CSE-234: Data Systems for Machine Learning, Winter 2025

8: Memory

Lecturer: Hao Zhang

Scribe: Aaryan Gupta, Alex Oshima, Akshit Agarwal, Bhavik Chandna, Mathew Luo, Manoj Gayala, Nikhil Paleti, Raghav Kachroo, Rohith Vutukuru, Ruobing Han, Shankara Narayanan Venkateswara Raju

1 The Big Picture & Learning Goals

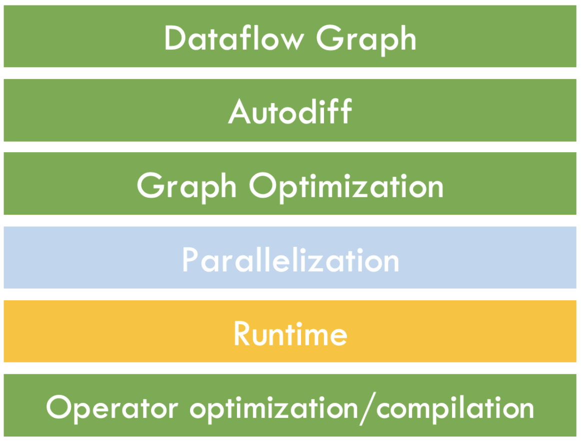


Figure 1: Big Picture Diagram

Overall, we have been taking a top-down approach to understand how we move from writing machine learning code to optimization. This process involves frameworks creating a dataflow graph, followed by autodifferentiation and graph optimization, parallelization and runtime, and finally, operator optimization/compilation.

Today's lecture focused on memory and the optimizations that can address the memory needs of modern machine learning models. Large training sets and other massive data inputs require significant amounts of memory, not to mention the memory needed to run and train the models.

1.1 Learning Goals

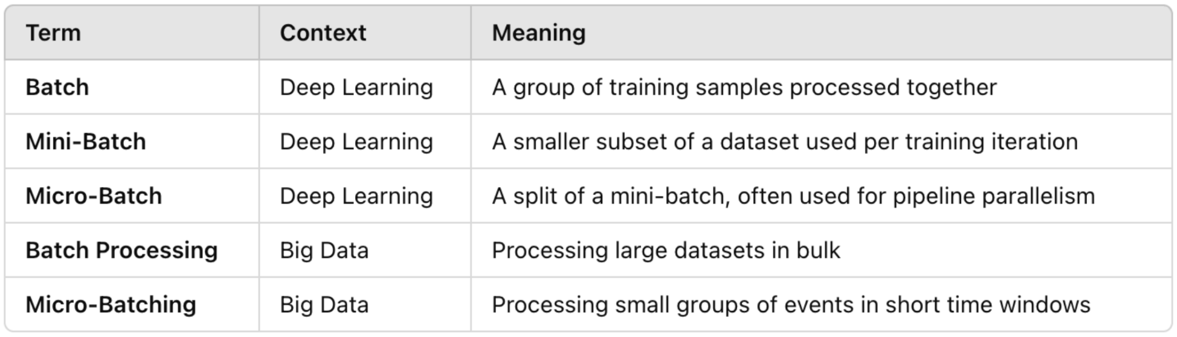
- Examine memory and scheduling strategies:
  - Checkpointing, rematerialization
  - Swapping
- Memory and computation:

- Quantization
- Mixed precision

2 Batch Processing Terminology

A fundamental part of training in machine learning is gradient descent. The original gradient descent algorithm calculates the loss across all data points, which can be very memory-intensive. This is where the concept of batches comes into play. Instead of loading all training data points at once and calculating the loss across all of them, only a subset of data points, known as a batch, is used.

However, the term "batch" is used in multiple fields.



Term	Context	Meaning
Batch	Deep Learning	A group of training samples processed together
Mini-Batch	Deep Learning	A smaller subset of a dataset used per training iteration
Micro-Batch	Deep Learning	A split of a mini-batch, often used for pipeline parallelism
Batch Processing	Big Data	Processing large datasets in bulk
Micro-Batching	Big Data	Processing small groups of events in short time windows

Figure 2: Terminology: Batches

In deep learning, the terms batch, mini-batch, and micro-batch refer to different portions of the original training data being actively used:

- **Batch**: A group of training samples from the larger training dataset that are processed together.
- **Mini-batch**: A smaller subset of a batch, typically used in an iteration.
- **Micro-batch**: A further subdivision of a batch, used in pipeline parallelism.

In the context of big data, batch processing refers to the processing of a static subset of data, as opposed to an incoming stream of live data. Micro-batching, on the other hand, involves processing smaller groups of events within short time windows.

3 Memory and Scheduling

3.1 Introduction - What to care about wrt. memory?

Our computer stores memory in the memory hierarchy. Recall the Figure 3 below. At higher layers like registers, global memory, cache etc., the memory is much faster, but its more expensive and has limited space. But at lower layers, it is slower and has more space and less costly.

We know that most of the Deep Learning Computations happen on GPUs. So, most of the time it happens in Global memory(**HBM**), and sometimes the data is also put on RAM. The important challenge here is that, we have very limited Global memory, but we have to train our Large Language Models on that.

