

TALK



Nijesh Kanjinghat

AI Engineering Lead

IBM

Operationalizing GenAI: Effective LLM Compression and Optimization Methods

ODSC **APAC**

FREE | VIRTUAL
AUGUST 13

About me

- Nijesh Kanjinghat
- AI Engineering Lead at IBM Singapore.
- Love math, code and LLMs ☺
- Passions: Yoga & Hiking , Books ,Music.....
- [LinkedIn](#)
- [ODSC Github Repository](#)



Agenda

Introduction to building Generative AI applications

Challenges in operationalizing Generative AI

Understanding LLM Compression

- Quantization
- Pruning
- Knowledge Distillation
- Low-rank approximation

LLM Optimization Techniques

- Batching
- Model and Data Parallelism
- Speculative Inference

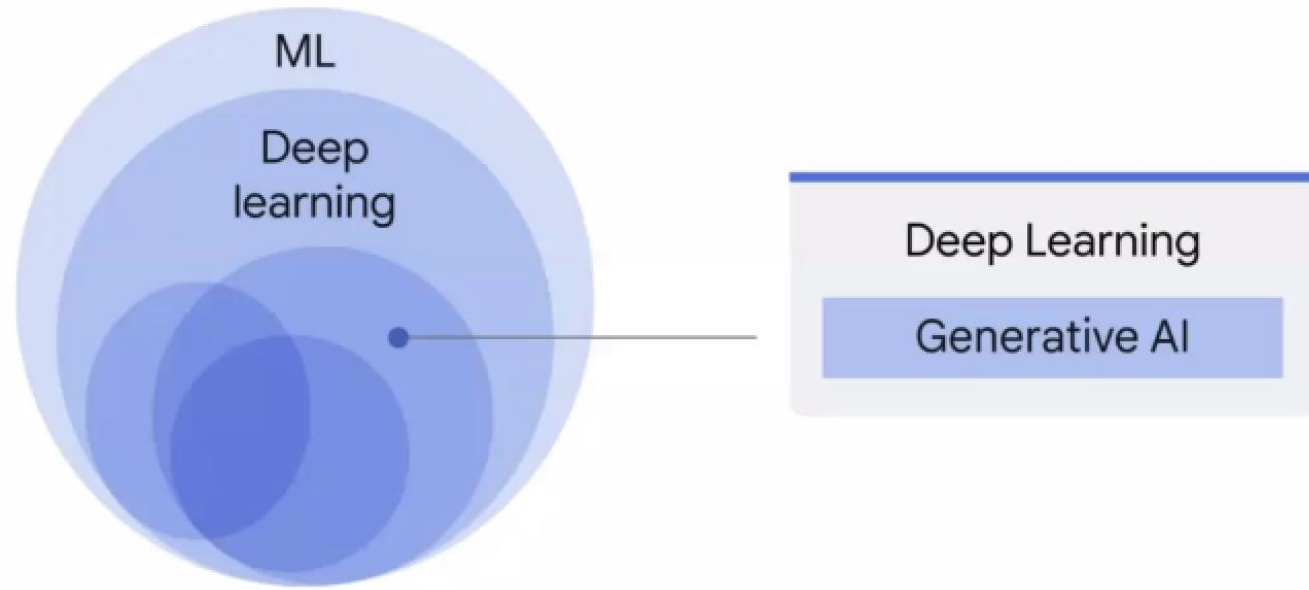
Example Case Study

Key Takeaways

Building Generative AI Applications

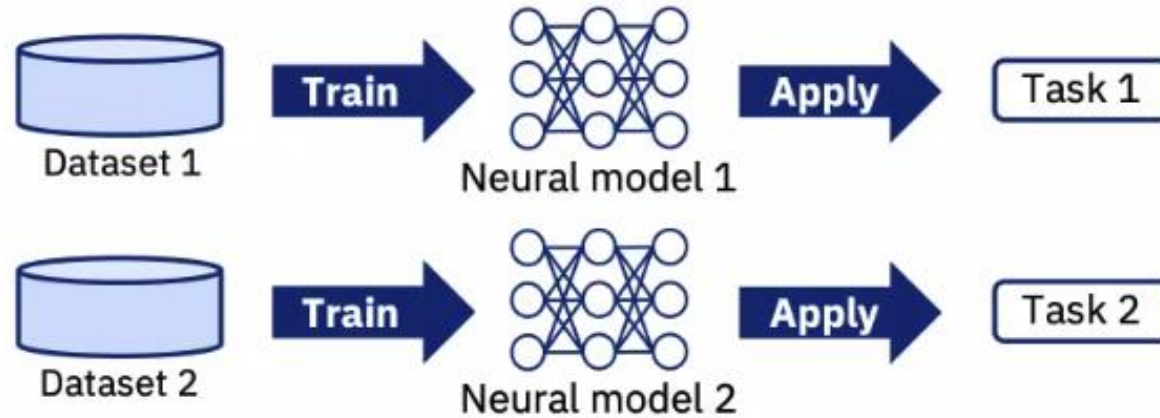
Generative artificial intelligence (AI) describes algorithms (such as ChatGPT) that can be used to create new content, including audio, code, images, text, simulations, and videos.

Generative AI
is a **subset of**
Deep Learning

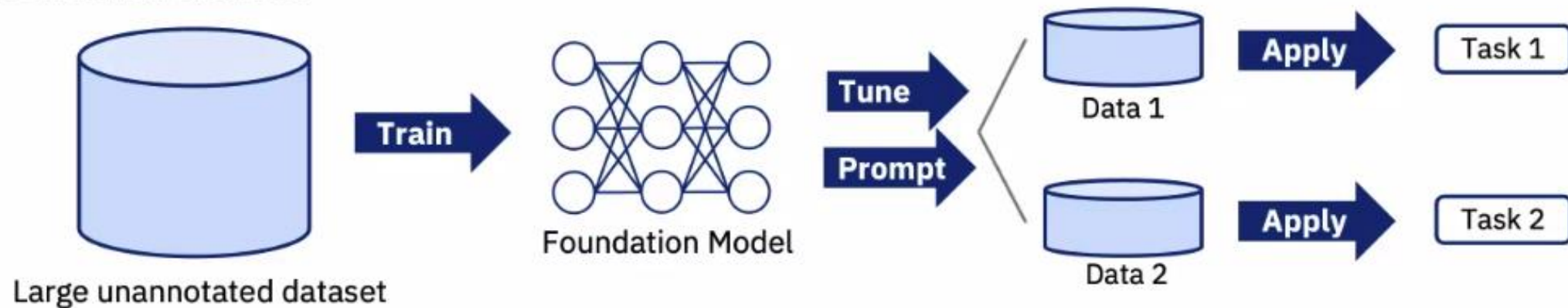


What are foundation models?

Conventional Machine Learning Systems:



Foundation Models:

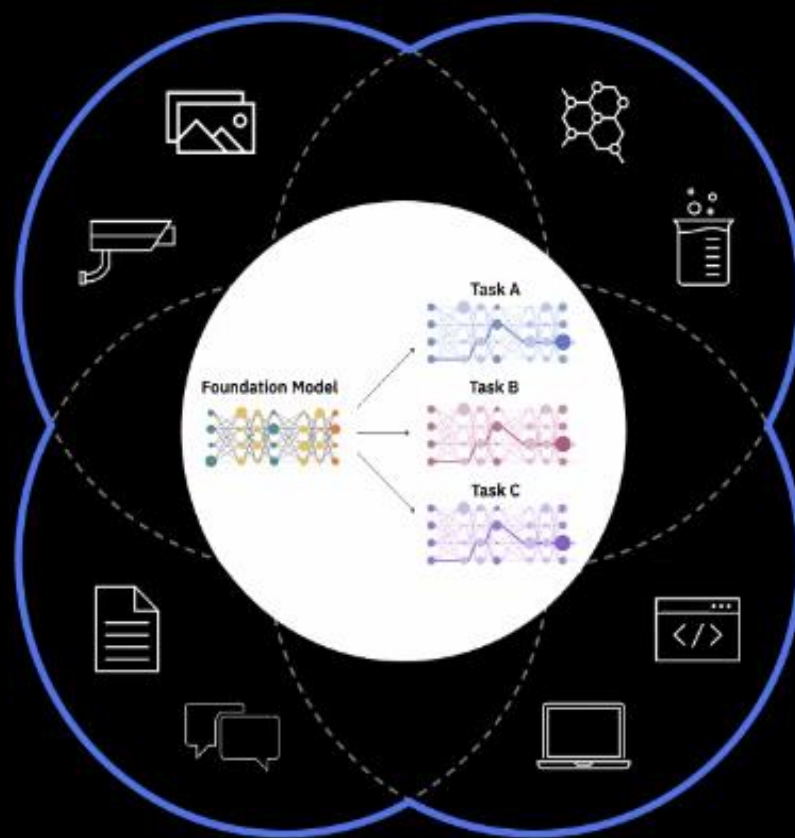


Foundation Models arise from NLP, but the opportunities and implications of Foundation Models go well beyond NLP

