implementation of Checkpointing

- **Torch Utils Checkpoint**: Wrap critical parts of the model with checkpointing utilities.
- Ensure consistency in layers with randomness to maintain gradient accuracy.

Offloading to CPU

- **Unified Memory**: CUDA unified memory can be used to offload activations to the CPU, preventing GPU memory overflow.
- Useful for handling varying batch sizes in language models, avoiding crashes due to unexpected large memory requirements.

Conclusion

- We can train models with optimized memory usage by:
  - Reducing memory for parameters and activations.
  - Accepting the computational cost of additional forward passes.
- **Next Steps**: Explore strategies for managing memory required for individual layer computations.

Further Considerations

- **Randomness Control**: Ensure deterministic outputs in layers with inherent randomness during recomputation.
- **Model Wrapping**: Activation checkpointing requires integration at the model level, not just optimizer adjustments.

By applying these strategies, we can significantly improve the memory efficiency of training deep learning models, allowing for more extensive and complex models to be trained on limited hardware resources.

3-4-flashattention.out.md

Lecture Notes: Memory Management and Optimization in Neural Networks

Overview

In this lecture, we covered various strategies to manage and optimize memory usage in neural networks, focusing on handling intermediate and scratch memory required during computations. We explored specific techniques for optimizing memory usage in attention mechanisms and strategies for efficiently implementing operations on GPUs.

Key Topics

Memory Management Challenges

1. **Memory for Weights and Activations:**
   - Initially, memory constraints related to weights and activations were significant, but solutions have been developed to manage these effectively.
2. **Intermediate/Scratch Memory:**
   - This is the temporary memory required for computations. Optimizing this memory is crucial for efficient neural network operation, particularly on GPUs.

Solutions for Memory Optimization

1. **Custom Kernels:**
   - Writing customized CUDA or Triton kernels to minimize memory usage during forward and backward passes.
2. **Flash Attention:**
   - A technique to optimize memory usage in attention mechanisms by avoiding the storage of large intermediate matrices in main memory, instead computing values on-the-fly.

Attention Mechanism Optimization

1. **Operation Breakdown:**
   - Attention involves projecting inputs into keys, values, and queries, computing weights, applying softmax, and aggregating outputs from multiple heads.
   - The memory-intensive part is often the matrix of weights, with dimensions based on sequence length.
2. **Flash Attention Technique:**
   - Flash attention avoids storing the entire matrix of weights and the softmax output, computing them as needed and storing only in fast GPU memory (SRAM).
   - This approach requires careful scheduling and memory management but offers significant memory savings.
3. **Numerical Stability:**
   - To maintain stability, operations like softmax should be computed with care, especially in terms of normalization.

Advanced Flash Attention Techniques

1. **Flash Attention 2 and 3:**
   - These versions improve computational efficiency and GPU utilization.
   - They adjust the order of operations and leverage specialized GPU hardware (tensor cores) for faster matrix multiplications.
   - Flash Attention 3 is tailored for the H100 GPU, providing further optimizations.

Other Memory-Saving Techniques

1. **Operation Fusion:**
   - Fusing operations like matrix multiplication and non-linear layers can save memory and increase speed.
2. **Chunking:**
   - Dividing computations into chunks to reduce memory requirements, useful for large operations.
3. **Torch Compile:**
   - A tool for automatically optimizing PyTorch models for specific hardware, potentially fusing operations and improving efficiency.

Conclusion

- **Memory Usage in Networks:**
  - Weights, gradients, activations, and intermediate operations consume memory.
  - Solutions include quantization, checkpointing, specialized implementations, and tools like Torch Compile.
- **Future of Optimization:**
  - While human-engineered optimizations are still crucial, tools like Torch Compile are becoming more sophisticated and may eventually rival manual optimizations.

These strategies and tools collectively help in managing memory efficiently while training large-scale neural networks, enabling more complex models to be trained on hardware with limited resources.

4-1-open-source-infrastructure.out.md

Lecture Notes: Open Source Infrastructure for Model Training

Overview

In this lecture, we explored various open-source infrastructure and techniques available for large-scale and small-scale model training using PyTorch. We covered checkpoints, memory management, data parallelism, distributed data parallelism, fully sharded data parallelism, and PyTorch Lightning.

Key Topics

Checkpointing

1. **Purpose**: Checkpointing helps manage memory usage during training by storing intermediate results and freeing memory.
2. **Core Function**: `torch.utils.checkpoint.checkpoint`
3. **Usage**:
   - Wraps a module to track checkpointing.
   - Handles randomness with `preserve_random_state` flag.
   - Ensures consistent gradient estimates even with random modules.

Memory Management

1. **Memory Consumption Analysis**:
   - Analyzed how memory is allocated and peaks during forward and backward passes.
   - Demonstrated efficient memory usage with forward/backward and optimizer steps.
2. **Impact of Batch Size**:
   - Increasing batch size increases memory usage especially for activations.
   - Checkpointing can reduce memory footprint by recomputing activations during backward passes.

Data Parallelism

1. **Purpose**: Run model training across multiple GPUs to speed up forward and backward passes.
2. **Implementation**: `torch.nn.DataParallel`
   - Wrap the model in `DataParallel` to distribute computations across GPUs.
   - Handles input splitting and gradient aggregation automatically.
3. **Limitations**:
   - Only suitable for up to 8 GPUs per node.
   - Beyond one node, requires distributed data parallelism.

Distributed Data Parallelism

1. **Purpose**: Scale model training across multiple nodes.
2. **Implementation**: `torch.nn.parallel.DistributedDataParallel`
   - Requires initializing a process group with `init_process_group`.
   - Uses `torchrun` to execute scripts in parallel for each GPU.
   - More involved setup compared to simple data parallelism.
3. **Considerations**:
   - Access the original model through `model.module`.
   - Manage logging and model saving carefully to avoid duplication across nodes.

Fully Sharded Data Parallel (FSDP)

1. **Purpose**: Extremely memory-efficient training across GPUs.

2. **Implementation**: PyTorch's FSDP
    - Shards model weights across GPUs to minimize memory usage.
    - Handles weight and gradient sharing efficiently.
3. **Benefits**: Allows larger batch sizes and models by reducing memory footprint significantly.

DeepSpeed

1. **Purpose**: Alternate implementation of FSDP with additional features.
2. **Integration**:
    - Uses `deepspeed.initialize` for model and optimizer setup.
    - Configuration-driven approach to manage optimization, checkpointing, and precision.

PyTorch Lightning

1. **Purpose**: Simplifies training code by automating common tasks like logging and multi-GPU handling.
2. **Structure**:
    - Define a `LightningModule` for model and training steps.
    - Use `Trainer` for managing training loops and strategies.
3. **Advantages**:
    - Easy setup for distributed training and logging.
    - Abstracts complex setup, making it beginner-friendly.
4. **Drawbacks**:
    - Limited flexibility for custom setups.
    - API changes can cause compatibility issues.

Conclusion

This lecture provided a comprehensive overview of tools and techniques for optimizing model training on multiple GPUs. It emphasized the importance of understanding memory management and different parallelism strategies to efficiently utilize computational resources. In practical applications, especially with limited resources, techniques like checkpointing and PyTorch Lightning can significantly enhance training efficiency.