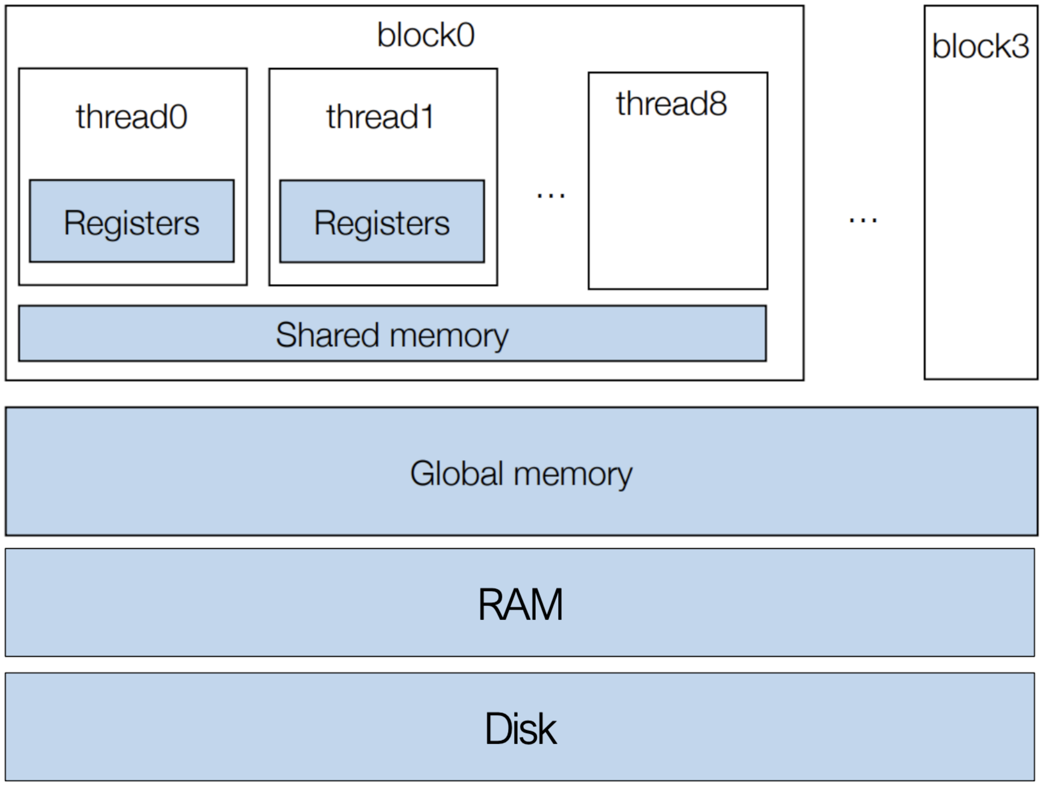


Figure 3: Data Movement Path

3.1.1 Goals on Memory

Our goal is that our peak memory during the execution is less than the available memory. In other words, unlike with the time of the compute, we need not try to minimize the memory usage or the peak memory utilized, but just need to make sure that the peak memory is less than the available memory.

$$peakMemory < availableMemory$$

Now the challenge is how to determine the peak memory. For that, we need to know what exactly consumes the memory when we train models.

3.1.2 Source of Memory Consumption

When training models, the things that needed to be stored are:

- **Model Weights** - for obvious reasons

- **Intermediate activation values** - because in the backward pass, we need to derive the gradients using these.

- **Optimizer States** - because we do not always update weights by just gradients directly, but we do some computations on gradients, like applying Adam's Optimizers etc. and then update the weights.

Now that we know the sources of memory consumption, we need to know how to determine the **peak memory** needed.

3.1.3 Determining Peak Memory

To determine the peak memory, we need the answers to the below questions.

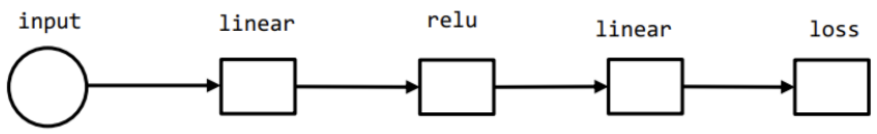
- How large is the memory? - we need the **size** of memory based on the sources of memory consumption in 3.1.

- What is the **lifetime** of the 'memory needed'? - As we always care about 'peak' memory, we need to take the lifetime of the memory needed also into consideration. Because, some pieces of memory are needed at some point in time, but are not needed at another point in time.

So for every memory consumption component, we need two things: **size** and **lifetime**. Now we can dive deeper into how to get the answers to the above questions.

3.2 Lifetime - Inference

Consider the inference process of a feedforward neural network:
Since this is an inference task, memory consumption for the *optimizer* is not a concern. We primarily focus



on the memory lifetime of *weights* and *activations*. In a feedforward network, the computation at each layer follows a consistent pattern:

1. Compute the weighted sum of the layer's input.
2. Apply the activation function to the weighted sum to obtain the layer's output.

These steps are repeated for every layer in the network. The activation from the previous layer can be discarded immediately after computing a weighted sum out of it. To minimize memory usage, we only need two memory buffers: one for storing the input of the layer, and the other to store the activated output. By cycling through these two buffers, a feedforward network with an arbitrary number of layers can be computed efficiently. In summary, memory of the model's weights live throughout the inference process, while that of layer's activation outputs are overwritten immediately as the next layer's computation begins.