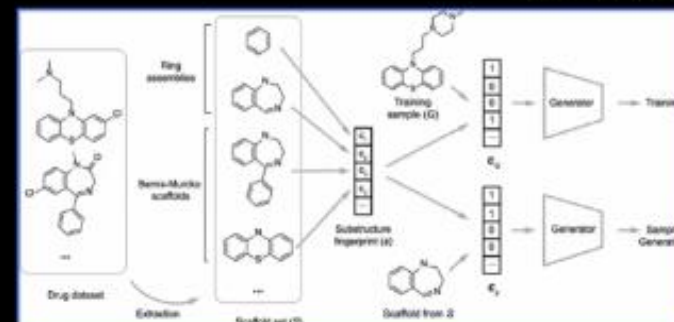
Computer vision



Natural language



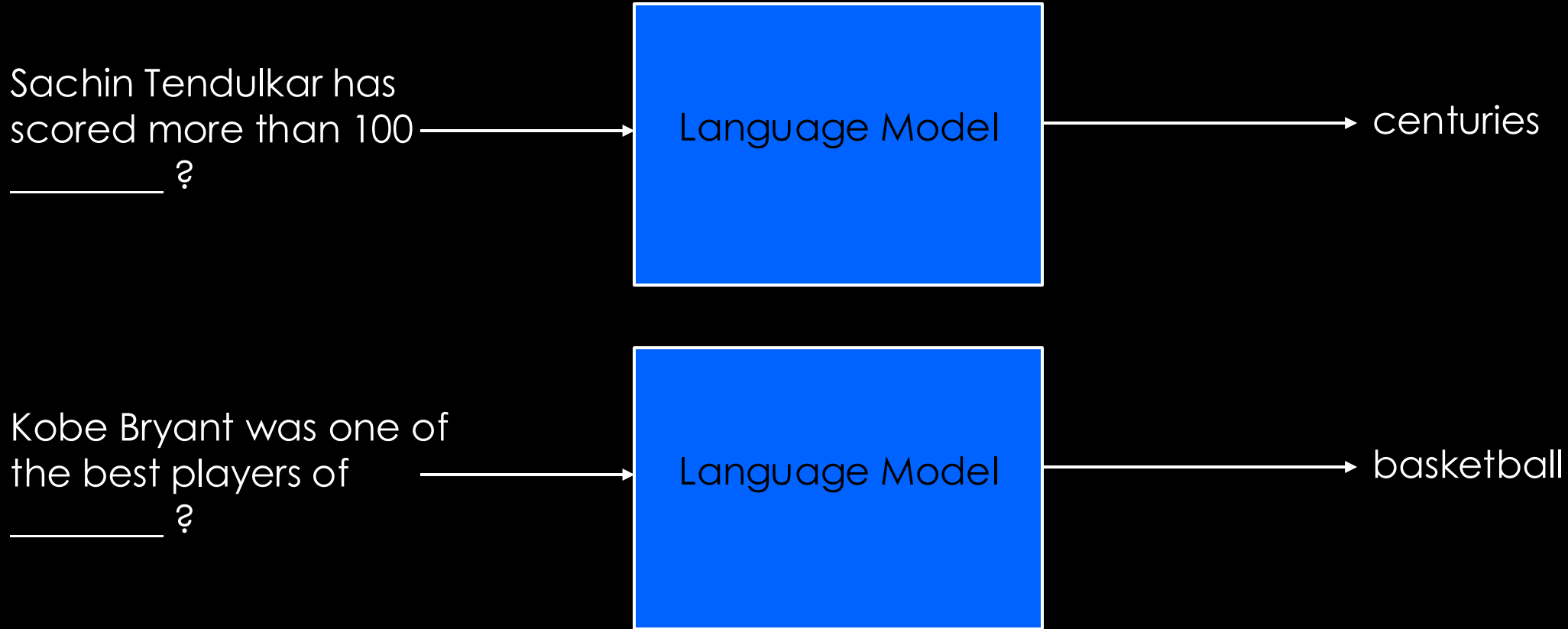
Chemistry



Programming Languages

What is a *Language Model*?

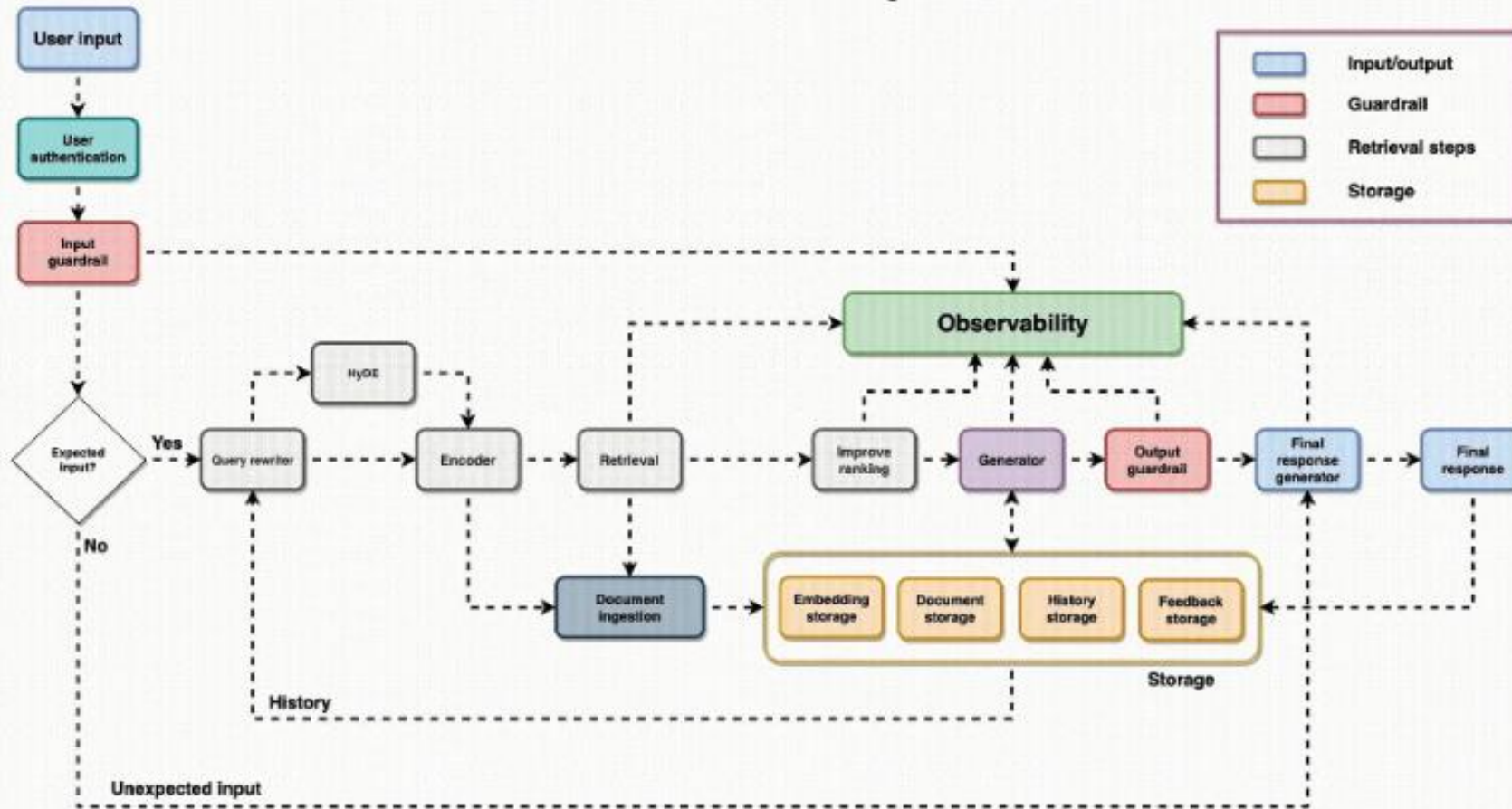
Language model predicts the next word in a sequence given the words that have appeared before