3.3 Size

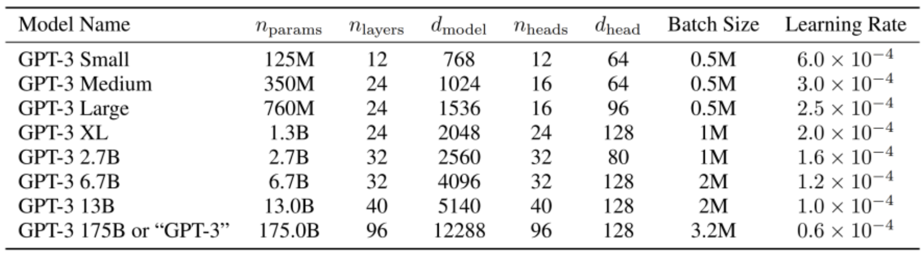
3.3.1 Floating Point Standards

To estimate the size of memory consumption, it's important to know the datatype of the variables taking up the memory. Common datatype for deep learning are:

- FP32 (4 Bytes each): 1-bit sign, 8-bit exponent, 23-bit mantissa
- FP16 (2 Bytes each): 1-bit sign, 5-bit exponent, 10-bit mantissa
- BFloat16 (also BF16, 2 Bytes each): 1-bit sign, 8-bit exponent, 7-bit mantissa

The details for calculating these floating point numbers from bit representations are covered in later sections.

3.3.2 GPT-3 Weight Size Estimation



Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

A simplistic estimation of the parameter memory consumption of the full 175B GPT-3 is

$$175 \text{ billion } \times \text{ sizeof(a\_parameter)}$$

where each parameter can take up either 2 Bytes (16 bits) or 4 Bytes (32 bits). This means the memory space needed for the weights are 175 billion $* 2 = 350$GB or 175 billion $* 4 = 700$GB.
A general rule of thumb for estimating model size: simply multiply the number of parameters with 2 or 4. Most models are stored in either 16- or 32-bit format.

3.3.3 Activation Size Estimation

The size of activations is relatively simple to calculate. For a convolution layer activation with batch size B, channel number C and spatial dimension H amd W, the total activation size in memory is:

$$B * C * H * W * \text{sizeof(element)}$$

Similar applies to multilayer perceptron (MLP) networks. For batch size B and output spatial dimension of $M \times P$, the activation size is

$$B * M * P * \text{sizeof(element)}$$

Ignoring the inner working mechanisms of transformer attention blocks, the size of memory its activated output takes:

$$B * H * L_{seq} * \text{sizeof(element)}$$

where B is the batch size, H the embedding dimension and L_{seq} the length of the input sequence to the block.

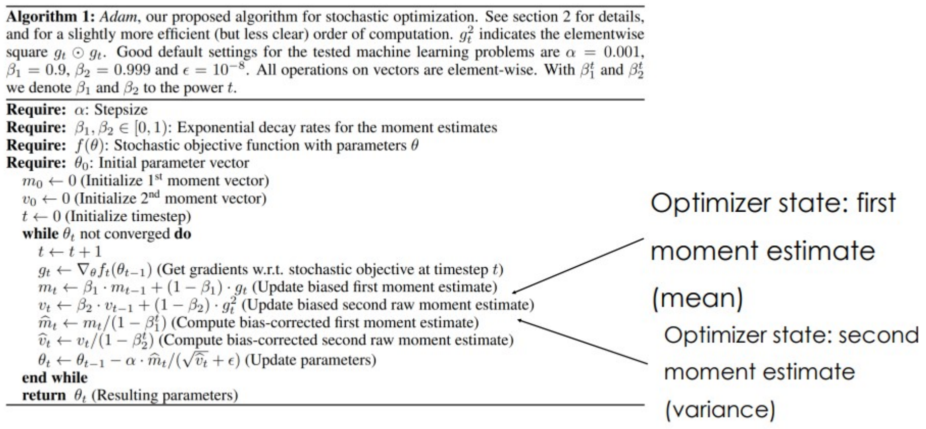
With this we can revisit the full GPT-3 model, with batch size 3.2M and vector embedding dimension 12,288. Assuming a sequence length of 1, the number of elements of each transformer layer's activation would be

$$3.2\text{M} * 12288 = 39.321\text{B}$$

which considering each element taking 2 or 4 bytes, gives the total memory needed for the activation to be 78.64GB or 157.28GB.

3.3.4 Optimizer State Size (Adam)

The main difference between Vanilla Stochastic Gradient Descent (SGD) and Adam is how they update the parameters using gradients. Vanilla SGD relies only on the first moment (the raw gradient) with a fixed learning rate. In contrast, Adam utilizes both the first moment (mean of gradients) and the second moment (variance of gradients) to adaptively adjust the learning rate for each parameter, leading to more stable and efficient updates.



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

Optimizer state: first moment estimate (mean)
Optimizer state: second moment estimate (variance)

Figure 4: Snippet of Adam Algorithm from the original ICLR paper

Figure 4 outlines the algorithm step by step. It demonstrates that when using mini-batches, certain states are initialized to track the first and second moments along the optimization trajectory. Each time the gradient is computed, these moments are updated. The normalized gradients are then derived using these moments. However, a key consideration is that we have to store first and second moments also throughout the optimization process, alongside the trajectory, to ensure accurate updates.