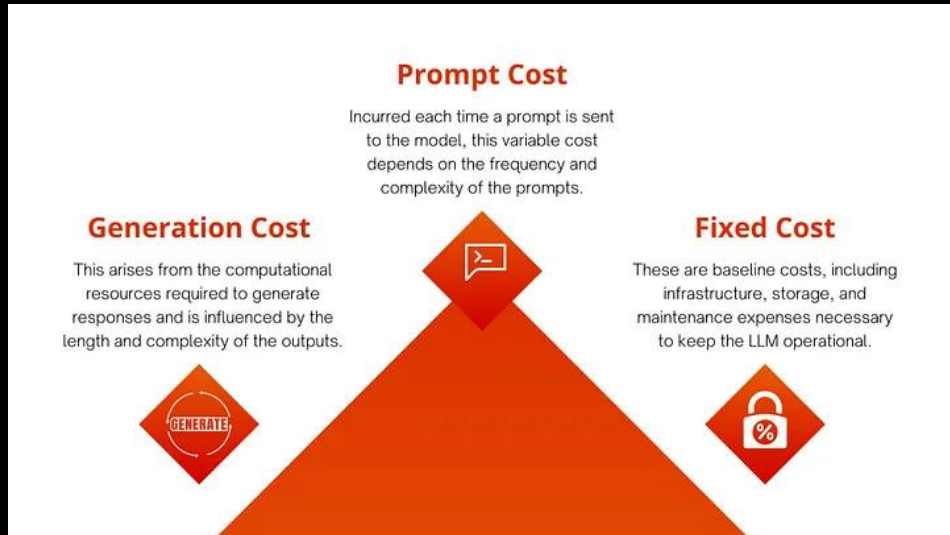
A TYPICAL GENERATIVE AI APPLICATION

8

Architecture For Enterprise RAG

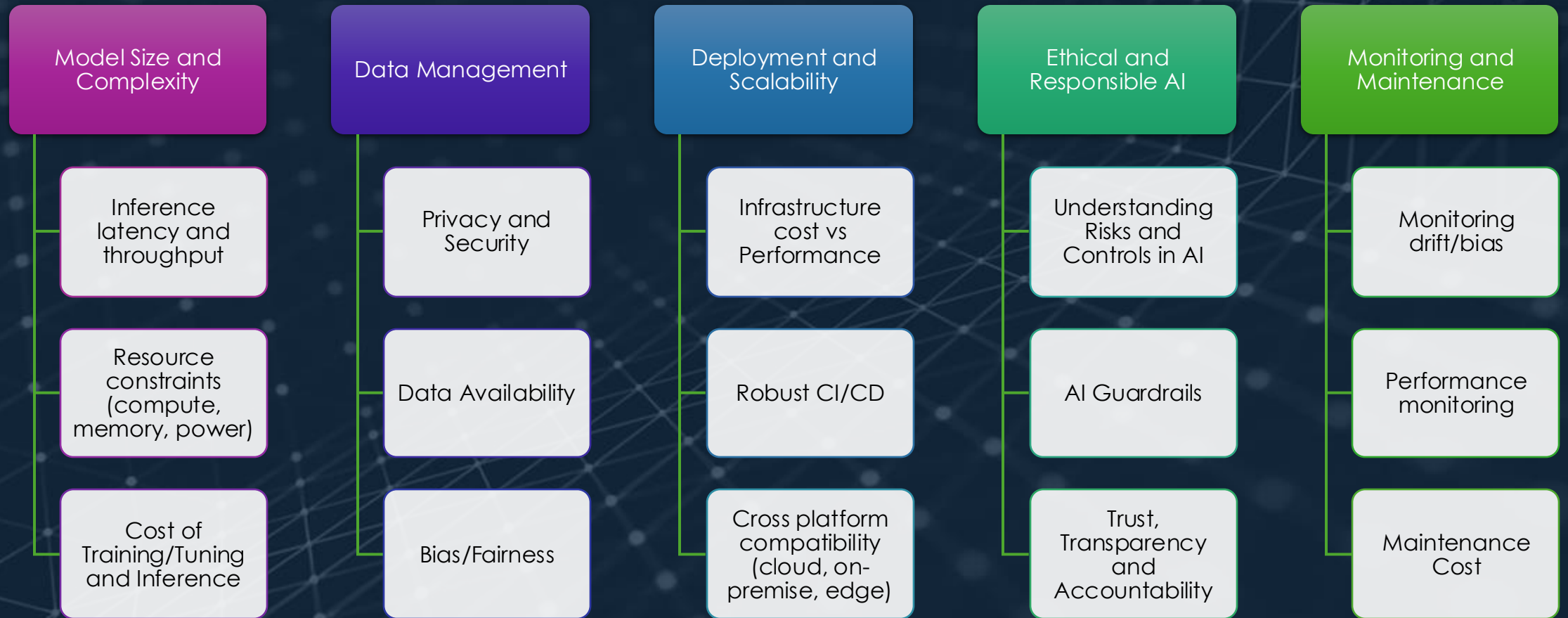


DAY -2 PROBLEMS WITH THESE APPLICATIONS



- **Combined Model Complexity:** Optimizing both the retrieval and generation models simultaneously requires careful consideration of their interdependencies.
- **Knowledge Base Integration:** Compressing the knowledge base or finding efficient representations can be challenging.
- **Performance Trade-offs:** Aggressive compression might impact the retrieval system's accuracy or the language model's quality.
- **Latency and Throughput:** Balancing the optimization of both retrieval and generation components to achieve desired performance metrics.
- **Hardware Constraints:** Efficiently utilizing hardware resources while considering the complexities of RAG systems.
- **Dynamic Knowledge Bases:** Ensuring optimization techniques can handle updates to the knowledge base.
- **Retrieval-Generation Interaction:** Optimizing the interaction between the retrieval and generation components for efficient inference.

Challenges in Operationalizing Gen AI



Operationalization Challenges

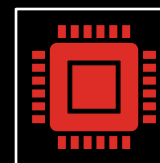
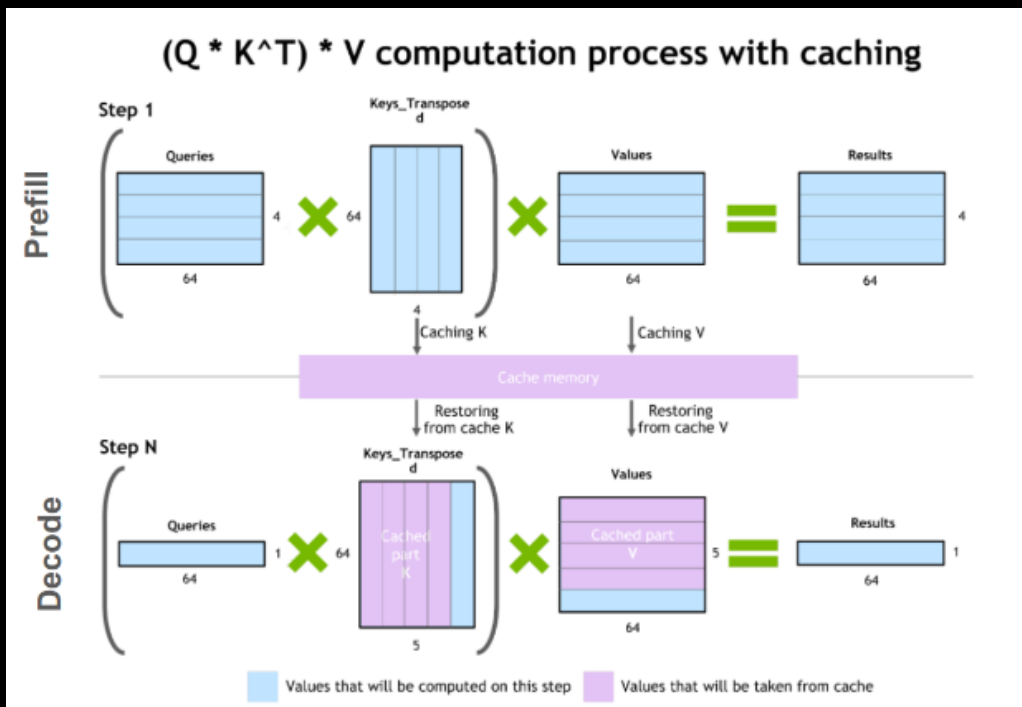
Focus for today

- High Memory footprint
- Limited Throughput and Latency
- Computational Cost

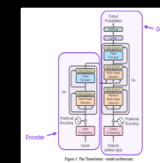


UNDERSTANDING LLM MEMORY FOOTPRINT

12



The GPU memory requirements for Large Language Models (LLMs) are humungous and are primarily driven by two factors:



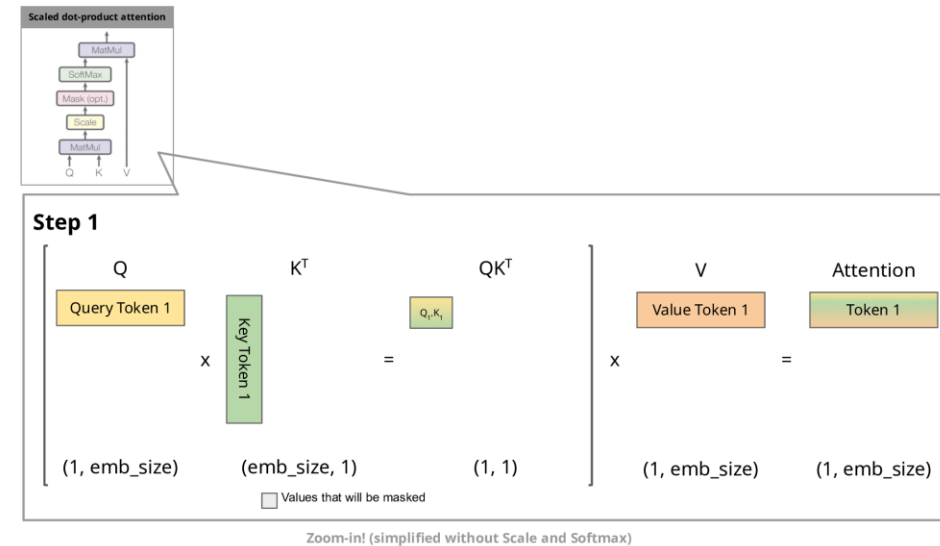
Model weights : Memory is consumed by the model's parameters. For example, a model with 8 billion parameters, like Llama 3 8B, when loaded in 16-bit precision (FP16 or BF16), requires approximately 16 GB of memory. This is calculated as $8B * \text{sizeof}(\text{FP16}) \approx 16 \text{ GB}$.



key-value (KV) cache : Memory is also required for caching self-attention tensors, which helps avoid redundant computations. In a batch setting, each request in the batch needs its own KV cache, leading to a significant memory footprint.


KV CACHE FOOTPRINT

- The size of the KV cache for most LLM architectures is given by the formula:
- Size of KV Cache per Token (in bytes) = $2 * (\text{num_layers}) * (\text{num_heads} * \text{dim_head}) * \text{precision_in_bytes}$.
- Here, the factor of 2 accounts for the K and V matrices. Typically, the product of $(\text{num_heads} * \text{dim_head})$ equals the `hidden_size` (or `d_model`) of the transformer.
- This memory requirement applies to each token in the input sequence across the entire batch.
- Assuming half-precision, the total KV cache size can be calculated as :
- Total Size of KV Cache (in bytes) = $(\text{batch_size}) * (\text{sequence_length}) * 2 * (\text{num_layers}) * (\text{hidden_size}) * \text{sizeof}(\text{FP16})$



UNDERSTANDING MODEL COMPRESSION

Model compression is a technique used to reduce the size and computational requirements of Large Language Models (LLMs) while preserving their performance as much as possible. This is crucial because LLMs are often massive, making them challenging to deploy on resource-constrained devices and computationally expensive to operate.



POPULAR COMPRESSION TECHNIQUES

Quantization

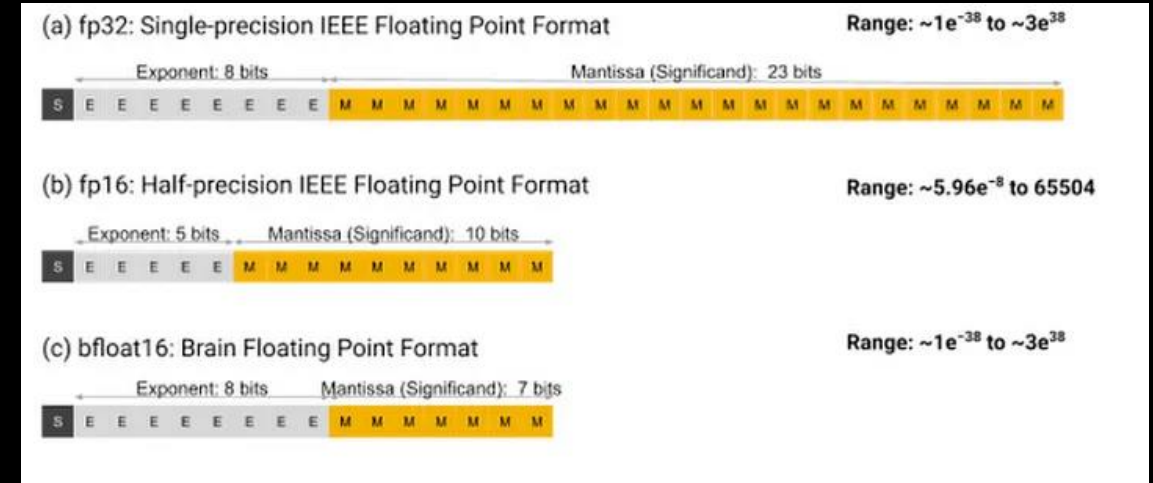
Pruning

Knowledge Distillation

Low rank approximation

QUANTIZATION

- process of reducing the precision of the weights, biases and activations of a neural network without significantly affecting the model performance.
- Most models are trained with 32 or 16 bits of precision, where each parameter and activation element takes up 32 or 16 bits of memory—a single-precision floating point.



QUANTIZATION

Quantization often can be done in two ways :

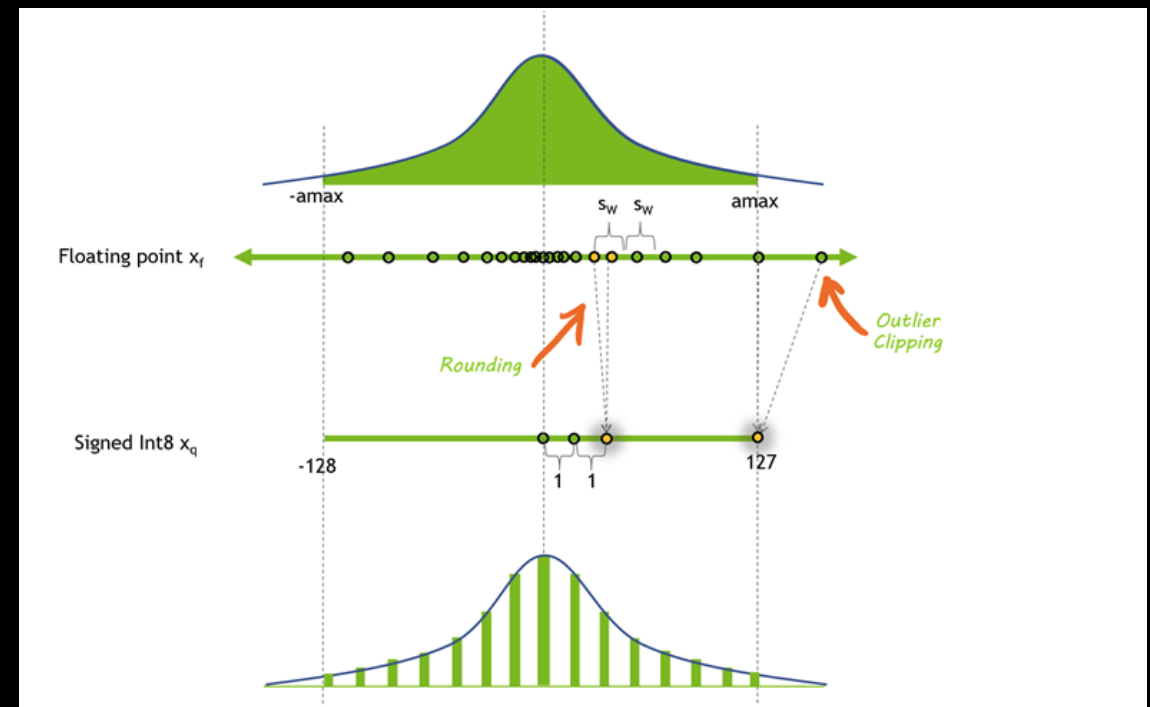
1. Post-training quantization
2. Quantization aware training

Quantization-Aware Training (QAT) simulates low-precision operations during training, allowing the model to adapt to quantization errors. This results in more robust parameters and minimal performance degradation.

Post-Training Quantization (PTQ) directly converts trained model parameters and activations to lower precision formats without retraining.

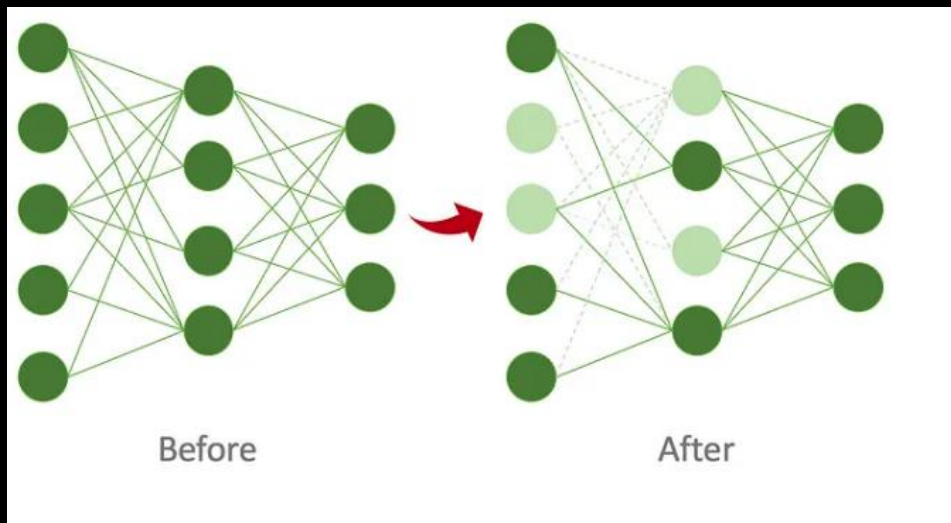
My Blog : <https://tinyurl.com/25r4tfw9>
NVIDIA : <https://tinyurl.com/yc6waz6y>

	Dynamic Range	Min Positive Value
FP32	$-3.4 \times 10^{38} \sim +3.4 \times 10^{38}$	1.4×10^{-45}
FP16	$-65504 \sim +65504$	5.96×10^{-8}
INT8	$-128 \sim +127$	1



The distribution of values before and after one possible method of quantization

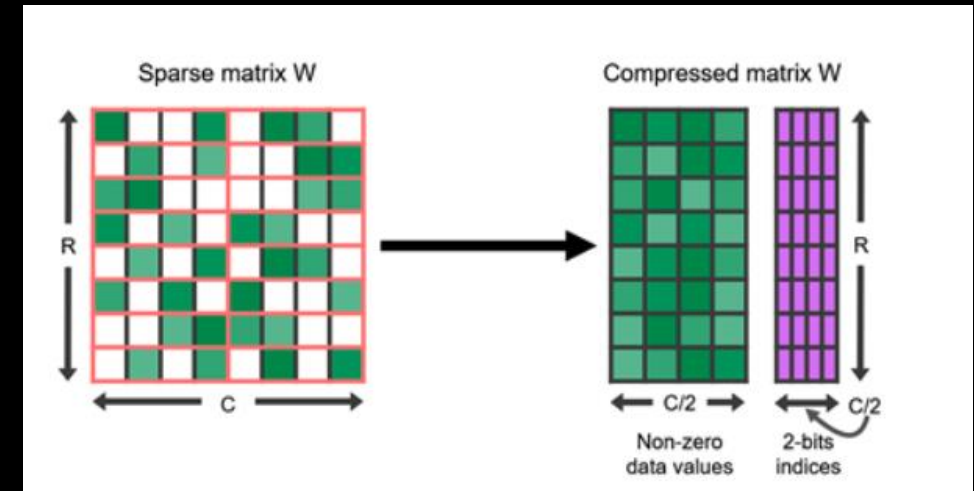
PRUNING



- Pruning involves removing unnecessary parameters or connections from a Large Language Model (LLM). The goal is to reduce the model's size and computational complexity without significantly impacting performance.

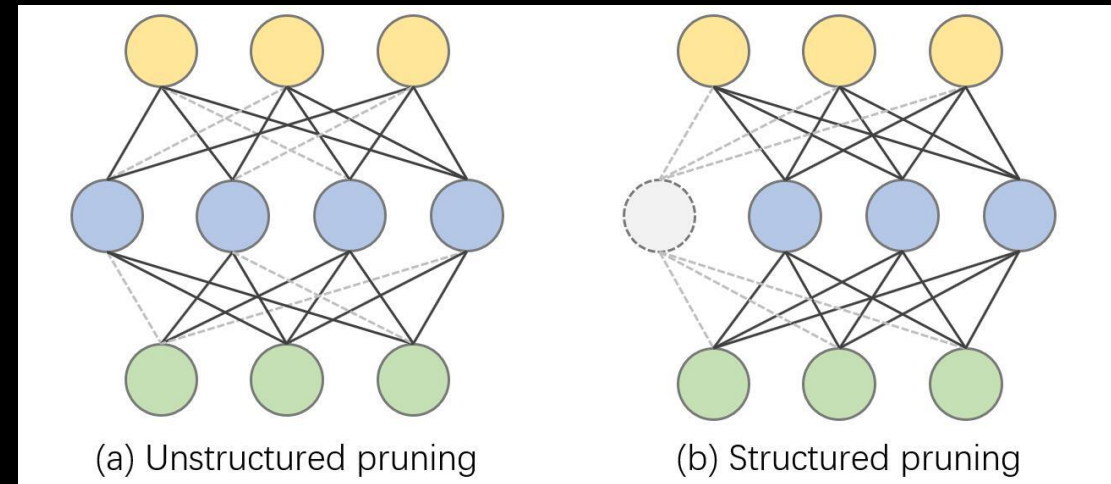
PRUNING TECHNIQUES

- Sparsity is a model compression technique where near-zero elements in a matrix are replaced with zeros. This creates a sparse matrix, storing only non-zero values and their positions, resulting in significant memory savings compared to a full matrix.



TYPES OF PRUNING

- Structured Pruning: removes entire components like neurons or layers, simplifying the model's structure while preserving its overall shape. This method provides more control and scales better for large models compared to unstructured pruning.
- Unstructured Pruning : targets individual weights or neurons, creating an irregular sparse model. While simpler, it requires specialized techniques to handle and can be less efficient for large-scale models.