Size The memory required to store the optimizer state for the Adam algorithm can be estimated as follows:

- **Gradient with respect to parameters**: $N \times \text{sizeof(element)}$
- **First moment**: $N \times \text{sizeof(element)}$
- **Second moment**: $N \times \text{sizeof(element)}$

In total, the memory required for the Adam optimizer state is:

$$\text{Total Memory} = 3N \times \text{sizeof(element)}$$

Here, N represents the number of parameters in the model, and sizeof(element) is the size (in bytes) of a single parameter or gradient element (e.g., 4 bytes for a 32-bit float). This calculation shows that the Adam algorithm requires three times the memory of the model's parameters to store the gradients, first moment, and second moment.

3.4 Lifetime - Training

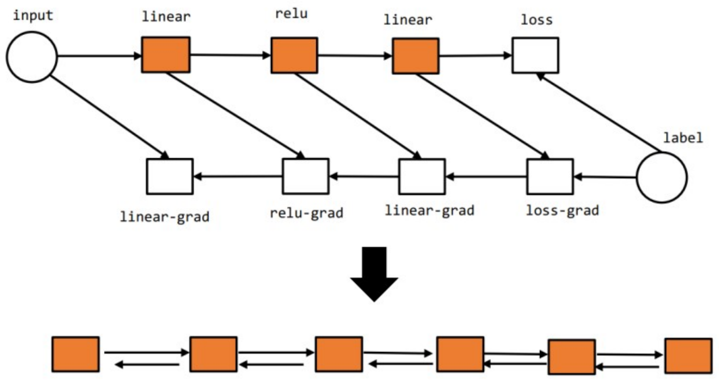


Figure 5: Training Graph and its Simplified version

In the context of training a neural network, tensors can be categorized as live tensors or dead tensors based on their usage during computation. Memory allocation is primarily required for live tensors, which are actively used in the computation graph.

We can enhance an inference graph a little bit to a training graph. In an inference graph, only forward computation is performed, while in a training graph, there is an additional backward graph for computing gradients and an optimizer state updating graph for applying updates. These components together form the full training graph.

Figure 5 shows an example where a backward graph can be mapped to its forward graph, the optimizer graph has been removed to simplify the structure. We can fold this together into one line as a chain of forward and backward edges.

Model parameters cannot be discarded at any time, as they are essential for either computing gradients or applying updates. They must remain allocated for the duration of training.

Activation tensors, however, may sometimes be discarded, but in most cases, they need to be retained. For Figure 5, it can't because we need it for intermediate values for backpropagation.

In brief, Training an N-layer neural network requires O(N) memory due to the need to store these intermediate activations. Optimization techniques can later be introduced to allow selective discarding of activations.

3.5 Memory overview for GPT-3

Note: Activation memory size is not accurate because transformers are composite layers.

1. Model Parameters

- **Parameters**: 175B

- **Precision**: FP16/ FP32
- **Memory**: 175B $\times$ 2 bytes = 350 GB(FP16) or 175B $\times$ 4 bytes = 700 GB(FB32)

2. **Activations**
   - **Layers**: 96 (At Transformer Boundary)
   - **Precision**: FP16/FP32
   - **Memory per Layer**: 78 GB
   - **Total Memory**: 96 $\times$ 78 GB = 7488 GB (FP16) or 96 $\times$ 156 GB = 14976 GB (FP32)

3. **Optimizer States**
   - **Optimizer**: Adam (first and second moments)
   - **Precision**: FP32
   - **Memory per Parameter**: 3 $\times$ 4 bytes = 12 bytes
   - **Total Memory**: 3 $\times$ 175B $\times$ 4 bytes = 2100 GB

4 Memory Optimization Techniques

Deep neural networks, including transformers, convolutional neural networks (CNNs), and large-scale generative models, require significant memory during training. One major contributor to this high memory consumption is the storage of intermediate activations during the forward pass. These activations are essential for computing gradients in the backward pass. However, many of these activations are not immediately needed, which leads to inefficient memory usage and limits the maximum feasible batch size or input resolution due to GPU VRAM constraints.

To overcome these challenges, several memory optimization techniques have been developed. In this document, we discuss three such techniques:

- **Gradient Checkpointing (Activation Checkpointing)**: Selectively storing only a subset of activations and recomputing the rest during backpropagation.
- **Gradient Accumulation**: Training with large effective batch sizes while keeping per-step memory consumption low.
- **CPU Swapping**: Offloading data from GPU high-bandwidth memory (HBM) to CPU DRAM when needed.

4.1 Gradient Checkpointing: Reducing Activation Memory

4.1.1 Motivation and Overview

Training deep networks typically involves storing the outputs (activations) of every layer during the forward pass. These activations are required during the backward pass for computing gradients via the chain rule. However, storing all these activations can quickly exhaust available GPU memory. **Activation checkpointing** addresses this by only storing activations at selected points (checkpoints) and recomputing intermediate activations as needed during the backward pass.

4.1.2 Key Concepts

- **Forward Pass**: The input is processed through multiple layers, generating activations.
- **Backward Pass**: The gradients are computed using the chain rule, requiring stored activations.
- **Checkpointing**: Instead of storing all activations, only a few are saved. The rest are recomputed when needed.

4.1.3 How It Works

1. **Divide the model into segments:** Save only the activations at the boundaries of segments.
2. **Recompute Intermediate Activations:** During backpropagation, recompute the activations for the layers between checkpoints as needed.

4.1.4 Memory-Compute Trade-off

Using activation checkpointing introduces a trade-off:

- **Memory Savings:** Reduced memory usage by not storing every activation.
- **Increased Computation:** Additional computation is required to recompute activations.