EXAMPLE IMPLEMENTATION : PRUNING

Before

```
fc3 weights:
tensor([[[-0.0727, -0.1643,  0.0144, -0.1935, -0.1697, -0.0771,  0.1008,  0.0208,
          0.1197, -0.0203]])
```

After

```
fc3 weights:
tensor([[[-0.0727, -0.1643,  0.0000, -0.1935, -0.1697, -0.0771,  0.1008,  0.0000,
          0.1197, -0.0000]])
```

```
import torch
import torch.nn as nn
import torch.nn.utils.prune as prune

# Define a simple feedforward neural network
class PrunableNN(nn.Module):
    def __init__(self):
        super(PrunableNN, self).__init__()
        self.fc1 = nn.Linear(10, 20)
        self.fc2 = nn.Linear(20, 10)
        self.fc3 = nn.Linear(10, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

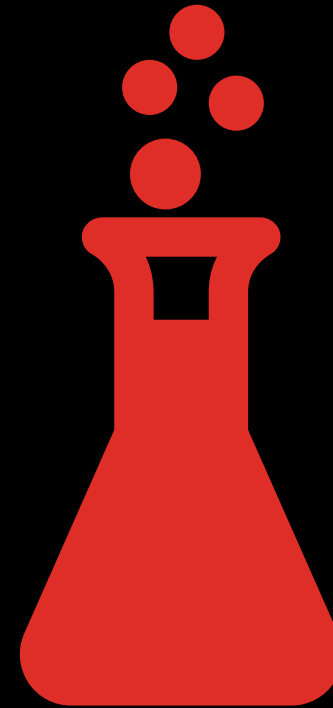
def apply_pruning(model, pruning_percent):
    # Apply weight pruning to linear layers
    for name, module in model.named_modules():
        if isinstance(module, nn.Linear):
            prune.l1_unstructured(module, name='weight', amount=pruning_percent)
            if module.bias is not None:
                prune.l1_unstructured(module, name='bias', amount=pruning_percent)

def remove_pruning(model):
    # Remove pruning reparametrization to finalize the model
    for name, module in model.named_modules():
        if isinstance(module, nn.Linear):
            prune.remove(module, 'weight')
            if module.bias is not None:
                prune.remove(module, 'bias')

def print_model_weights(model):
    for name, module in model.named_modules():
        if isinstance(module, nn.Linear):
            print(f"\n{name} weights:")
            print(module.weight.data)
            if module.bias is not None:
                print(f"{name} bias:")
                print(module.bias.data)
```

KNOWLEDGE DISTILLATION

- Knowledge distillation is about transferring knowledge from a large model to a smaller one. This process involves training a smaller model (called a student) to mimic the behavior of a larger model (a teacher).
- Response-based distillation: The case where the student model learns to produce similar output responses as teacher model.
- Feature-based distillation :In this case , the student model learns to reproduce similar intermediate layers.
- Relation-based distillation : The student model learns to reproduce the interaction between layers.

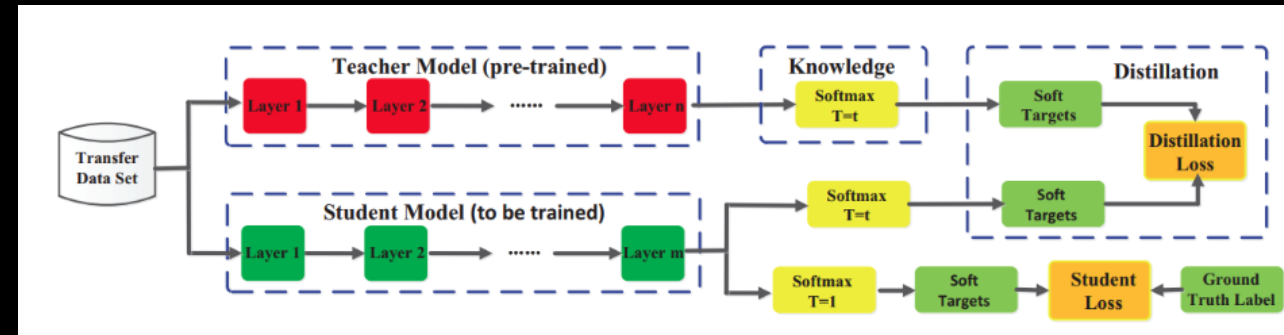


KNOWLEDGE DISTILLATION

Successful examples of distilled models include DistilBERT, which compresses a BERT model by 40% while retaining 97% of its language understanding capabilities at a speed 60% faster.

- As seen in diagram following are the common steps :

- The student learns by minimizing the difference between its outputs and the teacher's outputs. Additionally, the student can be trained to match the correct answers (ground truth).
- The teacher model can transfer knowledge to the student model by sharing information from its final layer (logits) or intermediate hidden layers (activations)
- Another approach : Use data synthesized by the teacher for supervised training of a student LLM



EXAMPLE KNOWLEDGE DISTILLATION

```
import torch.nn.functional as F

def knowledge_distillation(student_model, teacher_model, data_loader, optimizer, temperature=5.0, alpha=0.7,
                           num_epochs=5):
    """
    Performs knowledge distillation to train a student model using a teacher model.

    Args:
        student_model (nn.Module): The smaller, student model to be trained.
        teacher_model (nn.Module): The larger, pre-trained teacher model.
        data_loader (DataLoader): DataLoader for the training dataset.
        optimizer (Optimizer): Optimizer for the student model.
        temperature (float): Temperature to soften the logits from the teacher model.
        alpha (float): Weighting factor for the distillation loss vs. original loss.
        num_epochs (int): Number of epochs to train the student model.

    Returns:
        None
    """
    teacher_model.eval() # Set the teacher model to evaluation mode

    for epoch in range(num_epochs):
        student_model.train() # Set the student model to training mode
        for batch_idx, (data, target) in enumerate(data_loader):
            optimizer.zero_grad()

            # Forward pass through both teacher and student models
            teacher_logits = teacher_model(data).detach() # Teacher's output (no gradient)
            student_logits = student_model(data) # Student's output

            # Compute the distillation loss
            distill_loss = F.kl_div(
                F.log_softmax(student_logits / temperature, dim=1),
                F.softmax(teacher_logits / temperature, dim=1),
                reduction='batchmean'
            ) * (temperature * temperature)

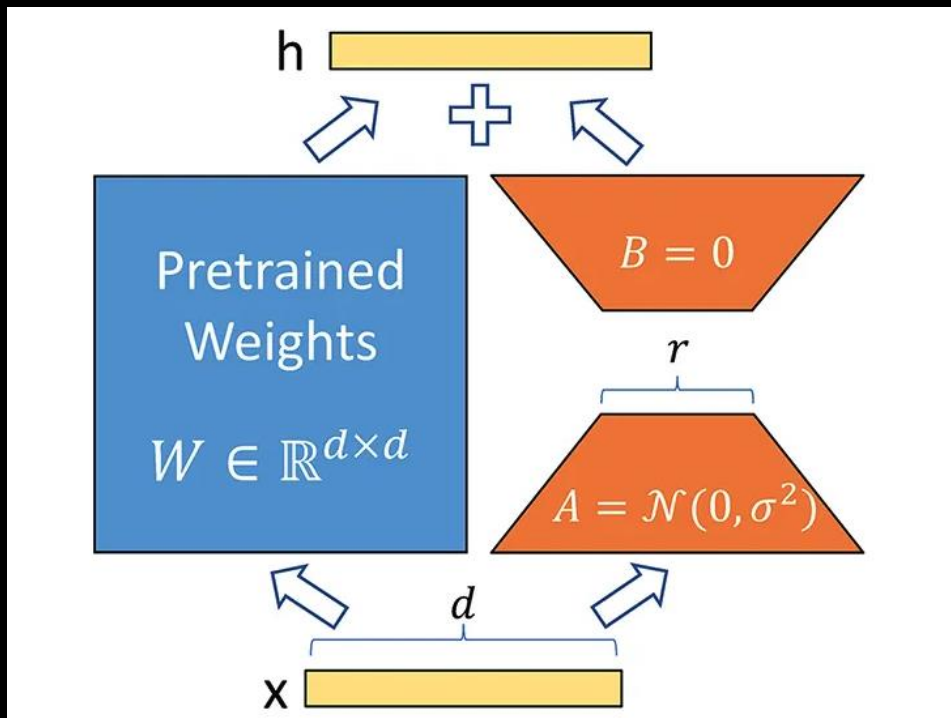
            # Compute the standard cross-entropy loss
            student_loss = F.cross_entropy(student_logits, target)

            # Combine losses
            loss = alpha * distill_loss + (1. - alpha) * student_loss

            # Backward pass and optimization
            loss.backward()
            optimizer.step()

        # Print loss for each epoch
        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {loss.item()}")
```

LOW-RANK APPROXIMATION



- Low-rank decomposition comes from the fact that neural network weight matrices can be approximated by products of low-dimension matrices
- The core idea is to approximate a large matrix (e.g., weight matrices in neural networks) with the product of two smaller matrices, thereby reducing the number of parameters and computational costs.
- Singular Value Decomposition (SVD)
 - Decomposes a matrix into three matrices: U, Σ, V^T .
- Truncated SVD :
 - Reduces matrix rank by discarding smaller singular values.
- Randomized SVD :
 - Faster approximation for large matrices.
- Adaptive Rank Selection:
 - Adjust matrix size based on different parts of the model.

EXAMPLE LRD

Original Weight Matrix:

```
tensor([[ -0.7256, -0.6537,  0.1875, -1.0692],
        [ 1.0797,  2.3320, -1.0701,  0.4611],
        [ 1.9490,  1.2280,  0.5906, -0.0383],
        [-0.8112,  0.5152,  0.6954, -0.8825],
        [-0.1008, -0.6321, -0.3549, -0.8376],
        [ 0.1346,  1.4949,  0.5132,  2.1478]])
```

Approximated Weight Matrix (Rank = 2):

```
tensor([[ -4.6462e-01, -9.4274e-01,  2.9598e-04, -9.0147e-01],
        [ 1.6215e+00,  1.9714e+00, -2.8258e-01,  5.5927e-01],
        [ 1.4765e+00,  1.4967e+00, -3.2101e-01, -7.6073e-02],
        [-1.0691e-01, -3.0761e-01, -1.9291e-02, -3.8531e-01],
        [-1.0658e-01, -5.6962e-01, -7.5369e-02, -8.9991e-01],
        [ 2.0681e-01,  1.3619e+00,  2.0103e-01,  2.2496e+00]])
```

```
import torch
```

```
def low_rank_decomposition(weight_matrix, rank):
```

```
    # Perform Singular Value Decomposition (SVD)
```

```
    U, S, V = torch.svd(weight_matrix)
```

```
    # Keep only the top 'rank' singular values and vectors
```

```
    U = U[:, :rank]
```

```
    S = S[:rank]
```

```
    V = V[:, :rank]
```

```
    # Reconstruct the low-rank approximation
```

```
    W_approx = torch.mm(U, torch.mm(torch.diag(S), V.t()))
```

```
    return W_approx
```

```
# Example usage
```

```
# Create a random weight matrix of size 6x4
```

```
original_weight = torch.randn(6, 4)
```

```
# Specify the rank for decomposition
```

```
rank = 2
```

```
# Apply low-rank decomposition
```

```
approximated_weight = low_rank_decomposition(original_weight, rank)
```

```
# Display the results
```

```
print("Original Weight Matrix:\n", original_weight)
```

```
print("\nApproximated Weight Matrix (Rank = {}):\n".format(rank), approximated_weight)
```



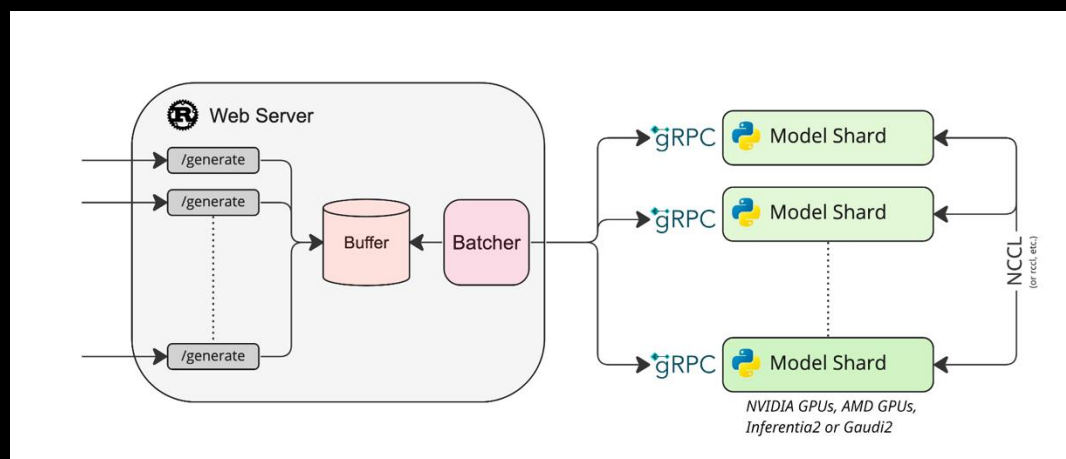
LLM INFERENCE OPTIMIZATION

- Optimizing inference is crucial to ensure that the models can serve predictions efficiently, especially in production environments where latency, throughput, and resource usage are critical.

Top approaches to do these are :

- Batching
- Model and Data Parallelism
- Speculative Inference

BATCHING

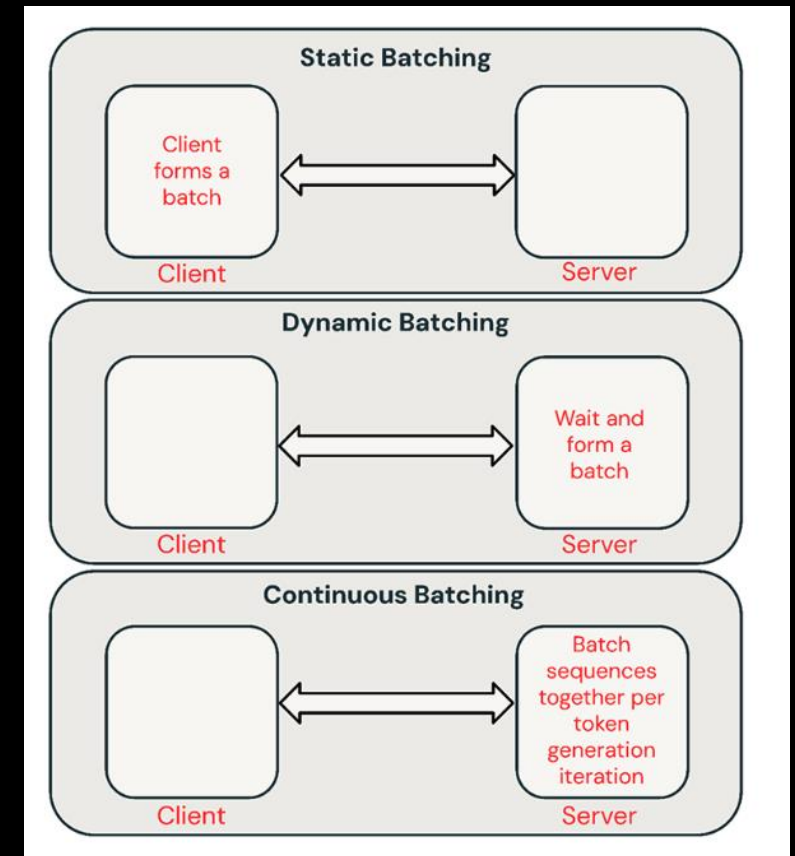


Example of batching in an Inference Server

- Batching refers to the technique of processing multiple text inputs simultaneously in a single forward pass through the model.
- This approach leverages the model's ability to handle multiple inputs at once, thereby improving efficiency and throughput.

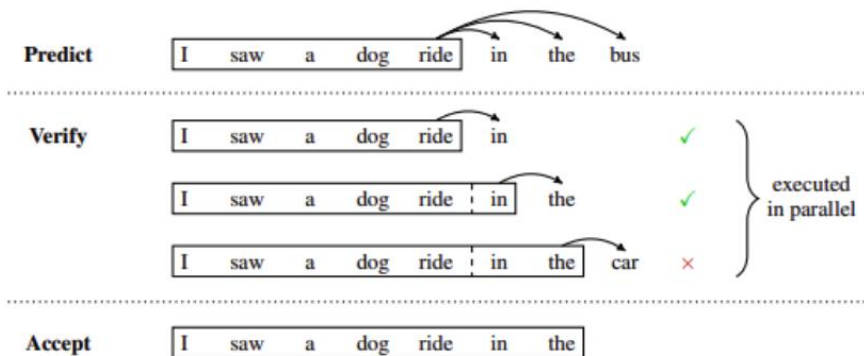
BATCHING

- Naïve Batching : involves grouping a fixed number of requests or inputs together and processing them in a single batch. Once the batch is processed, a new batch is formed for the next set of requests.
- Dynamic batching: Prompts are batched together on the fly inside the server. Typically, this method performs worse than static batching but can get close to optimal if responses are short or of uniform length. Does not work well when requests have different parameters
- Continuous Batching: is a more dynamic approach where requests are continuously aggregated into batches and processed as soon as they reach a certain threshold or within a specified time window



SPECULATIVE INFERENCE

- Aliases : speculative sampling, assisted generation, or blockwise parallel decoding.
- Each token generated depends on all the preceding tokens for context. As a result, in typical execution, generating multiple tokens from the same sequence in parallel is not feasible—you must generate the n th token before moving on to the $n+1$ token
- An Inference optimisation technique by predicting multiple potential next tokens in parallel, rather than sequentially processing them one at a time.



SPECULATIVE INFERENCE

- Speeds up token generation by predicting multiple possible next tokens simultaneously, rather than waiting for one token to be generated before predicting the next.
- The main model then verifies these predictions, selecting the correct token from the generated possibilities. This approach aims to improve efficiency by parallelizing parts of the token generation process.