4.1.5 Comparison Table

Strategy	**Memory Usage**	**Computation Cost**
Store all activations	High	Low
Recompute all activations	Low	High
Balanced checkpointing	Medium	Medium

Table 1: Trade-offs between memory usage and computation cost.

4.1.6 Mathematical Model

For a network with N layers, if checkpoints are placed every K layers, the overall memory cost is approximated by:

$$\text{Memory Cost} = O\left(\frac{N}{K}\right) + O(K)$$

Here:

- $O\left(\frac{N}{K}\right)$ is the cost to store the checkpointed activations.
- $O(K)$ is the cost incurred when recomputing activations between checkpoints.

The near-optimal trade-off is achieved when:

$$K = \sqrt{N}$$

4.1.7 Practical Implementation: `torch.utils.checkpoint`

PyTorch provides a convenient interface for activation checkpointing:

Listing 1: PyTorch Implementation of Activation Checkpointing

```
import torch
import torch.nn as nn
import torch.utils.checkpoint as checkpoint

class CheckpointedModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(1024, 1024)
        self.layer2 = nn.Linear(1024, 1024)
        self.layer3 = nn.Linear(1024, 1024)

    def forward(self, x):
        x = checkpoint.checkpoint(self.layer1, x)
        x = checkpoint.checkpoint(self.layer2, x)
        x = checkpoint.checkpoint(self.layer3, x)
        return x
```

4.1.8 Discussion and Considerations

- **When to Use:** Best suited for deep networks (e.g., Transformers, ResNets, GANs) where memory is a constraint.
- **When to Avoid:** Less effective for shallow networks or cases where the recomputation overhead is prohibitive.
- **Limitations:** Only activations are checkpointed. The technique increases computation and may introduce numerical inconsistencies in stochastic layers.

4.2 Gradient Accumulation: Training with Large Effective Batch Sizes

4.2.1 Motivation

Large batch sizes are often beneficial for training stability and convergence. However, a larger batch also means higher activation memory usage. **Gradient accumulation** allows training with a large effective batch size while keeping the per-iteration memory footprint small.

4.2.2 Concept

Instead of updating the model parameters after every mini-batch, gradients are accumulated over several micro-batches:

1. Compute gradients on several small mini-batches (micro-batches).
2. Accumulate the gradients.

3. Perform a weight update after the accumulation, simulating a larger batch size.

4.2.3 Mathematical Formulation

Let bs be the desired large batch size and mbs the mini-batch size, with $\#\text{mb} = \frac{bs}{mbs}$. The update rule becomes:

$$\text{for } t = 1 \to \#\text{mb}: \quad \nabla\theta \mathrel{+}= \nabla L(\{x_i\}_{i=1}^{mbs}, \ldots)$$

After accumulating:

$$\theta = \theta - \eta\nabla\theta$$

4.2.4 Implementation in PyTorch

Without Gradient Accumulation (Standard Training Loop):

Listing 2: Standard Training Loop Without Gradient Accumulation

```
optimizer = ...

for epoch in range(...):
    for i, sample in enumerate(dataloader):
        inputs, labels = sample

        # Forward Pass
        outputs = model(inputs)

        # Compute Loss and Backpropagate
        loss = loss_fn(outputs, labels)
        loss.backward()

        # Update parameters
        optimizer.step()
        optimizer.zero_grad()
```

With Gradient Accumulation:

Listing 3: Training Loop With Gradient Accumulation

```
optimizer = ...
NUM_ACCUMULATION_STEPS = ...

for epoch in range(...):
    for idx, sample in enumerate(dataloader):
        inputs, labels = sample

        # Forward Pass
        outputs = model(inputs)

        # Compute Loss and Backpropagate
        loss = loss_fn(outputs, labels)

        # Normalize the gradients to simulate large batch size
    loss = loss / NUM_ACCUMULATION_STEPS
    loss.backward()

    if ((idx + 1) % NUM_ACCUMULATION_STEPS == 0) or (idx + 1 == len(dataloader)):
        optimizer.step()
        optimizer.zero_grad()
```

4.2.5 Advantages and Challenges

- **Advantages:**
  - Enables training with large effective batch sizes under memory constraints.
  - Improves training stability.
  - Integrates easily with existing frameworks.
- **Challenges:**
  - Delayed weight updates may affect optimization dynamics.
  - Increased overall training time due to more forward/backward passes.
  - Synchronization overhead when using multiple GPUs.

4.3 CPU Swapping and Alternative Memory Management

4.3.1 Motivation: Leveraging the Memory Hierarchy

Modern systems feature different memory types:

- **SRAM (GPU Cache):** Extremely fast but very limited in capacity.
- **HBM (High Bandwidth Memory, e.g., GPU VRAM):** Fast and reasonably large (e.g., 40GB), but still limited.
- **DRAM (CPU Main Memory):** Much larger in capacity (e.g., $>$ 1TB) but slower.

When training large models, even HBM can become a bottleneck. CPU swapping is a technique that offloads data from GPU memory (HBM) to the larger CPU DRAM.

4.3.2 CPU Swapping: The Technique

CPU Swapping involves two primary operations:

a. **SwapOut:** Moving activations or model weights from GPU VRAM to CPU DRAM.

b. **SwapIn:** Retrieving data from CPU DRAM back to GPU VRAM when required for computation.

4.3.3 Mechanism in Deep Learning

During training:

- In the **forward pass**, some activations may be swapped out to free up GPU memory.
- In the **backward pass**, these activations are swapped back in before computing gradients.

This dynamic data transfer allows training models that exceed the GPU's VRAM capacity by leveraging the larger but slower CPU memory.

4.3.4 When CPU Swapping is Effective

- It works well when computations are heavy, allowing data transfers to occur asynchronously.
- Suitable when swapped data is not needed immediately.
- Efficient when data transfer overhead is minimized through optimization.

4.3.5 Limitations and Challenges

- Frequent swapping may cause data transfer bottlenecks.
- Increased latency can slow down training, especially if the model is memory-bandwidth constrained.
- Not ideal for scenarios requiring constant, rapid access to the data.

4.4 Summary of Memory Optimizations

4.4.1 Gradient Checkpoint

Pros: It can reduce the intermediate tensors.
Cons: Increases the number of FLOPs and slows down computation.

4.4.2 Gradient Accumulation

Pros: No additional FLOPs cost.
Cons: It splits the batches into smaller batches known as **micro-batching.** To ensure optimal GPU utilization in parallel computing, maintaining a sufficient level of arithmetic intensity is crucial. A larger batch size leverages the GPU's full computational capacity when using more data parallelism.

4.4.3 CPU Swapping

Pros: Reduces memory usage of parameters, activations, and optimization states.
Cons: It is extremely slow. However, techniques like unified memory can significantly improve the CPU swapping technique. **Unified Memory** uses a common memory pool shared between the CPU and GPU, allowing seamless data access without explicit memory transfers.

5 Memory and Compute

5.1 Introduction

Modern machine learning models, particularly deep learning architectures, demand significant memory and computational resources. As model sizes continue to grow, optimizing both memory usage and computational efficiency becomes critical for practical deployment and scalability.

Memory consumption arises from storing model parameters, activations, and optimizer states, while compute efficiency is affected by the precision of numerical representations and hardware optimizations. To address these challenges, techniques such as quantization, mixed-precision training, and efficient data representation are employed. These methods enable models to fit within limited hardware constraints, enhance execution speed, and reduce energy consumption.

This section delves into fundamental strategies for optimizing memory and compute efficiency. We begin with quantization, a widely used technique for reducing model size and computational complexity while maintaining acceptable accuracy. Understanding these principles is crucial for developing scalable machine learning systems and deploying models on resource-constrained devices such as mobile phones and embedded systems.

5.2 Quantization

5.2.1 What is Quantization?

Quantization is the process of constraining an input from a **continuous** or otherwise **large set of values** to a **discrete set**, as shown in Figure 6. This technique is widely used in digital signal processing, machine learning, and data compression to efficiently represent large-scale data while minimizing storage and computational costs.

Since continuous signals require **high precision and large memory** to store, directly storing them can be expensive.

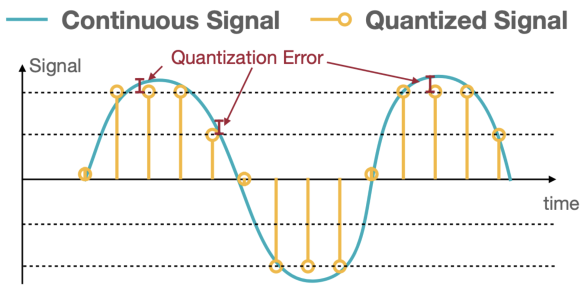


Figure 6: Representation of discrete values and continuous values in quantization.

In the figure 7's right image, we observe a **quantized image** with **lower precision**, which appears slightly blurred due to reduced detail and constrained pixel representation. On the other hand, the left image is the **high-precision original image**, which retains all the fine details and appears much clearer.

In this example, **quantization is constraining the number of pixels**, effectively reducing the level of detail stored in the image.

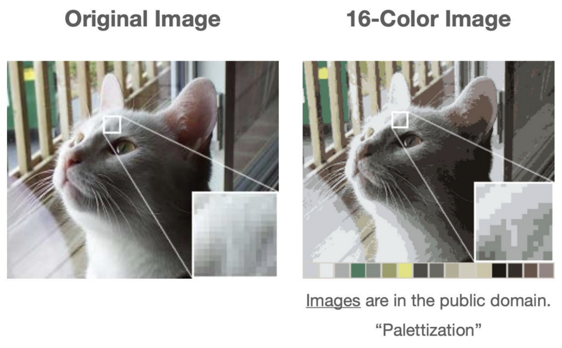


Figure 7: Example of Quantization on an image of a cat

5.2.2 Why Do we Care about Quantization ?

Quantization is a crucial technique in optimizing memory and computational efficiency, particularly in deep learning and other high-performance computing applications. One of the key motivations for quantization is to **reduce memory usage** by representing data with fewer bits. The total memory required for storing weights, activations, and optimizer states is a **multiple of the size of each element** (`sizeof(element)`). By reducing the size of each element, we can proportionally decrease the overall memory footprint, making it feasible to run large models.

Can we reduce `sizeof(element)`? Strategies like using **lower-precision data types** (e.g., FP16, INT8 instead of FP32) help achieve this reduction. This trade-off between memory requirement and precision requirement, must be carefully considered.

5.2.3 Benefits of Quantization

1. **Preserve ML Performance:** Despite using lower precision, the model should still be able to correctly identify key features.