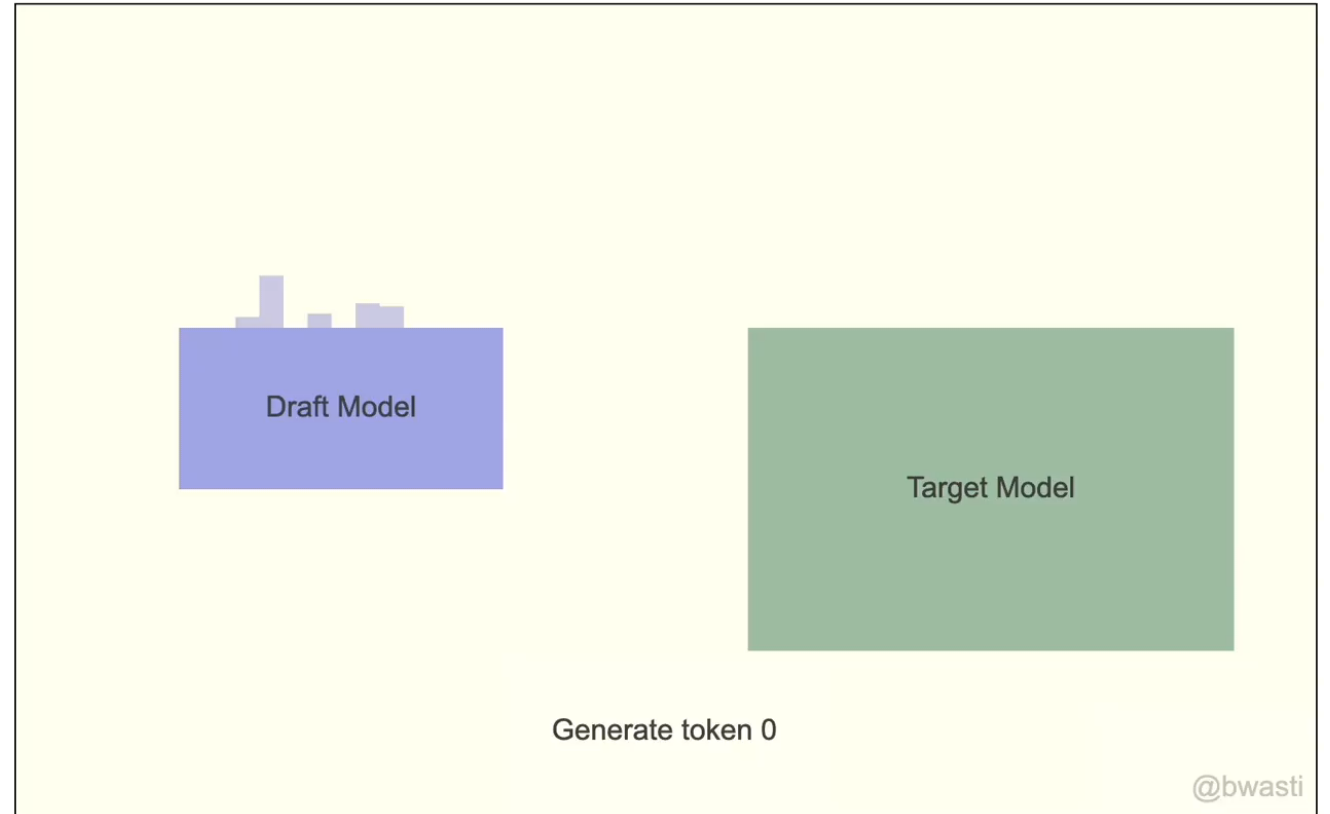
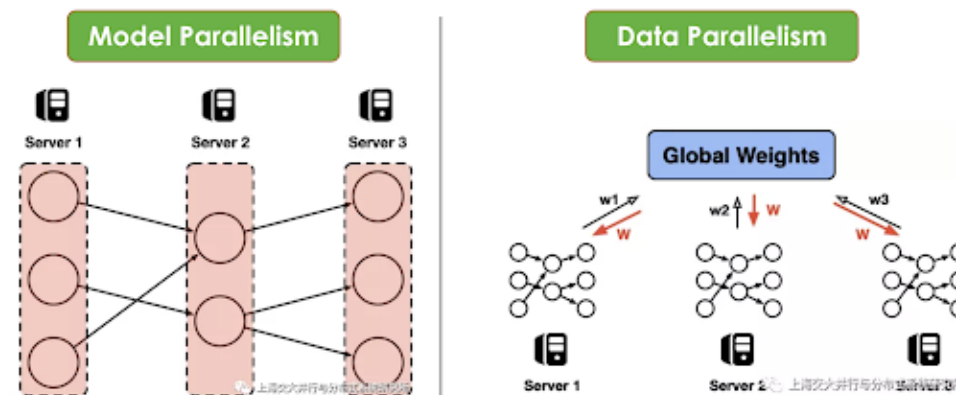


Image source : <https://x.com/i/status/1701929917514981712>

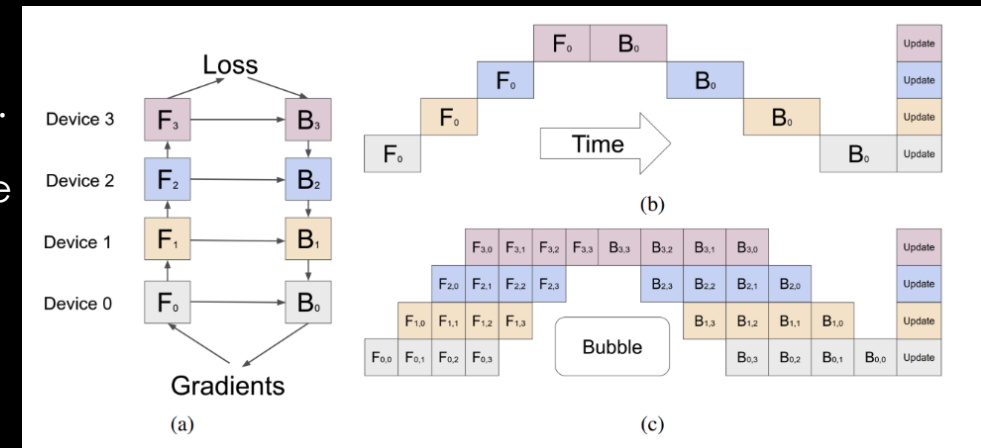
MODEL AND DATA PARALLELISM

- **Model Parallelism:** Splits the model across devices, distributing the layers or segments to enable training and inference of very large models.
- **Tensor Parallelism:** Splits the tensors within a layer across devices, allowing for fine-grained parallel computation within layers.
- **Sequence Parallelism:** Splits the input sequence itself across devices, enabling parallel processing of different sequence segments, useful for handling long sequences.



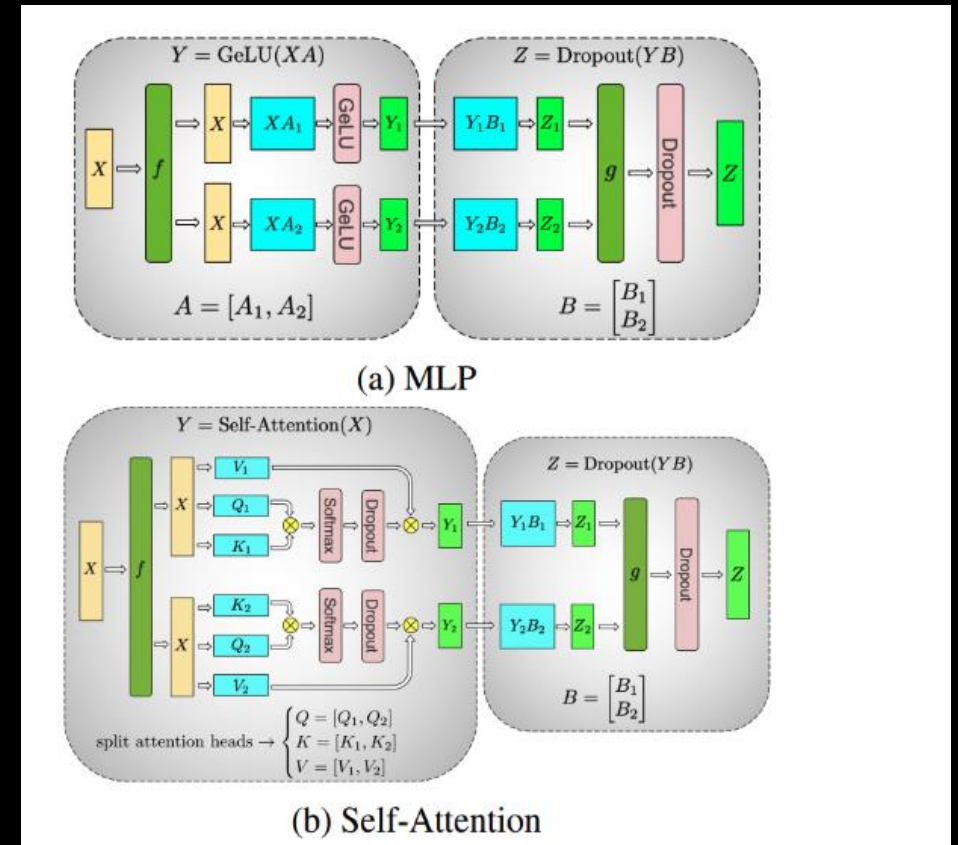
MODEL PARALLELISM: KEY FACTS

- **Pipeline Parallelism:** The model is vertically sharded into chunks, each comprising a subset of layers executed on a separate device. Outputs from one device are passed to the next, which continues executing the next chunk.
- **Memory Efficiency:** Memory required for storing model weights on each device is effectively reduced by the number of shards.
- **Sequential Processing Limitation:** Some devices may remain idle while waiting for outputs from previous layers, leading to inefficiencies or "pipeline bubbles".
- **Pipeline Bubbles:** Idle periods in the forward and backward passes, shown as empty spaces in pipeline visualizations, indicate underutilized devices.
- **Microbatching Mitigation:** The global batch size is split into smaller sub-batches, processed sequentially, and gradients are accumulated at the end. This reduces, but doesn't fully eliminate, pipeline bubbles.



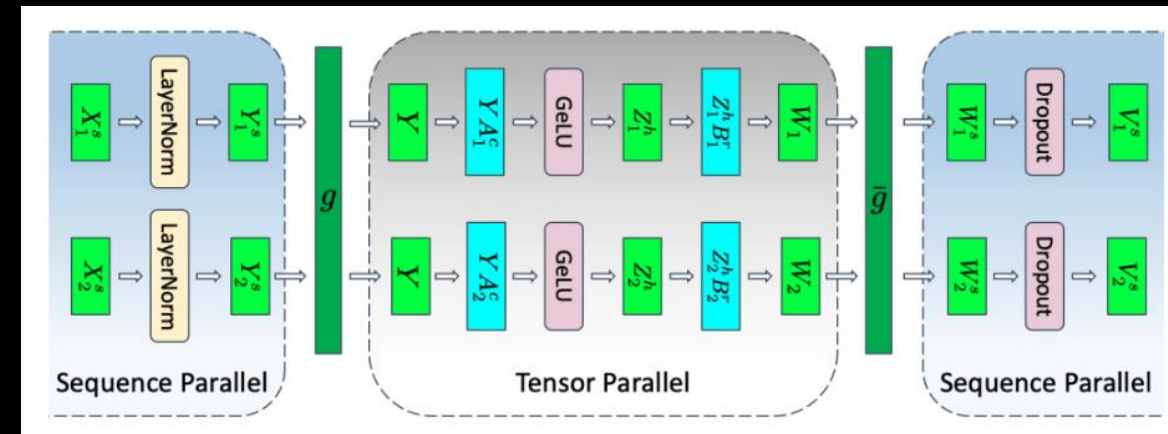
TENSOR PARALLELISM

- **Tensor Parallelism:** Horizontally shards individual layers of the model into smaller, independent computation blocks that run on different devices.
- **Key Components:** Attention blocks and multi-layer perceptron (MLP) layers in transformers benefit most from tensor parallelism.
- **Multi-head Attention:** Each attention head or group of heads can be assigned to a different device for independent, parallel computation.
- **Two-way Tensor Parallelism (MLP Example):** The weight matrix of a two-layer MLP is split into two parts (A_1 and A_2), allowing independent computation (XA_1 and XA_2) on different devices, halving the memory needed for weights on each device. Outputs are combined in the second layer using a reduction operation.
- **Two-way Tensor Parallelism (Self-Attention Example):** In the self-attention layer, the naturally parallel multiple attention heads are distributed across devices for parallel execution.