2. **Accelerate Computation:** Modern hardware, including specialized accelerators and AI chips, is optimized for quantized computations. Lower precision operations (e.g., INT8 vs. FP32) allow for faster execution and increased throughput.

3. **Reduce Memory Usage:** Deep learning models often have millions or even billions of parameters. Storing them in lower precision formats significantly reduces memory consumption, enabling deployment on resource-constrained devices.

4. **Save Energy:** Lower precision reduces the number of required matrix multiplications and arithmetic operations, leading to fewer instruction cycles and lower power consumption.

5.2.4 Key Concepts in Quantization and Training Techniques

- **Digital Representation of Data:** Data is encoded into discrete values for digital processing.
- **Basics of Quantization:** Reduces precision to map continuous values to a discrete set.
- **Quantization in ML:** Lowers model precision to save memory and speed up inference.

  - **Post-Training Quantization:** Converts trained models to lower precision after training.
  - **Quantization-Aware Training:** Trains models while keeping check over quantization effects to preserve accuracy.

- **Mixed Precision Training:** Combines different precisions (e.g., FP16 & FP32) for efficiency.

A General mobile Phone runs over above mentioned processes: Mobile devices use quantization and mixed precision to optimize AI models. **Mixed precision training is the key technique for Language Models:** Enables efficient training of large language models with lower memory usage.

5.3 Basics of Data Representation

5.3.1 Integer Representation

Integer representation defines how numerical values are stored in binary form. The three common types are:

1. **Unsigned Integer:**
   - Represents only non-negative values.
   - Range for an n-bit number: 0 to $2^n - 1$.
   - Example (4-bit): $0000 = 0$, $1111 = 15$.

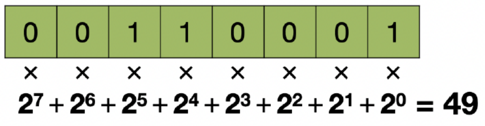


Figure 8: Unsigned Integer

2. **Signed Integer:**
   - Supports both positive and negative values.
   - Typically, the Most Significant Bit (MSB) acts as the sign bit (0 for positive, 1 for negative).
   - Range for an n-bit number: $-(2^{n-1} - 1)$ to $2^{n-1} - 1$.
   - Example (4-bit, Sign-Magnitude Representation): - $0000 = +0$, $1000 = -0$ (redundant representation).

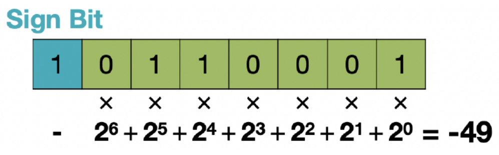


Figure 9: Signed Integer

3. **Two's Complement:**
   - A widely used method for representing signed integers.

- Negative numbers are stored by inverting the bits and adding one.
- Range for an n-bit number: -2^{n-1} to $2^{n-1} - 1$.
- Example (4-bit): - 0000 $= 0$, 1111 $= -1$, 1000 $= -8$, 0111 $= 7$.

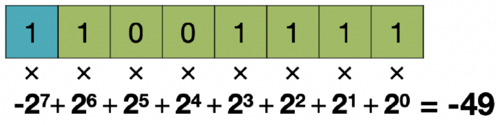


Figure 10: Two's Complement representation

5.3.2 Fixed-point Numbers

A fixed-point number is generally expressed as: $(SignBit\ IntegerBits\ .\ FractionalBits)$, where the decimal point is fixed at a specific position. Unlike floating-point representation, which dynamically adjusts the decimal point, fixed-point representation uses a predetermined position, ensuring predictable precision and reduced computational overhead. Therefore, it is primarily used in security-related applications and is less common in Machine Learning.

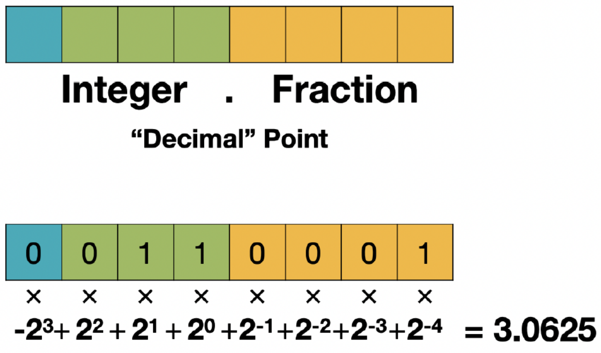


Figure 11: Fixed-Point Numbers

5.3.3 Floating Point Representation

The IEEE 754 standard is the most commonly used format for floating-point representation. A floating-point number consists of three key components:

- Sign bit (1 bit): Determines whether the number is positive or negative.
- Exponent (8 bits for single-precision): Stores the exponent in a biased form using a bias of 127.
- Fraction (23 bits, also called the Mantissa or Significant): Stores the fractional part of the number.

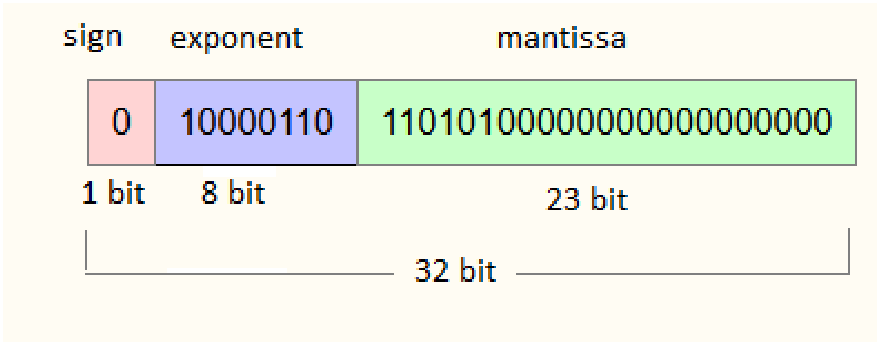


Figure 12: IEEE 754 floating point representation

Formula for IEEE 754 Representation

$$\underbrace{(-1)^{\text{sign}}}_{\text{(a)}} \times \underbrace{(1 + \text{Fraction})}_{\text{(b)}} \times \underbrace{2^{\text{Exponent}-127}}_{\text{(c)}}$$

(a) If the sign bit is 0, the number is positive; if 1, it's negative

(b) The mantissa represents the significant digits of the number, with an implicit leading 1

(c) The exponent is stored with a bias of 127, so we subtract 127 to get the actual exponent

How to represent 0 Since normal numbers assume an implicit leading 1, a direct representation of 0 is not possible. Instead, zero is represented as:

- Positive Zero (+0): 0 00000000 00000000000000000000000
- Negative Zero (-0): 1 00000000 00000000000000000000000

Both representations are treated as zero in arithmetic operations, but they can behave differently in cases such as division and underflow.

Normal vs. Subnormal Floating-point numbers in the IEEE 754 single-precision (fp32) format can be categorized into normal and subnormal numbers, depending on their exponent value.

- **Normal Numbers (Exponent $\neq 0$):**

  $$(-1)^{\text{sign}} \times (1 + \text{Fraction}) \times 2^{\text{Exponent}-127}$$

  These numbers utilize the full precision of the 23-bit fraction. The exponent is stored in biased form (Bias = 127), allowing representation of a wide dynamic range.

- **Subnormal Numbers (Exponent = 0):**

  $$(-1)^{\text{sign}} \times \text{Fraction} \times 2^{1-127}$$

  The leading 1 is no longer assumed, meaning the fraction is directly used. The exponent is forced to be 1 - 127 = -126.

Representation Power of fp32 The fp32 format can represent a wide range of numbers using its combination of exponent and fraction bits. The range is defined by:

- Maximum normal value: $\pm(2 - 2^{-23}) \times 2^{127} \approx \pm 3.4 \times 10^{38}$
- Minimum normal value: $\pm 2^{-126} \approx \pm 1.18 \times 10^{-38}$
- Maximum subnormal value: $\pm(1 - 2^{-23}) \times 2^{-126} \approx \pm 1.18 \times 10^{-38}$
- Minimum subnormal value: $\pm 2^{-149} \approx \pm 1.4 \times 10^{-45}$

Minimum Positive Values The smallest positive values that can be represented in fp32 depend on whether we're considering normal or subnormal numbers:

- Smallest positive normal number:
  - Exponent = 1 (unbiased = −126)
  - Fraction = 0
  - Value = $1.0 \times 2^{-126} \approx 1.18 \times 10^{-38}$
- Smallest positive subnormal number:
  - Exponent = 0 (denormal)
  - Fraction = smallest possible non-zero value (1 in LSB)
  - Value = $2^{-149} \approx 1.4 \times 10^{-45}$

Special Values IEEE 754 defines several special values for representing exceptional cases:

- **Infinity ($\pm\infty$)**:
  - Sign: 0/1 for $\pm\infty$
  - Exponent: All 1s (255)
  - Fraction: All 0s
  - Used for overflow and division by zero
- **NaN (Not a Number)**:
  - Sign: Either 0 or 1
  - Exponent: All 1s (255)
  - Fraction: Non-zero value
  - Used for undefined operations, such as 0/0 or $\infty - \infty$

These special values complete the IEEE 754 representation system, making it robust for handling both normal numerical operations and exceptional cases in floating-point arithmetic.

Summary of fp32 The IEEE 754 single-precision floating-point format (fp32) provides a comprehensive system for representing real numbers in computing:

- **Structure**: 32 bits total (1 sign bit, 8 exponent bits, 23 fraction bits)
- **Range**: Approximately $\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$ (normal numbers)
- **Special representations**:
  - Two zeros (± 0)
  - Subnormal numbers for gradual underflow
  - Infinities ($\pm\infty$)
  - NaN for undefined operations
- **Precision**: About 7 decimal digits of precision

Next Topic Preview The next lecture will build on our understanding of floating-point representation to explore practical applications in machine learning systems, focusing particularly on how these numerical representations affect the efficiency of model training and inference.

6 Scribe Contribution

- **Alex Oshima** : section 1, section 2
- **Rohith Vutukuru** : section 3.1
- **Ruobing Han** : section 3.2, section 3.3 (before 3.3.4)
- **Bhavik Chandna** : section 3.3.4, section 3.4, section 3.5, formatting, compiling
- **Mathew Luo** : section 4.1
- **Manoj Gayala** : section 4, section 4.2
- **Nikhil Paleti** : section 4, section 4.1, section 4.3
- **Akshit Agarwal** : section 4.4, 5.1, 5.3.1, 5.3.2, formatting.
- **Aaryan Gupta** : section 5.2
- **Shankara Narayanan Venkateswara Raju** : section 5.3.3 (before Representation Power of fp32)
- **Raghav Kachroo** : section 5.3.3 (Representation Power of fp32 and ahead)