SEQUENCE PARALLELISM

- **Sequence Parallelism:** Involves partitioning operations like LayerNorm and Dropout along the input sequence dimension.
- **Independence Across Sequence:** These operations are independent across the sequence, allowing them to be distributed across multiple devices.
- **Memory Efficiency:** Enhances memory efficiency by reducing the need for redundant computations.
- **Complementary to Tensor Parallelism:** Useful for managing memory in large models and works well alongside tensor parallelism.



EXAMPLE IMPLEMENTATION : MODEL PARALLELISM

```
# Define a simplified Transformer block with model parallelism
class ParallelTransformerBlock(nn.Module):
    def __init__(self):
        super(ParallelTransformerBlock, self).__init__()
        # Place the attention layer on GPU 0
        self.attention = nn.MultiheadAttention(embed_dim=512, num_heads=8).to('cuda:0')
        # Place the feedforward network on GPU 1
        self.feedforward = nn.Sequential(
            nn.Linear(512, 2048),
            nn.ReLU(),
            nn.Linear(2048, 512)
        ).to('cuda:0')
        # LayerNorm on GPU 1 (can also be placed on another GPU)
        self.norm1 = nn.LayerNorm(512).to('cuda:0')
        self.norm2 = nn.LayerNorm(512).to('cuda:0')

    def forward(self, x):
        # Attention layer on GPU 0
        x = x.to('cuda:0')
        attn_output, _ = self.attention(x, x, x)
        attn_output = attn_output.to('cuda:0')

        # Feedforward network on GPU 1
        x = self.norm1(attn_output + x)
        ff_output = self.feedforward(x)
        x = self.norm2(ff_output + x)
        return x

# Define a simple LLM with several transformer blocks
class SimpleLLM(nn.Module):
    def __init__(self, num_blocks=6):
        super(SimpleLLM, self).__init__()
        self.blocks = nn.ModuleList([ParallelTransformerBlock() for _ in range(num_blocks)])
        self.output_layer = nn.Linear(512, 10000).to('cuda:0') # Output layer on GPU 1

    def forward(self, x):
        for block in self.blocks:
            x = block(x)
        x = self.output_layer(x)
        return x

# Create the model and optimizer
model = SimpleLLM()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Dummy input tensor
input_data = torch.randn(10, 32, 512) # (sequence_length, batch_size, embedding_dim)

# Forward pass
output = model(input_data)
```

```
# Forward pass
output = model(input_data)

# Dummy target tensor
target = torch.randint(0, 10000, (10, 32)).to('cuda:0')

# Loss calculation
criterion = nn.CrossEntropyLoss()
loss = criterion(output.view(-1, 10000), target.view(-1))

# Backward pass and optimization
optimizer.zero_grad()
loss.backward()
optimizer.step()

print("Completed a forward and backward pass using model parallelism in a simple LLM.")
```

WHY LLM COMPRESSION ?

High Computational & Energy Costs

- LLMs demand significant computational power and energy, making them expensive to operate.
- **Compression Benefit:** Reduces costs and energy consumption.

Deployment on Resource-Limited Devices

- Full-scale LLMs are impractical for mobile and IoT devices.
- **Compression Benefit:** Enables deployment on a wider range of devices.

Improved Latency & Real-Time Performance

- Large models introduce latency in real-time applications.
- **Compression Benefit:** Enhances inference speed and reduces latency.

Memory Efficiency & Scalability

- Large models exceed memory limits, creating bottlenecks.
- **Compression Benefit:** Fits models within hardware constraints, improving scalability.

Environmental Impact

- LLMs contribute to high energy consumption and carbon footprint.
- **Compression Benefit:** Promotes energy-efficient AI systems.

Accessibility & Democratization of AI

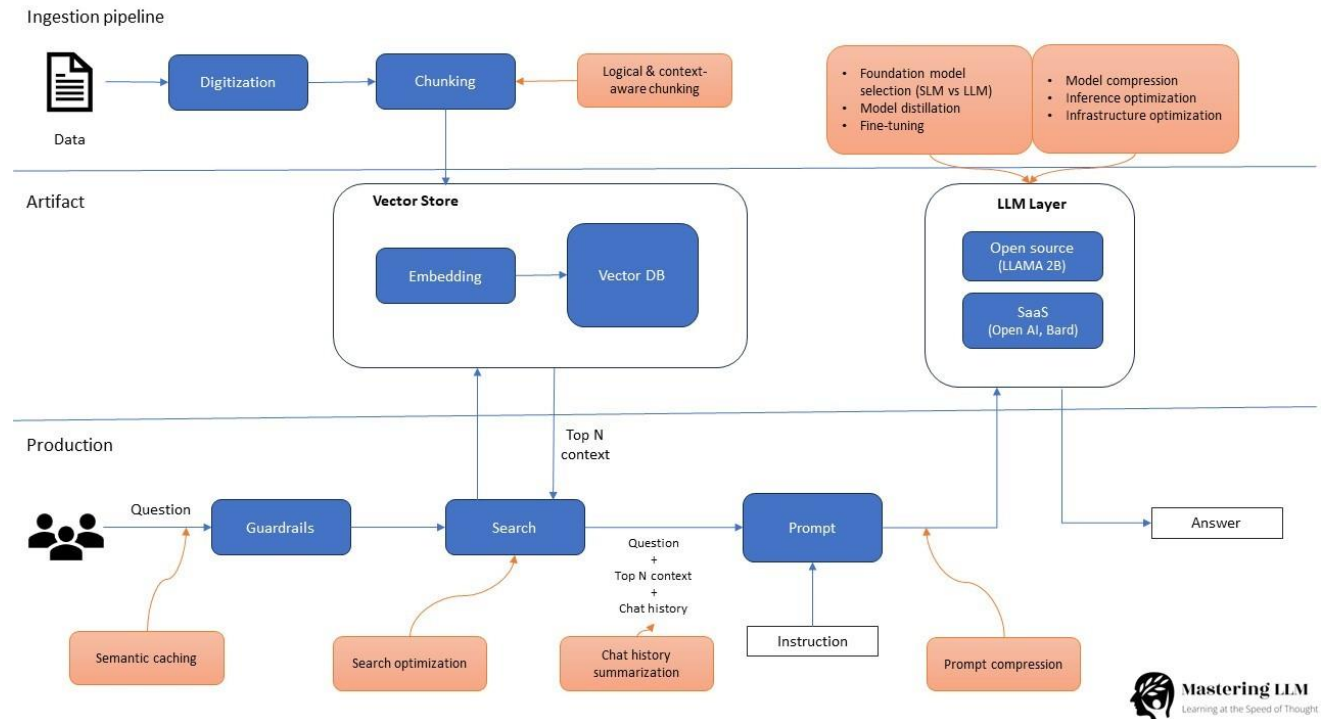
- High computational requirements limit access to LLMs.
- **Compression Benefit:** Makes LLMs more affordable and accessible.

Maintaining Performance with Reduced Complexity

- LLMs are complex and difficult to optimize.
- **Compression Benefit:** Simplifies deployment while preserving performance.

CASE STUDY

- **RAG System:** A large language model (e.g., GPT/Llama3) is used to generate responses based on retrieved information from a customer support knowledge base.
- **Challenges:** High latency due to large model size, potential for hallucinations, and high computational costs.
- **User Experience:** Users might experience delays in receiving responses, and the chatbot's answers could be inaccurate or irrelevant.



IMPROVEMENTS

1. Quantization:

Reduces model size by 75% and speeds up inference by 2x to 4x.

2. Pruning:

Cuts model size by up to 40% and improves inference speed by 1.5x to 2x.

3. Knowledge Distillation:

Results in a model 50% smaller and up to 2x faster, with similar performance.

4. Low-Rank Approximation:

Reduces parameters by up to 30% and speeds up inference by 1.5x to 2x.

5. LLM Optimization Techniques:

•Batching:

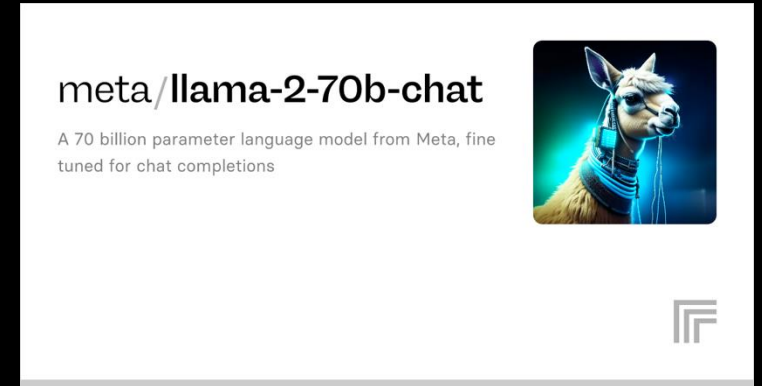
- Increases throughput by up to 4x and cuts latency by 50%.

•Model and Data Parallelism:

- Handles models 2x to 10x larger, reducing processing time.

•Speculative Inference:

- Reduces sequence generation time by up to 30% and speeds up response.



KEY TAKEAWAYS

- Understanding the challenges of operationalizing Generative AI is crucial for building efficient systems.
- Model compression techniques like quantization, pruning, knowledge distillation, and low-rank approximation significantly reduce LLM size and computational costs.
- Optimization strategies such as batching, model and data parallelism, and speculative inference enhance inference speed and throughput.
- Combining compression and optimization techniques is essential for deploying LLMs in resource-constrained environments.
- Effective RAG systems require careful consideration of both model and inference optimization.